The F# 3.1 Language Specification

Note: This documentation is the specification of the 3.1 release of F# for releases and updates made in 2013-15.

Discrepancies may exist between this specification and the 3.1 implementation. Some of these are noted as comments in this document. If you find further discrepancies please contact us and we will gladly address the issue in future releases of this specification. The F# team is always grateful for feedback on this specification, and on both the design and implementation of F#. You can submit feedback by opening issues, comments and pull requests at https://github.com/fsharp/fsfoundation/tree/gh-pages/specs/language-spec.

The latest version of this specification can be found at fsharp.org. Many thanks to the F# user community for their helpful feedback on the document so far.

Certain parts of this specification refer to the C# 4.0, Unicode, and IEEE specifications.

Authors: Don Syme, with assistance from Anar Alimov, Keith Battocchi, Jomo Fisher, Michael Hale, Jack Hu, Luke Hoban, Tao Liu, Dmitry Lomov, James Margetson, Brian McNamara, Joe Pamer, Penny Orwick, Daniel Quirk, Kevin Ransom, Chris Smith, Matteo Taveggia, Donna Malayeri, Wonseok Chae, Uladzimir Matsveyeu, Lincoln Atkinson, and others.

Notice

Document Updates:

- Updates for F# 3.1 and type providers, January 2016
- Edits to change version numbers for F# 3.1, May 2014
- Initial updates for F# 3.1, June 2013 (see online description of language updates)
- Updated to F# 3.0, September 2012
- Updated with formatting changes, April 2012
- Updated with grammar summary, December 2011
- Updated with glossary, index, and style corrections, February 2011
- Updated with glossary, index, and style corrections, August 2010

Table of Contents

1. Introduction

F# is a scalable, succinct, type-safe, type-inferred, efficiently executing functional/imperative/object-oriented programming language. It aims to be the premier typed functional programming language for the .NET framework and other implementations of the Ecma 335 Common Language Infrastructure (CLI) specification. F# was partly inspired by the OCaml language and shares some common core constructs with it.

1.1 A First Program

Over the next few sections, we will look at some small F# programs, describing some important aspects of F# along the way. As an introduction to F#, consider the following program:

```
let numbers = [ 1 .. 10 ]

let square x = x * x

let squares = List.map square numbers

printfn "N^2 = %A" squares
```

To explore this program, you can:

- Compile it as a project in a development environment such as Visual Studio.
- Manually invoke the F# command line compiler fsc.exe.
- Use F# Interactive, the dynamic compiler that is part of the F# distribution.

1.1.1 Lightweight Syntax

The F# language uses simplified, indentation-aware syntactic constructs known as lightweight syntax. The lines of the sample program in the previous section form a sequence of declarations and are aligned on the same column. For example, the two lines in the following code are two separate declarations:

```
let squares = List.map square numbers

printfn "N^2 = %A" squares
```

Lightweight syntax applies to all the major constructs of the F# syntax. In the next example, the code is incorrectly aligned. The declaration starts in the first line and continues to the second and subsequent lines, so those lines must be indented to the same column under the first line:

```
let computeDerivative f x =
    let p1 = f (x - 0.05)
  let p2 = f (x + 0.05)
```

```
        (p2 - p1) / 0.1
```

The following shows the correct alignment:

```
let computeDerivative f x =
    let p1 = f (x - 0.05)
    let p2 = f (x + 0.05)
    (p2 - p1) / 0.1
```

The use of lightweight syntax is the default for all F# code in files with the extension `.fs`, `.fsx`, `.fsi`, or `.fsscript`.

1.1.2 Making Data Simple

The first line in our sample simply declares a list of numbers from one through ten.

```
let numbers = [1 .. 10]
```

An F# list is an immutable linked list, which is a type of data used extensively in functional programming. Some operators that are related to lists include `::` to add an item to the front of a list and `@` to concatenate two lists. If we try these operators in F# Interactive, we see the following results:

```
> let vowels = ['e'; 'i'; 'o'; 'u'];;
val vowels: char list = ['e'; 'i'; 'o'; 'u']

> ['a'] @ vowels;;
val it: char list = ['a'; 'e'; 'i'; 'o'; 'u']

> vowels @ ['y'];;
val it: char list = ['e'; 'i'; 'o'; 'u'; 'y']
```

Note that double semicolons delimit lines in F# Interactive, and that F# Interactive prefaces the result with *val* to indicate that the result is an immutable value, rather than a variable.

F# supports several other highly effective techniques to simplify the process of modeling and manipulating data such as tuples, options, records, unions, and sequence expressions. A tuple is an ordered collection of values that is treated as an atomic unit. In many languages, if you want to pass around a group of related values as a single entity, you need to create a named type, such as a class or record, to store these values. A tuple allows you to keep things organized by grouping related values together, without introducing a new type.

To define a tuple, you separate the individual components with commas.

```
> let tuple = (1, false, "text");;
val tuple : int * bool * string = (1, false, "text")

> let getNumberInfo (x : int) = (x, x.ToString(), x * x);;
val getNumberInfo : int -> int * string * int

> getNumberInfo 42;;
```

```
val it : int * string * int = (42, "42", 1764)
```

A key concept in F# is *immutability*. Tuples and lists are some of the many types in F# that are immutable, and indeed most things in F# are immutable by default. Immutability means that once a value is created and given a name, the value associated with the name cannot be changed. Immutability has several benefits. Most notably, it prevents many classes of bugs, and immutable data is inherently thread-safe, which makes the process of parallelizing code simpler.

1.1.3 Making Types Simple

The next line of the sample program defines a function called `square`, which squares its input.

```
let square x = x * x
```

Most statically-typed languages require that you specify type information for a function declaration. However, F# typically infers this type information for you. This process is referred to as *type inference*.

From the function signature, F# knows that `square` takes a single parameter named `x` and that the function returns `x * x`. The last thing evaluated in an F# function body is the return value; hence there is no "return" keyword here. Many primitive types support the multiplication (`*`) operator (such as `byte`, `uint64`, and `double`); however, for arithmetic operations, F# infers the type `int` (a signed 32-bit integer) by default.

Although F# can typically infer types on your behalf, occasionally you must provide explicit type annotations in F# code. For example, the following code uses a type annotation for one of the parameters to tell the compiler the type of the input.

```
> let concat (x : string) y = x + y;;
val concat : string -> string -> string
```

Because `x` is stated to be of type `string`, and the only version of the `+` operator that accepts a left-hand argument of type `string` also takes a `string` as the right-hand argument, the F# compiler infers that the parameter `y` must also be a string. Thus, the result of `x + y` is the concatenation of the strings. Without the type annotation, the F# compiler would not have known which version of the `+` operator was intended and would have assumed `int` data by default.

The process of type inference also applies *automatic generalization* to declarations. This automatically makes code *generic* when possible, which means the code can be used on many types of data. For example, the following code defines a function that returns a new tuple in which the two values are swapped:

```
> let swap (x, y) = (y, x);;
val swap : 'a * 'b -> 'b * 'a

> swap (1, 2);;
val it : int * int = (2, 1)

> swap ("you", true);;
val it : bool * string = (true,"you")
```

Here the function `swap` is generic, and `'a` and `'b` represent *type variables*, which are placeholders for types in generic code. Type inference and automatic generalization greatly simplify the process of writing reusable code fragments.

1.1.4 Functional Programming

Continuing with the sample, we have a list of integers named `numbers`, and the `square` function, and we want to create a new list in which each item is the result of a call to our function. This is called *mapping* our function over each item in the list. The F# library function `List.map` does just that:

```
let squares = List.map square numbers
```

Consider another example:

```
> List.map (fun x -> x % 2 = 0) [1 .. 5];;

val it : bool list
= [false; true; false; true; false]
```

The code `(fun x -> x % 2 = 0)` defines an anonymous function, called a *function expression*, that takes a single parameter `x` and returns the result `x % 2 = 0`, which is a Boolean value that indicates whether `x` is even. The `->` symbol separates the argument list (`x`) from the function body (`x % 2 = 0`).

Both of these examples pass a function as a parameter to another function—the first parameter to `List.map` is itself another function. Using functions as *function values* is a hallmark of functional programming.

Another tool for data transformation and analysis is *pattern matching*. This powerful switch construct allows you to branch control flow and to bind new values. For example, we can match an F# list against a sequence of list elements.

```
let checkList alist =
    match alist with
    | [] -> 0
    | [a] -> 1
    | [a; b] -> 2
    | [a; b; c] -> 3
    | _ -> failwith "List is too big!"
```

In this example, `alist` is compared with each potentially matching pattern of elements. When `alist` matches a pattern, the result expression is evaluated and is returned as the value of the match expression. Here, the `->` operator separates a pattern from the result that a match returns.

Pattern matching can also be used as a control construct—for example, by using a pattern that performs a dynamic type test:

```
let getType (x : obj) =
    match x with
    | :? string           -> "x is a string"
    | :? int              -> "x is an int"
    | :? System.Exception -> "x is an exception"
```

The `:?` operator returns true if the value matches the specified type, so if `x` is a string, `getType` returns "`x is a string`".

Function values can also be combined with the *pipeline operator*, `|>`. For example, given these functions:

```
let square x          = x * x
let toStr (x : int)  = x.ToString()
let reverse (x : string) = new System.String(Array.rev
(x.ToCharArray()))
```

We can use the functions as values in a pipeline:

```
> let result = 32 |> square |> toStr |> reverse;;
val it : string = "4201"
```

Pipelining demonstrates one way in which F# supports *compositionality*, a key concept in functional programming. The pipeline operator simplifies the process of writing compositional code where the result of one function is passed into the next.

1.1.5 Imperative Programming

The next line of the sample program prints text in the console window.

```
printfn "N^2 = %A" squares
```

The F# library function `printfn` is a simple and type-safe way to print text in the console window. Consider this example, which prints an integer, a floating-point number, and a string:

```
> printfn "%d * %f = %s" 5 0.75 ((5.0 * 0.75).ToString());;
5 * 0.750000 = 3.75
val it : unit = ()
```

The format specifiers `%d`, `%f`, and `%s` are placeholders for integers, floats, and strings. The `%A` format can be used to print arbitrary data types (including lists).

The `printfn` function is an example of *imperative programming*, which means calling functions for their side effects. Other commonly used imperative programming techniques include arrays and dictionaries (also called hash tables). F# programs typically use a mixture of functional and imperative techniques.

1.1.6 .NET Interoperability and CLI Fidelity

The Common Language Infrastructure (CLI) function `System.Console.ReadKey` to pause the program before the console window closes.

```
System.Console.ReadKey(true)
```

Because F# is built on top of CLI implementations, you can call any CLI library from F#. Furthermore, other CLI languages can easily use any F# components.

1.1.7 Parallel and Asynchronous Programming

F# is both a *parallel* and a *reactive* language. During execution, F# programs can have multiple parallel active evaluations and multiple pending reactions, such as callbacks and agents that wait to react to events and messages.

One way to write parallel and reactive F# programs is to use F# *async* expressions. For example, the code below is similar to the original program in §1.1 except that it computes the Fibonacci function (using a technique that will take some time) and schedules the computation of the numbers in parallel:

```
let rec fib x = if x < 2 then 1 else fib(x-1) + fib(x-2)

let fibs =
    Async.Parallel [ for i in 0..40 -> async { return fib(i) } ]
    |> Async.RunSynchronously

printfn "The Fibonacci numbers are %A" fibs

System.Console.ReadKey(true)
```

The preceding code sample shows multiple, parallel, CPU-bound computations.

F# is also a reactive language. The following example requests multiple web pages in parallel, reacts to the responses for each request, and finally returns the collected results.

```
open System
open System.IO
open System.Net

let http url =
    async { let req =  WebRequest.Create(Uri url)
            use! resp = req.AsyncGetResponse()
            use stream = resp.GetResponseStream()
            use reader = new StreamReader(stream)
            let contents = reader.ReadToEnd()
            return contents }

let sites = ["http://www.bing.com"; "http://www.google.com";
             "http://www.yahoo.com"; "http://www.search.com"]
```

```
let htmlOfSites =
    Async.Parallel [for site in sites -> http site ]
    |> Async.RunSynchronously
```

By using asynchronous workflows together with other CLI libraries, F# programs can implement parallel tasks, parallel I/O operations, and message-receiving agents.

1.1.8 Strong Typing for Floating-Point Code

F# applies type checking and type inference to floating-point-intensive domains through *units of measure inference and checking*. This feature allows you to type-check programs that manipulate floating-point numbers that represent physical and abstract quantities in a stronger way than other typed languages, without losing any performance in your compiled code. You can think of this feature as providing a type system for floating-point code.

Consider the following example:

```
[<Measure>] type kg
[<Measure>] type m
[<Measure>] type s

let gravityOnEarth = 9.81<m/s^2>
let heightOfTowerOfPisa = 55.86<m>
let speedOfImpact = sqrt(2.0 * gravityOnEarth * heightOfTowerOfPisa)
```

The `Measure` attribute tells F# that `kg`, `s`, and `m` are not really types in the usual sense of the word, but are used to build units of measure. Here `speedOfImpact` is inferred to have type `float<m/s>`.

1.1.9 Object-Oriented Programming and Code Organization

The sample program shown at the start of this chapter is a *script*. Although scripts are excellent for rapid prototyping, they are not suitable for larger software components. F# supports the transition from scripting to structured code through several techniques.

The most important of these is *object-oriented programming* through the use of *class type definitions*, *interface type definitions*, and *object expressions*. Object-oriented programming is a primary application programming interface (API) design technique for controlling the complexity of large software projects. For example, here is a class definition for an encoder/decoder object.

```
open System

/// Build an encoder/decoder object that maps characters to an
/// encoding and back. The encoding is specified by a sequence
/// of character pairs, for example, [('a','Z'); ('Z','a')]
type CharMapEncoder(symbols: seq<char*char>) =
    let swap (x, y) = (y, x)

    /// An immutable tree map for the encoding
    let fwd  = symbols |> Map.ofSeq
```

```
    /// An immutable tree map for the decoding
    let bwd  = symbols |> Seq.map swap |> Map.ofSeq

    let encode (s:string) =
       String [| for c in s -> if fwd.ContainsKey(c) then fwd.[c] else
c |]
    let decode (s:string) =
       String [| for c in s -> if bwd.ContainsKey(c) then bwd.[c] else
c |]

    /// Encode the input string
    member x.Encode(s) = encode s

    /// Decode the given string
    member x.Decode(s) = decode s
```

You can instantiate an object of this type as follows:

```
let rot13 (c:char) =
    char(int 'a' + ((int c - int 'a' + 13) % 26))
let encoder =
    CharMapEncoder( [for c in 'a'..'z' -> (c, rot13 c)])
```

And use the object as follows:

```
> "F# is fun!" |> encoder.Encode ;;
val it : string = "F# vf sha!"

> "F# is fun!" |> encoder.Encode |> encoder.Decode ;;
val it : String = "F# is fun!"
```

An interface type can encapsulate a family of object types:

```
open System

type IEncoding =
    abstract Encode : string -> string
    abstract Decode : string -> string
```

In this example, `IEncoding` is an interface type that includes both `Encode` and `Decode` object types.

Both object expressions and type definitions can implement interface types. For example, here is an object expression that implements the `IEncoding` interface type:

```
let nullEncoder =
    { new IEncoding with
        member x.Encode(s) = s
        member x.Decode(s) = s }
```

Modules are a simple way to encapsulate code during rapid prototyping when you do not want to spend the time to design a strict object-oriented type hierarchy. In the following example, we place a portion of our original script in a module.

```
module ApplicationLogic =
    let numbers n = [1 .. n]
    let square x = x * x
    let squares n = numbers n |> List.map square

printfn "Squares up to 5 = %A" (ApplicationLogic.squares 5)
printfn "Squares up to 10 = %A" (ApplicationLogic.squares 10)
System.Console.ReadKey(true)
```

Modules are also used in the F# library design to associate extra functionality with types. For example, `List.map` is a function in a module.

Other mechanisms aimed at supporting software engineering include *signatures*, which can be used to give explicit types to components, and *namespaces*, which serve as a way of organizing the name hierarchies for larger APIs.

1.1.10 Information-rich Programming

F# Information-rich programming addresses the trend toward greater availability of data, services, and information. The key to information-rich programming is to eliminate barriers to working with diverse information sources that are available on the Internet and in modern enterprise environments. Type providers and query expressions are a significant part of F# support for information-rich programming.

The F# Type Provider mechanism allows you to seamlessly incorporate, in a strongly typed manner, data and services from external sources. A *type provider* presents your program with new types and methods that are typically based on the schemas of external information sources. For example, an F# type provider for Structured Query Language (SQL) supplies types and methods that allow programmers to work directly with the tables of any SQL database:

```
// Add References to FSharp.Data.TypeProviders, System.Data, and
System.Data.Linq
type schema = SqlDataConnection<"Data Source=localhost;Integrated
Security=SSPI;">

let db = schema.GetDataContext()
```

The type provider connects to the database automatically and uses this for IntelliSense and type information.

Query expressions (added in F# 3.0) add the established power of query-based programming against SQL, Open Data Protocol (OData), and other structured or relational data sources. Query expressions provide support for Language-Integrated Query (LINQ) in F#, and several query operators enable you to construct more complex queries. For example, we can create a query to filter the customers in the data source:

```
let countOfCustomers =
    query { for customer in db.Customers do
            where (customer.LastName.StartsWith("N"))
            select (customer.FirstName, customer.LastName) }
```

Now it is easier than ever to access many important data sources—including enterprise, web, and cloud—by using a set of built-in type providers for SQL databases and web data protocols. Where necessary, you can create your own custom type providers or reference type providers that others have created. For example, assume your organization has a data service that provides a large and growing number of named data sets, each with its own stable data schema. You may choose to create a type provider that reads the schemas and presents the latest available data sets to the programmer in a strongly typed way.

1.2 Notational Conventions in This Specification

This specification describes the F# language by using a mixture of informal and semiformal techniques. All examples in this specification use lightweight syntax, unless otherwise specified.

Regular expressions are given in the usual notation, as shown in the table:

Notation	Meaning
`regexp+`	One or more occurrences
`regexp*`	Zero or more occurrences
`regexp?`	Zero or one occurrences
`[ char - char ]`	Range of ASCII characters
`[ ^ char - char ]`	Any characters except those in the range

Unicode character classes are referred to by their abbreviation as used in CLI libraries for regular expressions—for example, `\Lu` refers to any uppercase letter. The following characters are referred to using the indicated notation:

Character	Name	Notation
`\b`	backspace	ASCII/UTF-8/UTF-16/UTF-32 code 08
`\n`	newline	ASCII/UTF-8/UTF-16/UTF-32 code 10
`\r`	return	ASCII/UTF-8/UTF-16/UTF-32 code 13
`\t`	tab	ASCII/UTF-8/UTF-16/UTF-32 code 09

Strings of characters that are clearly not a regular expression are written verbatim. Therefore, the following string

```
abstract
```

matches precisely the characters `abstract`.

Where appropriate, apostrophes and quotation marks enclose symbols that are used in the specification of the grammar itself, such as `'<'` and `'|'`. For example, the following regular expression matches `(+)` or `(-)`:

```
'(' (+|-) ')'
```

This regular expression matches precisely the characters `#if`:

```
"#if"
```

Regular expressions are typically used to specify tokens.

```
token token-name =  regexp
```

In the grammar rules, the notation *element-name*$_{\text{opt}}$ indicates an optional element. The notation `...` indicates repetition of the preceding non-terminal construct and the separator token. For example, *expr* `','` `...` `','` *expr* means a sequence of one or more *expr* elements separated by commas.

2. Program Structure

The inputs to the F# compiler or the F# Interactive dynamic compiler consist of:

- Source code files, with extensions `.fs`, `.fsi`, `.fsx`, or `.fsscript.`
  - Files with extension `.fs` must conform to grammar element *`implementation-file`* in §12.1.
  - Files with extension `.fsi` must conform to grammar element *`signature-file`* in §12.2.
  - Files with extension `.fsx` or `.fsscript` must conform to grammar element *`script-file`* in §12.3.
- Script fragments (for F# Interactive). These must conform to grammar element *`script-fragment`*. Script fragments can be separated by `;;` tokens.
- Assembly references that are specified by command line arguments or interactive directives.
- Compilation parameters that are specified by command line arguments or interactive directives.
- Compiler directives such as `#time`.

The `COMPILED` compilation symbol is defined for input that the F# compiler has processed. The `INTERACTIVE` compilation symbol is defined for input that F# Interactive has processed.

Processing the source code portions of these inputs consists of the following steps:

1. **Decoding**. Each file and source code fragment is decoded into a stream of Unicode characters, as described in the C# specification, sections 2.3 and 2.4. The command-line options may specify a code page for this process.
2. **Tokenization**. The stream of Unicode characters is broken into a token stream by the lexical analysis described in §3.
3. **Lexical Filtering**. The token stream is filtered by a state machine that implements the rules described in §15. Those rules describe how additional (artificial) tokens are inserted into the token stream and how some existing tokens are replaced with others to create an augmented token stream.
4. **Parsing**. The augmented token stream is parsed according to the grammar specification in this document.
5. **Importing**. The imported assembly references are resolved to F# or CLI assembly specifications, which are then imported. From the F# perspective, this results in the pre-definition of numerous namespace declaration groups (§12.1), types and type provider instances. The namespace declaration groups are then combined to form an initial name resolution environment (§14.1).

6. **Checking**. The results of parsing are checked one by one. Checking involves such procedures as Name Resolution (§14.1), Constraint Solving (§14.5), and Generalization (§14.6.7), as well as the application of other rules described in this specification.

   Type inference uses variables to represent unknowns in the type inference problem. The various checking processes maintain tables of context information including a name resolution environment and a set of current inference constraints. After the processing of a file or program fragment is complete, all such variables have been either generalized or resolved and the type inference environment is discarded.

7. **Elaboration**. One result of checking is an elaborated program fragment that contains elaborated declarations, expressions, and types. For most constructs, such as constants, control flow, and data expressions, the elaborated form is simple. Elaborated forms are used for evaluation, CLI reflection, and the F# expression trees that are returned by quoted expressions (§6.8).

8. **Execution**. Elaborated program fragments that are successfully checked are added to a collection of available program fragments. Each fragment has a static initializer. Static initializers are executed as described in (§12.5).

3. Lexical Analysis

Lexical analysis converts an input stream of Unicode characters into a stream of tokens by iteratively processing the stream. If more than one token can match a sequence of characters in the source file, lexical processing always forms the longest possible lexical element. Some tokens, such as `block-comment-start`, are discarded after processing as described later in this section.

3.1 Whitespace

Whitespace consists of spaces and newline characters.

```
regexp whitespace = ' '+
regexp newline = '\n' | '\r' '\n'
token whitespace-or-newline = whitespace | newline
```

Whitespace tokens `whitespace-or-newline` are discarded from the returned token stream.

3.2 Comments

Block comments are delimited by `(*` and `*)` and may be nested. Single-line comments begin with two backslashes (`//`) and extend to the end of the line.

```
token block-comment-start = "(*"
token block-comment-end = "*)"
token end-of-line-comment = "//" [^'\n' '\r']*
```

When the input stream matches a `block-comment-start` token, the subsequent text is tokenized recursively against the tokens that are described in §3 until a `block-comment-end` token is found. The intermediate tokens are discarded.

For example, comments can be nested, and strings that are embedded within comments are tokenized by the rules for `string`, `verbatim-string`, and `triple-quoted string`. In particular, strings that are embedded in comments are tokenized in their entirety, without considering closing `*)` marks. As a result of this rule, the following is a valid comment:

```
(* Here's a code snippet: let s = "*)" *)
```

However, the following construct, which was valid in F# 2.0, now produces a syntax error because a closing comment token `*)` followed by a triple-quoted mark is parsed as part of a string:

```
(* """ *)
```

For the purposes of this specification, comment tokens are discarded from the returned lexical stream. In practice, XML documentation tokens are `end-of-line-comments` that begin with ///. The delimiters are retained and are associated with the remaining elements to generate XML documentation.

3.3 Conditional Compilation

The lexical preprocessing directives `#if ident/#else/#endif` delimit conditional compilation sections. The following describes the grammar for such sections:

```
token if-directive = "#if" whitespace ident-text
token else-directive = "#else"
token endif-directive = "#endif"
```

A preprocessing directive always occupies a separate line of source code and always begins with a # character followed immediately by a preprocessing directive name, with no intervening whitespace. However, whitespace can appear before the # character. A source line that contains the `#if`, `#else`, or `#endif` directive can end with whitespace and a single-line comment. Multiple-line comments are not permitted on source lines that contain preprocessing directives.

If an *if-directive* token is matched during tokenization, text is recursively tokenized until a corresponding *else-directive* or *endif-directive*. If the compilation environment defines the associated *ident-text* (for example, by using the command line option `–define`), the token stream includes the tokens between the *if-directive* and the corresponding *else-directive* or *endif-directive*. Otherwise, the tokens are discarded. The converse applies to the text between any corresponding *else-directive* and the *endif-directive*.

- In skipped text, `#if ident/#else/#endif` sections can be nested.
- Strings and comments are not treated as special

3.4 Identifiers and Keywords

Identifiers follow the specification in this section.

```
regexp digit-char = [0-9]
regexp letter-char = '\Lu' | '\Ll' | '\Lt' | '\Lm' | '\Lo' | '\Nl'
regexp connecting-char = '\Pc'
regexp combining-char = '\Mn' | '\Mc'
regexp formatting-char = '\Cf'

regexp ident-start-char =
  | letter-char
  | _

regexp ident-char =
  | letter-char
  | digit-char
  | connecting-char
  | combining-char
  | formatting-char
  | '
  | _

regexp ident-text = ident-start-char ident-char*
```

```
token ident =
  | ident-text      For example, myName1
  | `` ( [^'`' '\n' '\r' '\t'] | '`' [^ '`' '\n' '\r' '\t'] )+ ``
                               For example, ``value.with odd#name``
```

Any sequence of characters that is enclosed in double-backtick marks (`` ``), excluding newlines, tabs, and double-backtick pairs themselves, is treated as an identifier. Note that when an identifier is used for the name of a types, union type case, module, or namespace, the following characters are not allowed even inside double-backtick marks:

‘.', '+', '$', '&', '[', ']', '/', '\\', '*', '\"', '`'

All input files are currently assumed to be encoded as UTF-8. See the C# specification for a list of the Unicode characters that are accepted for the Unicode character classes \Lu, \Li, \Lt, \Lm, \Lo, \Nl, \Pc, \Mn, \Mc, and \Cf.

The following identifiers are treated as keywords of the F# language:

```
token ident-keyword =
      abstract and as assert base begin class default delegate do
done
      downcast downto elif else end exception extern false finally
for
      fun function global if in inherit inline interface internal
lazy let
      match member module mutable namespace new null of open or
      override private public rec return sig static struct then to
      true try type upcast use val void when while with yield
```

The following identifiers are reserved for future use:

```
token reserved-ident-keyword =
    atomic break checked component const constraint constructor
    continue eager fixed fori functor include
    measure method mixin object parallel params process protected pure
    recursive sealed tailcall trait virtual volatile
```

A future revision of the F# language may promote any of these identifiers to be full keywords.

The following token forms are reserved, except when they are part of a symbolic keyword (§3.6).

```
token reserved-ident-formats =
    | ident-text ( '!' | '#')
```

In the remainder of this specification, we refer to the token that is generated for a keyword simply by using the text of the keyword itself.

3.5 Strings and Characters

String literals may be specified for two types:

- Unicode strings, type `string` = `System.String`
- Unsigned byte arrays, type `byte[]` = `bytearray`

Literals may also be specified by using C#-like verbatim forms that interpret `\` as a literal character rather than an escape sequence. In a UTF-8-encoded file, you can directly embed the following in a string in the same way as in C#:

- Unicode characters, such as "`\u0041bc`"
- Identifiers, as described in the previous section, such as "`abc`"
- Trigraph specifications of Unicode characters, such as "\067" which represents "C"

```
regexp escape-char =  '\' ["\'ntbrafv]
regexp non-escape-chars =  '\' [^"\'ntbrafv]
regexp simple-char-char =
      | (any char except '\n' '\t' '\r' '\b' '\a' '\f' '\v' ' \ ")

regexp unicodegraph-short = '\' 'u' hexdigit hexdigit hexdigit
hexdigit
regexp unicodegraph-long =  '\' 'U' hexdigit hexdigit hexdigit
hexdigit
                                    hexdigit hexdigit hexdigit
hexdigit

regexp trigraph = '\' digit-char digit-char digit-char

regexp char-char =
     | simple-char-char
     | escape-char
     | trigraph
     | unicodegraph-short

regexp string-char =
     | simple-string-char
     | escape-char
     | non-escape-chars
     | trigraph
     | unicodegraph-short
     | unicodegraph-long
     | newline

regexp string-elem =
     | string-char
     | '\' newline whitespace* string-elem

token char                =  ' char-char '
token string              =  " string-char* "

regexp verbatim-string-char =
```

```
     | simple-string-char
     | non-escape-chars
     | newline
     | \
     | ""

token verbatim-string     = @" verbatim-string-char* "

token bytechar            =  ' simple-or-escape-char 'B
token bytearray           =  " string-char* "B
token verbatim-bytearray = @" verbatim-string-char* "B
token simple-or-escape-char = escape-char | simple-char
token simple-char = any char except
newline,return,tab,backspace,',\,"


token triple-quoted-string = """ simple-or-escape-char* """
```

To translate a string token to a string value, the F# parser concatenates all the Unicode characters for the *string-char* elements within the string. Strings may include `\n` as a newline character. However, if a line ends with `\`, the newline character and any leading whitespace elements on the subsequent line are ignored. Thus, the following gives `s` the value `"abcdef"`:

```
let s = "abc\
    def"
```

Without the backslash, the resulting string includes the newline and whitespace characters. For example:

```
let s = "abc
    def"
```

In this case, `s` has the value `"abc\010    def"` where `\010` is the embedded control character for `\n`, which has Unicode UTF-16 value 10.

Verbatim strings may be specified by using the `@` symbol preceding the string as in C#. For example, the following assigns the value `"abc\def"` to `s`.

```
let s = @"abc\def"
```

String-like and character-like literals can also be specified for unsigned byte arrays (type `byte[]`). These tokens cannot contain Unicode characters that have surrogate-pair UTF-16 encodings or UTF-16 encodings greater than 127.

A triple-quoted string is specified by using three quotation marks (`"""`) to ensure that a string that includes one or more escaped strings is interpreted verbatim. For example, a triple-quoted string can be used to embed XML blobs:

```
let catalog = """
<?xml version="1.0"?>
<catalog>
   <book id="book">
```

```
        <author>Author</author>
        <title>F#</title>
        <genre>Computer</genre>
        <price>44.95</price>
        <publish_date>2012-10-01</publish_date>
        <description>An in-depth look at creating applications in
F#</description>
   </book>
</catalog>
"""
```

3.6 Symbolic Keywords

The following symbolic or partially symbolic character sequences are treated as keywords:

```
token symbolic-keyword =
    let! use! do! yield! return!
    | -> <- . : ( ) [ ] [< >] [| |] { }
    ' # :?> :? :> .. :: := ;; ; =
    _ ? ?? (*) <@ @> <@@ @@>
```

The following symbols are reserved for future use:

```
token reserved-symbolic-sequence =
    ~ `
```

3.7 Symbolic Operators

User-defined and library-defined symbolic operators are sequences of characters as shown below, except where the sequence of characters is a symbolic keyword (§3.6).

```
regexp first-op-char = !%&*+-./<=>@^|~
regexp op-char       = first-op-char | ?

token quote-op-left  =
    |  <@ <@@

token quote-op-right  =
    |  @> @@>

token symbolic-op  =
    | ?
    | ?<-
    | first-op-char op-char*
    | quote-op-left
    | quote-op-right
```

For example, `&&&` and `|||` are valid symbolic operators. Only the operators `?` and `?<-` may start with `?`.

The *quote-op-left* and *quote-op-right* operators are used in quoted expressions (§6.8).

For details about the associativity and precedence of symbolic operators in expression forms, see §4.4.

3.8 Numeric Literals

The lexical specification of numeric literals is as follows:

```
regexp digit= [0-9]
regexp hexdigit     = digit | [A-F] | [a-f]
regexp octaldigit   = [0-7]
regexp bitdigit     = [0-1]

regexp int =
       | digit+     For example, 34

regexp xint =
       | 0 (x|X) hexdigit+ For example, 0x22
       | 0 (o|O) octaldigit+       For example, 0o42
       | 0 (b|B) bitdigit+ For example, 0b10010

token sbyte       = (int|xint) 'y'  For example, 34y
token byte        = (int|xint) 'uy' For example, 34uy
token int16       = (int|xint) 's'  For example, 34s
token uint16      = (int|xint) 'us' For example, 34us
token int32       = (int|xint) 'l'  For example, 34l
token uint32      = (int|xint) 'ul' For example, 34ul
                  | (int|xint) 'u'  For example, 34u
token nativeint   = (int|xint) 'n'  For example, 34n
token unativeint  = (int|xint) 'un' For example, 34un
token int64       = (int|xint) 'L'  For example, 34L
token uint64      = (int|xint) 'UL' For example, 34UL
                  | (int|xint) 'uL' For example, 34uL

token ieee32      =
       | float [Ff] For example, 3.0F or 3.0f
       | xint 'lf'  For example, 0x00000000lf
token ieee64      =
       | float                For example, 3.0
       | xint 'LF'  For example, 0x0000000000000000LF

token bignum  = int ('Q' | 'R' | 'Z' | 'I' | 'N' | 'G')
                                          For example,
34742626263193832612536171N

token decimal = (float|int) [Mm]

token float =
      digit+ . digit*
      digit+ (. digit* )? (e|E) (+|-)? digit+
```

3.8.1 Post-filtering of Adjacent Prefix Tokens

Negative integers are specified using the `-` token; for example, `-3`. The token steam is post-filtered according to the following rules:

- If the token stream contains the adjacent tokens `- token`:

  If *token* is a constant numeric literal, the pair of tokens is merged. For example, adjacent tokens `-` and `3` becomes the single token "`-3`". Otherwise, the tokens remain separate. However the "`-`" token is marked as an `ADJACENT_PREFIX_OP` token.

  This rule does not apply to the sequence `token1 - token2`, if all three tokens are adjacent and *token1* is a terminating token from expression forms that have lower precedence than the grammar production `expr = MINUS expr`.

  For example, the `-` and `b` tokens in the following sequence are not merged if all three tokens are adjacent:

  ```
  a-b
  ```

- Otherwise, the usual grammar rules apply to the uses of `-` and `+`, with an addition for `ADJACENT_PREFIX_OP`:

  ```
  expr = expr MINUS expr
       | MINUS expr
       | ADJACENT_PREFIX_OP expr
  ```

3.8.2 Post-filtering of Integers Followed by Adjacent ".."

Tokens of the form

```
token intdotdot = int..
```

such as `34..` are post-filtered to two tokens: one *int* and one *symbolic-keyword*, "`..`".

This rule allows "`..`" to immediately follow an integer. This construction is used in expressions of the form `[for x in 1..2 -> x + x ]`. Without this rule, the longest-match rule would consider this sequence to be a floating-point number followed by a "`.`".

3.8.3 Reserved Numeric Literal Forms

The following token forms are reserved for future numeric literal formats:

```
token reserved-literal-formats =
       | (xint | ieee32 | ieee64) ident-char+
```

3.8.4 Shebang

A shebang (#!) directive may exist at the beginning of F# source files. Such a line is treated as a comment. This allows F# scripts to be compatible with the Unix convention whereby a script indicates the interpreter to use by providing the path to that interpreter on the first line, following the #! directive.

```
#!/bin/usr/env fsharpi --exec
```

3.9 Line Directives

Line directives adjust the source code filenames and line numbers that are reported in error messages, recorded in debugging symbols, and propagated to quoted expressions. F# supports the following line directives:

```
token line-directive =
    # int
    # int string
    # int verbatim-string
    #line int
    #line int string
    #line int verbatim-string
```

A line directive applies to the line that immediately follows the directive. If no line directive is present, the first line of a file is numbered 1.

3.10 Hidden Tokens

Some hidden tokens are inserted by lexical filtering (§15) or are used to replace existing tokens. See §15 for a full specification and for the augmented grammar rules that take these into account.

3.11 Identifier Replacements

The following table lists identifiers that are automatically replaced by expressions.

Identifier	Replacement
`__SOURCE_DIRECTORY__`	A literal verbatim string that specifies the name of the directory that contains the current file. For example: `C:\source` The name of the current file is derived from the most recent line directive in the file. If no line directive has appeared, the name is derived from the name that was specificed to the command-line compiler in combination with `System.IO.Path.GetFullPath`. In F# Interactive, the name `stdin` is used. When F# Interactive is used from tools such as Visual Studio, a line directive is implicitly added before the interactive execution of each script fragment.
`__SOURCE_FILE__`	A literal verbatim string that contains the name of the current file. For example: `file.fs`
`__LINE__`	A literal string that specifies the line number in the source file, after taking into account adjustments from line directives.

4. Basic Grammar Elements

This section defines grammar elements that are used repeatedly in later sections.

4.1 Operator Names

Several places in the grammar refer to an *ident-or-op* rather than an *ident*:

```
ident-or-op :=
      | ident
      | ( op-name )
      | (*)

op-name :=
      | symbolic-op
      | range-op-name
      | active-pattern-op-name

range-op-name :=
      | ..
      | .. ..

active-pattern-op-name :=
      | | ident | ... | ident |
      | | ident | ... | ident | _ |
```

In operator definitions, the operator name is placed in parentheses. For example:

```
let (+++) x y = (x, y)
```

This example defines the binary operator `+++`. The text `(+++)` is an *ident-or-op* that acts as an identifier with associated text `+++`. Likewise, for active pattern definitions (§7), the active pattern case names are placed in parentheses, as in the following example:

```
let (|A|B|C|) x = if x < 0 then A elif x = 0 then B else C
```

Because an *ident-or-op* acts as an identifier, such names can be used in expressions. For example:

```
List.map ((+) 1) [ 1; 2; 3 ]
```

The three character token `(*)`defines the `*` operator:

```
let (*) x y = (x + y)
```

To define other operators that begin with `*`, whitespace must follow the opening parenthesis; otherwise `(*` is interpreted as the start of a comment:

```
let ( *+* ) x y = (x + y)
```

Symbolic operators and some symbolic keywords have a compiled name that is visible in the compiled form of F# programs. The compiled names are shown below.

```
[]   op_Nil
::   op_ColonColon
+    op_Addition
-    op_Subtraction
*    op_Multiply
/    op_Division
**   op_Exponentiation
@    op_Append
^    op_Concatenate
%    op_Modulus
&&&  op_BitwiseAnd
|||  op_BitwiseOr
^^^  op_ExclusiveOr
<<<  op_LeftShift
~~~  op_LogicalNot
>>>  op_RightShift
~+   op_UnaryPlus
~-   op_UnaryNegation
=    op_Equality
<>   op_Inequality
<=   op_LessThanOrEqual
>=   op_GreaterThanOrEqual
<    op_LessThan
>    op_GreaterThan
?    op_Dynamic
?<-  op_DynamicAssignment
|>   op_PipeRight
||>  op_PipeRight2
|||> op_PipeRight3
<|   op_PipeLeft
<||  op_PipeLeft2
<||| op_PipeLeft3
!    op_Dereference
>>   op_ComposeRight
<<   op_ComposeLeft
<@ @>        op_Quotation
<@@ @@> op_QuotationUntyped
~%   op_Splice
~%%  op_SpliceUntyped
~&   op_AddressOf
~&&  op_IntegerAddressOf
||   op_BooleanOr
&&   op_BooleanAnd
+=   op_AdditionAssignment
-=   op_SubtractionAssignment
*=   op_MultiplyAssignment
/=   op_DivisionAssignment
..   op_Range
.. ..        op_RangeStep
```

Compiled names for other symbolic operators are `op_N1...Nn` where N_1 to N_n are the names for the characters as shown in the table below. For example, the symbolic identifier `<*` has the compiled name `op_LessMultiply`:

```
>    Greater
<    Less
+    Plus
-    Minus
*    Multiply
=    Equals
~    Twiddle
%    Percent
.    Dot
&    Amp
|    Bar
@    At
#    Hash
^    Hat
!    Bang
?    Qmark
/    Divide
.    Dot
:    Colon
(    LParen
,    Comma
)    RParen
[    LBrack
]    RBrack
```

4.2 Long Identifiers

Long identifiers *long-ident* are sequences of identifiers that are separated by '.' and optional whitespace. Long identifiers *long-ident-or-op* are long identifiers that may terminate with an operator name.

```
long-ident :=  ident '.' ... '.' ident
long-ident-or-op :=
   | long-ident '.' ident-or-op
   | ident-or-op
```

4.3 Constants

The constants in the following table may be used in patterns and expressions. The individual lexical formats for the different constants are defined in §3.

```
const :=
       | sbyte
       | int16
       | int32
```

```
        | int64             -- 8, 16, 32 and 64-bit signed integers
        | byte
        | uint16
        | uint32
        | int          -- 32-bit signed integer
        | uint64            -- 8, 16, 32 and 64-bit unsigned integers
        | ieee32            -- 32-bit number of type "float32"
        | ieee64            -- 64-bit number of type "float"
        | bignum            -- User or library-defined integral
literal type
        | char         -- Unicode character of type "char"
        | string            -- String of type "string"
(System.String)
        | verbatim-string   -- String of type "string"
(System.String)
        | triple-quoted-string      -- String of type "string"
(System.String)
        | bytestring   -- String of type "byte[]"
        | verbatim-bytearray -- String of type "byte[]"
        | bytechar     -- Char of type "byte"
        | false | true      -- Boolean constant of type "bool"
        | ()           -- unit constant of type "unit"
```

4.4 Operators and Precedence

4.4.1 Categorization of Symbolic Operators

The following *symbolic-op* tokens can be used to form prefix and infix expressions. The marker OP represents all *symbolic-op* tokens that begin with the indicated prefix, except for tokens that appear elsewhere in the table.

```
infix-or-prefix-op :=
    +,  -, +., -., %, &, &&

prefix-op :=
    infix-or-prefix-op
    ~ ~~ ~~~                (and any repetitions of ~)
    !OP                   (except !=)

infix-op :=
    infix-or-prefix-op
    -OP +OP || <OP >OP = |OP &OP ^OP *OP /OP %OP !=
                          (or any of these preceded by one or more
'.')
    :=
    ::
    $
    or
    ?
```

The operators `+`, `-`, `+.`, `-.`, `%`, `%%`, `&`, `&&` can be used as both prefix and infix operators. When these operators are used as prefix operators, the tilde character is prepended internally to generate the operator name so that the parser can distinguish such usage from an infix use of the operator. For example, `-x` is parsed as an application of the operator `~-` to the identifier `x`. This generated name is also used in definitions for these prefix operators. Consequently, the definitions of the following prefix operators include the `~` character:

```
// To completely redefine the prefix + operator:
let (~+) x = x

// To completely redefine the infix + operator to be addition
modulo-7
let (+) a b = (a + b) % 7

// To define the operator on a type:
type C(n:int) =
    let n = n % 7
    member x.N = n
    static member (~+) (x:C) = x
    static member (~-) (x:C) = C(-n)
    static member (+) (x1:C,x2:C) = C(x1.N+x2.N)
    static member (-) (x1:C,x2:C) = C(x1.N-x2.N)
```

The `::` operator is special. It represents the union case for the addition of an element to the head of an immutable linked list, and cannot be redefined, although it may be used to form infix expressions. It always accepts arguments in tupled form—as do all union cases—rather than in curried form.

4.4.2 Precedence of Symbolic Operators and Pattern/Expression Constructs

Rules of precedence control the order of evaluation for ambiguous expression and pattern constructs. Higher precedence items are evaluated before lower precedence items.

The following table shows the order of precedence, from highest to lowest, and indicates whether the operator or expression is associated with the token to its left or right. The `OP` marker represents the *symbolic-op* tokens that begin with the specified prefix, except those listed elsewhere in the table. For example, `+OP` represents any token that begins with a plus sign, unless the token appears elsewhere in the table.

Operator or expression	Associativity	Comments
`f<types>`	Left	High-precedence type application; see §15.3
`f(x)`	Left	High-precedence application; see §15.2
`.`	Left	
prefix-op	Left	Applies to prefix uses of these symbols
`"| rule"`	Right	Pattern matching rules

`"f x"`
`"lazy x"`
`"assert x"`
	Left	

`**OP`	Right	
`*OP /OP %OP`	Left	
`-OP +OP`	Left	Applies to infix uses of these symbols

Operator or expression	Associativity	Comments
`:?`	Not associative	
`::`	Right	
`^OP`	Right	
`!=OP <OP >OP = |OP &OP $`	Left	
`:> :?>`	Right	
`& &&`	Left	
`or ||`	Left	
`,`	Not associative	
`:=`	Right	
`->`	Right	
`if`	Not associative	
`function, fun, match, try`	Not associative	
`let`	Not associative	
`;`	Right	
`|`	Left	
`when`	Right	
`as`	Right	

If ambiguous grammar rules (such as the rules from §6) involve tokens in the table, a construct that appears earlier in the table has higher precedence than a construct that appears later in the table. The associativity indicates whether the operator or construct applies to the item to the left or the right of the operator.

For example, consider the following token stream:

```
a + b * c
```

In this expression, the *`expr infix-op expr`* rule for `b * c` takes precedence over the *`expr infix-op expr`* rule for `a + b`, because the * operator has higher precedence than the + operator. Thus, this expression can be pictured as follows:

```
a + b * c
```

rather than

```
a + b * c
```

Likewise, given the tokens

```
a * b * c
```

the left associativity of `*` means we can picture the resolution of the ambiguity as:

```
a * b * c
```

In the preceding table, leading `.` characters are ignored when determining precedence for infix operators. For example, `.*` has the same precedence as `*`. This rule ensures that operators such as `.*`, which is frequently used for pointwise-operation on matrices, have the expected precedence.

The table entries marked as "High-precedence application" and "High-precedence type application" are the result of the augmentation of the lexical token stream, as described in §15.2 and §15.3.

5. Types and Type Constraints

The notion of *type* is central to both the static checking of F# programs and to dynamic type tests and reflection at runtime. The word is used with four distinct but related meanings:

- **Type definitions,** such as the actual CLI or F# definitions of `System.String` or `FSharp.Collections.Map<_,_>`.
- **Syntactic types,** such as the text `option<_>` that might occur in a program text. Syntactic types are converted to static types during the process of type checking and inference.
- **Static types**, which result from type checking and inference, either by the translation of syntactic types that appear in the source text, or by the application of constraints that are related to particular language constructs. For example, `option<int>` is the fully processed static type that is inferred for an expression `Some(1+1)`. Static types may contain *type variables* as described later in this section.
- **Runtime types**, which are objects of type `System.Type` and represent some or all of the information that type definitions and static types convey at runtime. The `obj.GetType()` method, which is available on all F# values, provides access to the runtime type of an object. An object's runtime type is related to the static type of the identifiers and expressions that correspond to the object. Runtime types may be tested by built-in language operators such as `:?` and `:?>`, the expression form `downcast expr`, and pattern matching type tests. Runtime types of objects do not contain type variables. Runtime types that `System.Reflection` reports may contain type variables that are represented by `System.Type` values.

The following describes the syntactic forms of types as they appear in programs:

```
type :=
    ( type )
    type -> type     -- function type
    type * ... * type       -- tuple type
    typar            -- variable type
    long-ident       -- named type, such as int
    long-ident<type-args>   -- named type, such as list<int>
    long-ident< >    -- named type, such as IEnumerable< >
    type long-ident -- named type, such as int list
    type[ , ... , ] -- array type
    type typar-defns        -- type with constraints
    typar :> type    -- variable type with subtype constraint
    #type            -- anonymous type with subtype constraint

type-args :=  type-arg, ..., type-arg

type-arg :=
    type             -- type argument
    measure          -- unit of measure argument
    static-parameter        -- static parameter

atomic-type :=
```

```
     type :  one of
           #type typar ( type ) long-ident long-ident<type-args>

typar :=
    _                         -- anonymous variable type
    'ident          -- type variable
    ^ident          -- static head-type type variable

constraint :=
    typar :> type   -- coercion constraint
    typar : null    -- nullness constraint
    static-typars : (member-sig )  -- member "trait" constraint
    typar : (new : unit -> 'T)     -- CLI default constructor
constraint
    typar : struct  -- CLI non-Nullable struct
    typar : not struct      -- CLI reference type
    typar : enum<type>      -- enum decomposition constraint
    typar : unmanaged       -- unmanaged constraint
    typar : delegate<type, type>   -- delegate decomposition
constraint
    typar : equality
    typar : comparison


typar-defn := attributes_opt typar

typar-defns  := < typar-defn, ..., typar-defn typar-constraints_opt >

typar-constraints := when constraint and ... and constraint

static-typars :=
    ^ident
    (^ident or ... or ^ident)

member-sig := <see Section 10>
```

In a type instantiation, the type name and the opening angle bracket must be syntactically adjacent with no intervening whitespace, as determined by lexical filtering (§15). Specifically:

```
array<int>
```

and not

```
array <  int   >
```

5.1 Checking Syntactic Types

Syntactic types are checked and converted to *static types* as they are encountered. Static types are a specification device used to describe

- The process of type checking and inference.

- The connection between syntactic types and the execution of F# programs.

Every expression in an F# program is given a unique inferred static type, possibly involving one or more explicit or implicit generic parameters.

For the remainder of this specification we use the same syntax to represent syntactic types and static types. For example `int32 * int32` is used to represent the syntactic type that appears in source code and the static type that is used during checking and type inference.

The conversion from syntactic types to static types happens in the context of a *name resolution environment* (§14.1), a *floating type variable environment*, which is a mapping from names to type variables, and a *type inference environment* (§14.5).

The phrase "fresh type" means a static type that is formed from a *fresh type inference variable*. Type inference variables are either solved or generalized by *type inference* (§14.5). During conversion and throughout the checking of types, expressions, declarations, and entire files, a set of *current inference constraints* is maintained. That is, each static type is processed under input constraints *X*, and results in output constraints *X'*. Type inference variables and constraints are progressively *simplified* and *eliminated* based on these equations through *constraint solving* (§14.5).

5.1.1 Named Types

Named types have several forms, as listed in the following table.

Form	Description
long-ident<ty_1,…,ty_n>	Named type with one or more suffixed type arguments.
long-ident	Named type with no type arguments
type long-ident	Named type with one type argument; processed the same as *long-ident*<*type*>
ty_1 `->` ty_2	A function type, where: ▪ `ty1` is the domain of the function values associated with the type ▪ `ty2` is the range. In compiled code it is represented by the named type `FSharp.Core.FastFunc`<ty_1,ty_2>.

Named types are converted to static types as follows:

- *Name Resolution for Types* (§14.1) resolves *long-ident* to a type definition with formal generic parameters <$typar_1$,…, $typar_n$> and formal constraints *C*. The number of type arguments *n* is used during the name resolution process to distinguish between similarly named types that take different numbers of type arguments.
- Fresh type inference variables <ty'_1,…,ty'_n> are generated for each formal type parameter. The formal constraints *C* are added to the current inference constraints for the new type inference variables; and constraints $ty_i = ty'_i$ are added to the current inference constraints.

5.1.2 Variable Types

A type of the form *'ident* is a *variable type*. For example, the following are all variable types:

```
'a
```

```
'T
'Key
```

During checking, *Name Resolution* (§14.1) is applied to the identifier.

- If name resolution succeeds, the result is a variable type that refers to an existing declared type parameter.
- If name resolution fails, the current *floating type variable environment* is consulted, although only in the context of a syntactic type that is embedded in an expression or pattern. If the type variable name is assigned a type in that environment, F# uses that mapping. Otherwise, a fresh type inference variable is created (see §14.5) and added to both the type inference environment and the floating type variable environment.

A type of the form `_` is an *anonymous variable type*. A fresh type inference variable is created and added to the type inference environment (see §14.5) for such a type.

A type of the form `^ident` is a *statically resolved type variable*. A fresh type inference variable is created and added to the type inference environment (see §14.5). This type variable is tagged with an attribute that indicates that it can be generalized only at `inline` definitions (see §14.6.7). The same restriction on generalization applies to any type variables that are contained in any type that is equated with the `^ident` type in a type inference equation.

> Note: this specification generally uses uppercase identifiers such as `'T` or `'Key` for user-declared generic type parameters, and uses lowercase identifiers such as `'a` or `'b` for compiler-inferred generic parameters.

5.1.3 Tuple Types

A *tuple type* has the following form:

ty_1 * ... * ty_n

The elaborated form of a tuple type is shorthand for a use of the family of F# library types `System.Tuple<_,...,_>`. See §6.3.2 for the details of this encoding.

When considered as static types, tuple types are distinct from their encoded form. However, the encoded form of tuple types is visible in the F# type system through runtime types. For example, `typeof<int * int>` is equivalent to `typeof<System.Tuple<int,int>>`.

5.1.4 Array Types

Array types have the following forms:

```
ty[]
ty[ , ... , ]
```

A type of the form `ty[]` is a *single-dimensional array type*, and a type of the form `ty[ , ... , ]` is a *multidimensional array type*. For example, `int[,,]` is an array of integers of rank 3.

Except where specified otherwise in this document, these array types are treated as named types, as if they are an instantiation of a fictitious type definition `System.Array`$_n$`<`*ty*`>` where *n* corresponds to the rank of the array type.

> Note: The type `int[][,]` in F# is the same as the type `int[,][]` in C# although the dimensions are swapped. This ensures consistency with other postfix type names in F# such as `int list list`.
>
> F# supports multidimensional array types only up to rank 4.

5.1.5 Constrained Types

A *type with constraints* has the following form:

```
type when constraints
```

During checking, *type* is first checked and converted to a static type, then *constraints* are checked and added to the current inference constraints. The various forms of constraints are described in§5.2.

A type of the form `typar :> type` is a *type variable with a subtype constraint* and is equivalent to `typar when typar :> type`.

A type of the form `#type` is an *anonymous type with a subtype constraint* and is equivalent to `'a when 'a :> type`, where `'a` is a fresh type inference variable.

5.2 Type Constraints

A *type constraint* limits the types that can be used to create an instance of a type parameter or type variable. F# supports the following type constraints:

- Subtype constraints
- Nullness constraints
- Member constraints
- Default constructor constraints
- Value type constraints
- Reference type constraints
- Enumeration constraints
- Delegate constraints
- Unmanaged constraints
- Equality and comparison constraints

5.2.1 Subtype Constraints

An *explicit subtype constraint* has the following form:

```
typar :> type
```

During checking, *typar* is first checked as a variable type, *type* is checked as a type, and the constraint is added to the current inference constraints. Subtype constraints affect type coercion as specified in §5.4.7.

Note that subtype constraints also result implicitly from:

- Expressions of the form `expr :> type`.
- Patterns of the form `pattern :> type`.
- The use of generic values, types, and members with constraints.
- The implicit use of subsumption when using values and members (§14.4.3).

A type variable cannot be constrained by two distinct instantiations of the same named type. If two such constraints arise during constraint solving, the type instantiations are constrained to be equal. For example, during type inference, if a type variable is constrained by both `IA<int>` and `IA<string>`, an error occurs when the type instantiations are constrained to be equal. This limitation is specifically necessary to simplify type inference, reduce the size of types shown to users, and help ensure the reporting of useful error messages.

5.2.2 Nullness Constraints

An *explicit nullness constraint* has the following form:

```
typar: null
```

During checking, *typar* is checked as a variable type and the constraint is added to the current inference constraints. The conditions that govern when a type satisfies a nullness constraint are specified in §5.4.8.

In addition:

- The *typar* must be a statically resolved type variable of the form *^ident*. This limitation ensures that the constraint is resolved at compile time, and means that generic code may not use this constraint unless that code is marked `inline` (§14.6.7).

> Note: Nullness constraints are primarily for use during type checking and are used relatively rarely in F# code.
>
> Nullness constraints also arise from expressions of the form `null`.

5.2.3 Member Constraints

An *explicit member constraint* has the following form:

```
(typar or ... or typar) : (member-sig)
```

For example, the F# library defines the `+` operator with the following signature:

```
val inline (+) : ^a -> ^b -> ^c
      when (^a or ^b) : (static member (+) : ^a * ^b -> ^c)
```

This definition indicates that each use of the `+` operator results in a constraint on the types that correspond to parameters `^a`, `^b`, and `^c`. If these are named types, then either the named type for `^a` or the named type for `^b` must support a static member called `+` that has the given signature.

In addition:

- Each *`typar`* must be a statically resolved type variable (§5.1.2) in the form *`^ident`*. This ensures that the constraint is resolved at compile time against a corresponding named type. It also means that generic code cannot use this constraint unless that code is marked `inline` (§14.6.7).
- The *`member-sig`* cannot be generic; that is, it cannot include explicit type parameter definitions.
- The conditions that govern when a type satisfies a member constraint are specified in §14.5.4 .

> Note: Member constraints are primarily used to define overloaded functions in the F# library and are used relatively rarely in F# code.
>
> Uses of overloaded operators do not result in generalized code unless definitions are marked as `inline`. For example, the function
>
> ```
> let f x = x + x
> ```
>
> results in a function `f` that can be used only to add one type of value, such as `int` or `float`. The exact type is determined by later constraints.

A type variable may not be involved in the support set of more than one member constraint that has the same name, staticness, argument arity, and support set (§14.5.4). If it is, the argument and return types in the two member constraints are themselves constrained to be equal. This limitation is specifically necessary to simplify type inference, reduce the size of types shown to users, and ensure the reporting of useful error messages.

5.2.4 Default Constructor Constraints

An *explicit default constructor constraint* has the following form:

```
typar : (new : unit -> 'T)
```

During constraint solving (§14.5), the constraint *`type`* `: (new : unit -> 'T)` is met if *`type`* has a parameterless object constructor.

> **Note:** This constraint form exists primarily to provide the full set of constraints that CLI implementations allow. It is rarely used in F# programming.

5.2.5 Value Type Constraints

An *explicit value type constraint* has the following form:

```
typar : struct
```

During constraint solving (§14.5), the constraint *`type`* `: struct` is met if *`type`* is a value type other than the CLI type `System.Nullable<_>`.

> **Note:** This constraint form exists primarily to provide the full set of constraints that CLI implementations allow. It is rarely used in F# programming.
>
> The restriction on `System.Nullable` is inherited from C# and other CLI languages, which give this type a special syntactic status. In F#, the type `option<_>` is similar to some uses of `System.Nullable<_>`. For various technical reasons the two types cannot be equated, notably because types such as `System.Nullable<System.Nullable<_>>` and `System.Nullable<string>` are not valid CLI types.

5.2.6 Reference Type Constraints

An *explicit reference type constraint* has the following form:

```
typar : not struct
```

During constraint solving (§14.5), the constraint `type : not struct` is met if `type` is a reference type.

> **Note:** This constraint form exists primarily to provide the full set of constraints that CLI implementations allow. It is rarely used in F# programming.

5.2.7 Enumeration Constraints

An *explicit enumeration constraint* has the following form:

```
typar : enum<underlying-type>
```

During constraint solving (§14.5), the constraint `type : enum<underlying-type>` is met if `type` is a CLI or F# enumeration type that has constant literal values of type `underlying-type`.

> Note: This constraint form exists primarily to allow the definition of library functions such as `enum`. It is rarely used directly in F# programming.
>
> The `enum` constraint does not imply anything about subtypes. For example, an `enum` constraint does not imply that the type is a subtype of `System.Enum`.

5.2.8 Delegate Constraints

An *explicit delegate constraint* has the following form:

```
typar : delegate<tupled-arg-type, return-type>
```

During constraint solving (§14.5), the constraint `type : delegate<tupled-arg-type, return-types>` is met if `type` is a delegate type `D` with declaration `type D = delegate of object * arg1 * ... * argN` and `tupled-arg-type = arg1 * ... * argN`. That is, the delegate must match the CLI design pattern where the `sender` object is the first argument to the event.

> **Note**: This constraint form exists primarily to allow the definition of certain F# library functions that are related to event programming. It is rarely used directly in F# programming.

> The `delegate` constraint does not imply anything about subtypes. In particular, a 'delegate' constraint does not imply that the type is a subtype of `System.Delegate`.
>
> The `delegate` constraint applies only to delegate types that follow the usual form for CLI event handlers, where the first argument is a "sender" object. The reason is that the purpose of the constraint is to simplify the presentation of CLI event handlers to the F# programmer.

5.2.9 Unmanaged Constraints

An *unmanaged constraint* has the following form:

```
typar : unmanaged
```

During constraint solving (§14.5), the constraint *`type`* `: unmanaged` is met if *`type`* is unmanaged as specified below:

- Types `sbyte`, `byte`, `char`, `nativeint`, `unativeint`, `float32`, `float`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `decimal` are unmanaged.
- Type `nativeptr<`*`type`*`>` is unmanaged.
- A non-generic struct type whose fields are all unmanaged types is unmanaged.

5.2.10 Equality and Comparison Constraints

Equality constraints and *comparison constraints* have the following forms, respectively:

```
typar : equality
typar : comparison
```

During constraint solving (§14.5), the constraint *`type`* `: equality` is met if both of the following conditions are true:

- The type is a named type, and the type definition does not have, and is not inferred to have, the `NoEquality` attribute.
- The type has *equality dependencies* ty_1, ..., ty_n, each of which satisfies ty_i `: equality`.

The constraint *`type`* `: comparison` is a *comparison constraint*. Such a constraint is met if all the following conditions hold:

- If the type is a named type, then the type definition does not have, and is not inferred to have, the `NoComparison` attribute, and the type definition implements `System.IComparable` or is an array type or is `System.IntPtr` or is `System.UIntPtr`.
- If the type has *comparison dependencies* ty_1, ..., ty_n, then each of these must satisfy ty_i `: comparison`

An equality constraint is a relatively weak constraint, because with two exceptions, all CLI types satisfy this constraint. The exceptions are F# types that are annotated with the `NoEquality` attribute and structural types that are inferred to have the `NoEquality` attribute. The reason is that in other CLI languages, such as C#, it possible to use reference equality on all reference types.

A comparison constraint is a stronger constraint, because it usually implies that a type must implement `System.IComparable`.

5.3 Type Parameter Definitions

Type parameter definitions can occur in the following locations:

- Value definitions in modules
- Member definitions
- Type definitions
- Corresponding specifications in signatures

For example, the following defines the type parameter 'T in a function definition:

```
let id<'T> (x:'T) = x
```

Likewise, in a type definition:

```
type Funcs<'T1,'T2> =
    { Forward: 'T1 -> 'T2;
      Backward : 'T2 -> 'T2 }
```

Likewise, in a signature file:

```
val id<'T> : 'T -> 'T
```

Explicit type parameter definitions can include *explicit constraint declarations*. For example:

```
let dispose2<'T when 'T :> System.IDisposable> (x: 'T, y: 'T) =
    x.Dispose()
    y.Dispose()
```

The constraint in this example requires that `'T` be a type that supports the `IDisposable` interface.

However, in most circumstances, declarations that imply subtype constraints on arguments can be written more concisely:

```
let throw (x: Exception) = raise x
```

Multiple explicit constraint declarations use `and`:

```
let multipleConstraints<'T when 'T :> System.IDisposable and
                                'T :> System.IComparable > (x: 'T, y:
'T) =
    if x.CompareTo(y) < 0 then x.Dispose() else y.Dispose()
```

Explicit type parameter definitions can declare custom attributes on type parameter definitions (§13.1).

5.4 Logical Properties of Types

During type checking and elaboration, syntactic types and constraints are processed into a reduced form composed of:

- Named types `op<types>`, where each `op` consists of a specific type definition, an operator to form function types, an operator to form array types of a specific rank, or an operator to form specific *n*-tuple types.
- Type variables `'ident`.

5.4.1 Characteristics of Type Definitions

Type definitions include CLI type definitions such as `System.String` and types that are defined in F# code (§8). The following terms are used to describe type definitions:

- Type definitions may be *generic*, with one or more type parameters; for example, `System.Collections.Generic.Dictionary<'Key,'Value>`.
- The generic parameters of type definitions may have associated *formal type constraints*.
- Type definitions may have *custom attributes* (§13.1), some of which are relevant to checking and inference.
- Type definitions may be *type abbreviations* (§8.3). These are eliminated for the purposes of checking and inference (see §5.4.2).
- Type definitions have a *kind* which is one of the following:
    - *Class*
    - *Interface*
    - *Delegate*
    - *Struct*
    - *Record*
    - *Union*
    - *Enum*
    - *Measure*
    - *Abstract*

  The kind is determined at the point of declaration by Type Kind Inference (§8.2) if it is not specified explicitly as part of the type definition. The *kind* of a type refers to the kind of its outermost named type definition, after expanding abbreviations. For example, a type is a *class* type if it is a named type `C<types>` where `C` is of kind *class*. Thus, `System.Collections.Generic.List<int>` is a class type.

- Type definitions may be *sealed.* Record, union, function, tuple, struct, delegate, enum, and array types are all sealed, as are class types that are marked with the `SealedAttribute` attribute.
- Type definitions may have zero or one *base type declarations*. Each base type declaration represents an additional type that is supported by any values that are formed using the type definition. Furthermore, some aspects of the base type are used to form the implementation of the type definition.
- Type definitions may have one or more *interface declarations*. These represent additional encapsulated types that are supported by values that are formed using the type.

Class, interface, delegate, function, tuple, record, and union types are all *reference* type definitions. A type is a reference type if its outermost named type definition is a reference type, after expanding type definitions.

Struct types are *value types*.

5.4.2 Expanding Abbreviations and Inference Equations

Two static types are considered equivalent and indistinguishable if they are equivalent after taking into account both of the following:

- The inference equations that are inferred from the current inference constraints (§14.5).
- The expansion of type abbreviations (§8.3).

For example, static types may refer to type abbreviations such as `int`, which is an abbreviation for `System.Int32`and is declared by the F# library:

```
type int = System.Int32
```

This means that the types `int32` and `System.Int32` are considered equivalent, as are `System.Int32 -> int` and `int -> System.Int32`.

Likewise, consider the process of checking this function:

```
let checkString (x:string) y =
    (x = y), y.Contains("Hello")
```

During checking, fresh type inference variables are created for values `x` and `y`; let's call them ty_1 and ty_2. Checking imposes the constraints ty_1 `= string` and $ty_1 = ty_2$. The second constraint results from the use of the generic `=` operator. As a result of constraint solving, ty_2 `= string` is inferred, and thus the type of `y` is `string`.

All relations on static types are considered after the elimination of all equational inference constraints and type abbreviations. For example, we say `int` is a struct type because `System.Int32` is a struct type.

> Note: Implementations of F# should attempt to preserve type abbreviations when reporting types and errors to users. This typically means that type abbreviations should be preserved in the logical structure of types throughout the checking process.

5.4.3 Type Variables and Definition Sites

Static types may be type variables. During type inference, static types may be *partial*, in that they contain type inference variables that have not been solved or generalized. Type variables may also refer to explicit type parameter definitions, in which case the type variable is said to be *rigid* and have a *definition site*.

For example, in the following, the definition site of the type parameter `'T` is the type definition of `C`:

```
type C<'T> = 'T * 'T
```

Type variables that do not have a binding site are *inference variables*. If an expression is composed of multiple sub-expressions, the resulting constraint set is normally the union of the constraints that result from checking all the sub-expressions. However, for some constructs (notably function, value and member definitions), the checking process applies *generalization* (§14.6.7). Consequently, some intermediate inference variables and constraints are factored out of the intermediate constraint sets and new implicit definition site(s) are assigned for these variables.

For example, given the following declaration, the type inference variable that is associated with the value `x` is generalized and has an implicit definition site at the definition of function `id`:

```
let id x = x
```

Occasionally in this specification we use a more fully annotated representation of inferred and generalized type information. For example:

$$\texttt{let id}_{<'a>}\ \texttt{x}_{'a} = \texttt{x}_{'a}$$

Here, `'a` represents a generic type parameter that is inferred by applying type inference and generalization to the original source code (§14.6.7), and the annotation represents the definition site of the type variable.

5.4.4 Base Type of a Type

The *base type* for the static types is shown in the table. These types are defined in the CLI specifications and corresponding implementation documentation.

Static Type	Base Type
Abstract types	`System.Object`
All array types	`System.Array`
Class types	The declared base type of the type definition if the type has one; otherwise, `System.Object`. For generic types `C<type-inst>`, substitute the formal generic parameters of `C` for *`type-inst`*.
Delegate types	`System.MulticastDelegate`
Enum types	`System.Enum`
Exception types	`System.Exception`
Interface types	`System.Object`
Record types	`System.Object`
Struct types	`System.ValueType`
Union types	`System.Object`
Variable types	`System.Object`

5.4.5 Interfaces Types of a Type

The *interface types* of a named type `C<`*`type-inst`*`>` are defined by the transitive closure of the interface declarations of `C` and the interface types of the base type of `C`, where formal generic parameters are substituted for the actual type instantiation *`type-inst`*.

The interface types for single dimensional array types *`ty`*`[]` include the transitive closure that starts from the interface `System.Collections.Generic.IList<`*`ty`*`>`, which includes `System.Collections.Generic.ICollection<`*`ty`*`>` and `System.Collections.Generic.IEnumerable<`*`ty`*`>`.

5.4.6 Type Equivalence

Two static types ty_1 and ty_2 are *definitely equivalent* (with respect to a set of current inference constraints) if either of the following is true:

- ty_1 has form $op\langle ty_{11}, \ldots, ty_{1n}\rangle$, ty_2 has form $op\langle ty_{21}, \ldots, ty_{2n}\rangle$ and each ty_{1i} is definitely equivalent to ty_{2i} for all $1 \le i \le n$.

—OR—

- ty_1 and ty_2 are both variable types, and they both refer to the same definition site or are the same type inference variable.

This means that the addition of new constraints may make types definitely equivalent where previously they were not. For example, given X = { `'a = int` }, we have `list<int>` = `list<'a>`.

Two static types ty_1 and ty_2 are *feasibly equivalent* if ty_1 and ty_2 may become definitely equivalent if further constraints are added to the current inference constraints. Thus `list<int>` and `list<'a>` are feasibly equivalent for the empty constraint set.

5.4.7 Subtyping and Coercion

A static type ty_2 *coerces to* static type ty_1 (with respect to a set of current inference constraints X), if ty_1 is in the transitive closure of the base types and interface types of ty_2. Static coercion is written with the `:>` symbol:

$$ty_2 :> ty_1,$$

Variable types `'T` coerce to all types *`ty`* if the current inference constraints include a constraint of the form `'T :>` ty_2, and *`ty`* is in the inclusive transitive closure of the base and interface types of ty_2.

A static type ty_2 *feasibly coerces to* static type ty_1 if ty_2 *coerces to* ty_1 may hold through the addition of further constraints to the current inference constraints. The result of adding constraints is defined in *Constraint Solving* (§14.5).

5.4.8 Nullness

The design of F# aims to greatly reduce the use of `null` literals in common programming tasks, because they generally result in error-prone code. However:

- The use of some `null` literals is required for interoperation with CLI libraries.
- The appearance of `null` values during execution cannot be completely precluded for technical reasons related to the CLI and CLI libraries.

As a result, F# types differ in their treatment of the `null` literal and `null` values. All named types and type definitions fall into one of the following categories:

- **Types with the `null` literal.** These types have `null` as an "extra" value. The following types are in this category:
  - All CLI reference types that are defined in other CLI languages.
  - All types that are defined in F# and annotated with the `AllowNullLiteral` attribute.

  For example, `System.String` and other CLI reference types satisfy this constraint, and these types permit the direct use of the `null` literal.

- **Types with `null` as an abnormal value.** These types do not permit the `null` literal, but do have `null` as an abnormal value. The following types are in this category:
  - All F# list, record, tuple, function, class, and interface types.
  - All F# union types except those that have `null` as a normal value, as discussed in the next bullet point.

  For types in this category, the use of the `null` literal is not directly allowed. However, strictly speaking, it is possible to generate a `null` value for these types by using certain functions such as `Unchecked.defaultof<`*`type`*`>`. For these types, `null` is considered an abnormal value. Operations differ in their use and treatment of `null` values; for details about evaluation of expressions that might include `null` values, see §6.9.

- **Types with `null` as a representation value.** These types do not permit the `null` literal but use the `null` value as a representation.

  For these types, the use of the `null` literal is not directly permitted. However, one or all of the "normal" values of the type is represented by the `null` value. The following types are in this category:

  - The unit type. The `null` value is used to represent all values of this type.
  - Any union type that has the `FSharp.Core.CompilationRepresentation(CompilationRepresentationFlags.UseNullAsTrueValue)` attribute flag and a single null union case. The `null` value represents this case. In particular, `null` represents `None` in the F# `option<_>` type.

- **Types without `null`.** These types do not permit the `null` literal and do not have the `null` value. All value types are in this category, including primitive integers, floating-point numbers, and any value of a CLI or F# `struct` type.

A static type *ty* *satisfies a nullness constraint* *ty* `: null` if it:

- Has an outermost named type that has the `null` literal.
- Is a variable type with a *typar* `: null` constraint.

5.4.9 Default Initialization

Related to nullness is the *default initialization* of values of some types to *zero values*. This technique is common in some programming languages, but the design of F# deliberately de-emphasizes it. However, default initialization is allowed in some circumstances:

- Checked default initialization may be used when a type is known to have a valid and "safe" default zero value. For example, the types of fields that are labeled with `DefaultValue(true)` are checked to ensure that they allow default initialization.
- CLI libraries sometimes perform unchecked default initialization, as do the F# library primitives `Unchecked.defaultof<_>` and `Array.zeroCreate`.

The following types *permit default initialization*:

- Any type that satisfies the nullness constraint.
- Primitive value types.
- Struct types whose field types all permit default initialization.

5.4.10 Dynamic Conversion Between Types

A runtime type *vty* *dynamically converts to* a static type *ty* if any of the following are true:

- *vty* coerces to *ty*.
- *vty* is `int32[]`and *ty* is `uint32[]`(or conversely). Likewise for `sbyte[]`/`byte[]`, `int16[]`/`uint16[]`, `int64[]`/`uint64[]`, and `nativeint[]`/`unativeint[]`.
- *vty* is *enum*`[]` where *enum* has underlying type *underlying*, and *ty* is *underlying*`[]` (or conversely), or the (un)signed equivalent of *underlying*`[]` by the immediately preceding rule.
- *vty* is $elemty_1$`[]`, *ty* is $elemty_2$`[]`, $elemty_1$ is a reference type, and $elemty_1$ converts to $elemty_2$.
- *ty* is `System.Nullable<`*vty*`>`.

Note that this specification does not define the full algebra of the conversions of runtime types to static types because the information that is available in runtime types is implementation dependent. However, the specification does state the conditions under which objects are guaranteed to have a runtime type that is compatible with a particular static type.

> **Note**: This specification covers the additional rules of CLI dynamic conversions, all of which apply to F# types. For example:

```
let x = box [| System.DayOfWeek.Monday |]
let y = x :? int32[]
printf "%b" y // true
```

In the previous code, the type `System.DayOfWeek.Monday[]` does not statically coerce to `int32[]`, but the expression `x :? int32[]` evaluates to true.

```
let x = box [| 1 |]
let y = x :? uint32 []
printf "%b" y // true
```

In the previous code, the type `int32[]` does not statically coerce to `uint32[]`, but the expression `x :? uint32 []` evaluates to true.

```
let x = box [| "" |]
let y = x :? obj []
printf "%b" y // true
```

In the previous code, the type `string[]` does not statically coerce to `obj[]`, but the expression `x :? obj []`evaluates to true.

```
let x = box 1
let y = x :? System.Nullable<int32>
printf "%b" y // true
```

In the previous code, the type `int32` does not coerce to `System.Nullable<int32>`, but the expression `x :? System.Nullable<int32>` evaluates to true.

6. Expressions

The expression forms and related elements are as follows:

```
expr :=
    const                    -- a constant value
    ( expr )                 -- block expression
    begin expr end           -- block expression
    long-ident-or-op                 -- lookup expression
    expr '.' long-ident-or-op        -- dot lookup expression
    expr expr                        -- application expression
    expr(expr)                       -- high precedence application
    expr<types>              -- type application expression
    expr infix-op expr               -- infix application expression
    prefix-op expr           -- prefix application expression
    expr.[expr]              -- indexed lookup expression
    expr.[slice-ranges]      -- slice expression
    expr <- expr             -- assignment expression
    expr , ... , expr                -- tuple expression
    new type expr            -- simple object expression
    { new base-call object-members interface-impls } -- object
expression
    { field-initializers }           -- record expression
    { expr with field-initializers }         -- record cloning
expression
    [ expr ; ... ; expr ]    -- list expression
    [| expr ; ... ; expr |]-- array expression
    expr { comp-or-range-expr }      -- computation expression
    [ comp-or-range-expr]    -- computed list expression
    [| comp-or-range-expr |]         -- computed array expression
    lazy expr                        -- delayed expression
    null                     -- the "null" value for a reference type
    expr : type              -- type annotation
    expr :> type             -- static upcast coercion
    expr :? type             -- dynamic type test
    expr :?> type            -- dynamic downcast coercion
    upcast expr              -- static upcast expression
    downcast expr            -- dynamic downcast expression
    let function-defn in expr        -- function definition expression
    let value-defn in expr           -- value definition expression
    let rec function-or-value-defns in expr  -- recursive definition
expression
    use ident = expr in expr                   -- deterministic
disposal expression
    fun argument-pats -> expr        -- function expression
    function rules           -- matching function expression
    expr ; expr              -- sequential execution expression
    match expr with rules    -- match expression
    try expr with rules              -- try/with expression
    try expr finally expr    -- try/finally expression
```

```
    if expr then expr elif-branches_opt else-branch_opt -- conditional
expression
    while expr do expr done -- while loop
    for ident = expr to expr do expr done           -- simple for
loop
    for pat in expr-or-range-expr do expr done      -- enumerable for
loop
    assert expr                 -- assert expression
    <@ expr @>                          -- quoted expression
    <@@ expr @@>                -- quoted expression

    %expr                       -- expression splice
    %%expr                      -- weakly typed expression splice

    (static-typars : (member-sig) expr) -- static member invocation
```

Expressions are defined in terms of patterns and other entities that are discussed later in this specification. The following constructs are also used:

```
exprs := expr ',' ... ',' expr

expr-or-range-expr :=
    expr
    range-expr

elif-branches := elif-branch ... elif-branch

elif-branch := elif expr then expr

else-branch := else expr

function-or-value-defn :=
    function-defn
    value-defn

function-defn :=
    inline_opt access_opt ident-or-op typar-defns_opt argument-pats return-
type_opt = expr

value-defn :=
    mutable_opt access_opt pat typar-defns_opt return-type_opt = expr

return-type :=
    : type

function-or-value-defns :=
    function-or-value-defn and ... and function-or-value-defn

argument-pats:= atomic-pat ... atomic-pat

field-initializer :=
    long-ident = expr       -- field initialization

field-initializers := field-initializer ; ... ; field-initializer

object-construction :=
    type expr       -- construction expression
    type     -- interface construction expression

base-call :=
    object-construction            -- anonymous base construction
    object-construction as ident   -- named base construction

interface-impls := interface-impl ... interface-impl

interface-impl :=
    interface type object-members_opt -- interface implementation

object-members := with member-defns end
```

```
member-defns :=  member-defn ... member-defn
```

Computation and range expressions are defined in terms of the following productions:

```
comp-or-range-expr :=
    comp-expr
    short-comp-expr
    range-expr

comp-expr :=
    let! pat = expr in comp-expr   -- binding computation
    let  pat = expr in comp-expr
    do!  expr in comp-expr -- sequential computation
    do   expr in comp-expr
    use! pat = expr in comp-expr   -- auto cleanup computation
    use  pat = expr in comp-expr
    yield! expr             -- yield computation
    yield expr                     -- yield result
    return! expr            -- return computation
    return expr             -- return result
    if expr then comp-expr -- control flow or imperative action
    if expr then expr else comp-expr
    match expr with pat -> comp-expr |     … | pat -> comp-expr
    try comp-expr with pat -> comp-expr |  … | pat -> comp-expr
    try comp-expr finally expr
    while expr do comp-expr done
    for ident = expr to expr do comp-expr done
    for pat in expr-or-range-expr do comp-expr done
    comp-expr ; comp-expr
    expr

short-comp-expr :=
    for pat in expr-or-range-expr -> expr  -- yield result

range-expr :=
    expr .. expr            -- range sequence
    expr .. expr .. expr           -- range sequence with skip

slice-ranges := slice-range , … , slice-range

slice-range :=
    expr                    -- slice of one element of dimension
    expr..                  -- slice from index to end
    ..expr                  -- slice from start to index
    expr..expr                     -- slice from index to index
    '*'                     -- slice from start to end
```

6.1 Some Checking and Inference Terminology

The rules applied to check individual expressions are described in the following subsections. Where necessary, these sections reference specific inference procedures such as *Name Resolution* (§14.1) and *Constraint Solving* (§14.5).

All expressions are assigned a static type through type checking and inference. During type checking, each expression is checked with respect to an *initial type*. The initial type establishes some of the information available to resolve method overloading and other language constructs. We also use the following terminology:

- The phrase "the type ty_1 is asserted to be equal to the type ty_2" or simply "$ty_1 = ty_2$ is asserted" indicates that the constraint "$ty_1 = ty_2$" is added to the current inference constraints.
- The phrase "ty_1 is asserted to be a subtype of ty_2" or simply "ty_1 `:>` ty_2 is asserted" indicates that the constraint ty_1 `:>` ty_2 is added to the current inference constraints.
- The phrase "type *ty* is known to ..." indicates that the initial type satisfies the given property given the current inference constraints.
- The phrase "the expression *expr* has type *ty*" means the initial type of the expression is asserted to be equal to *ty*.

Additionally:

- The addition of constraints to the type inference constraint set fails if it causes an inconsistent set of constraints (§14.5). In this case either an error is reported or, if we are only attempting to *assert* the condition, the state of the inference procedure is left unchanged and the test fails.

6.2 Elaboration and Elaborated Expressions

Checking an expression generates an *elaborated expression* in a simpler, reduced language that effectively contains a fully resolved and annotated form of the expression. The elaborated expression provides more explicit information than the source form. For example, the elaborated form of `System.Console.WriteLine("Hello")` indicates exactly which overloaded method definition the call has resolved to. Elaborated forms are underlined in this specification, for example, `let x = 1 in x + x`.

Except for this extra resolution information, elaborated forms are syntactically a subset of syntactic expressions, and in some cases (such as constants) the elaborated form is the same as the source form. This specification uses the following elaborated forms:

- Constants
- Resolved value references: *path*
- Lambda expressions: `(fun ident -> expr)`
- Primitive object expressions
- Data expressions (tuples, union cases, array creation, record creation)
- Default initialization expressions
- Local definitions of values: `let ident = expr in expr`
- Local definitions of functions:
  `let rec ident = expr and ... and ident = expr in expr`

- Applications of methods and functions (with static overloading resolved)
- Dynamic type coercions: `expr :?> type`
- Dynamic type tests: `expr :? type`
- For-loops: `for ident in ident to ident do expr done`
- While-loops: `while expr do expr done`
- Sequencing: `expr; expr`
- Try-with: `try expr with expr`
- Try-finally: `try expr finally expr`
- The constructs required for the elaboration of pattern matching (§7).
  - Null tests
  - Switches on integers and other types
  - Switches on union cases
  - Switches on the runtime types of objects

The following constructs are used in the elaborated forms of expressions that make direct assignments to local variables and arrays and generate "byref" pointer values. The operations are loosely named after their corresponding primitive constructs in the CLI.

- Assigning to a byref-pointer: `expr <-stobj expr`
- Generating a byref-pointer by taking the address of a mutable value: `&path`.
- Generating a byref-pointer by taking the address of a record field: `&(expr.field)`
- Generating a byref-pointer by taking the address of an array element: `&(expr.[expr])`

Elaborated expressions form the basis for evaluation (see §6.9) and for the expression trees that *quoted expressions* return(see §6.8).

By convention, when describing the process of elaborating compound expressions, we omit the process of recursively elaborating sub-expressions.

6.3 Data Expressions

This section describes the following data expressions:

- Simple constant expressions
- Tuple expressions
- List expressions
- Array expressions
- Record expressions
- Copy-and-update record expressions
- Function expressions

- Object expressions
- Delayed expressions
- Computation expressions
- Sequence expressions
- Range expressions
- Lists via sequence expressions
- Arrays via sequence expressions
- Null expressions
- 'printf' formats

6.3.1 Simple Constant Expressions

Simple constant expressions are numeric, string, Boolean and unit constants. For example:

```
3y              // sbyte
32uy            // byte
17s             // int16
18us            // uint16
86              // int/int32
99u             // uint32
99999999L       // int64
10328273UL      // uint64
1.              // float/double
1.01            // float/double
1.01e10         // float/double
1.0f            // float32/single
1.01f           // float32/single
1.01e10f        // float32/single
99999999n       // nativeint       (System.IntPtr)
10328273un      // unativeint      (System.UIntPtr)
99999999I       // bigint          (System.Numerics.BigInteger or user-
specified)
'a'             // char            (System.Char)
"3"             // string          (String)
"c:\\home"      // string          (System.String)
@"c:\home"      // string          (Verbatim Unicode, System.String)
"ASCII"B        // byte[]
()              // unit            (FSharp.Core.Unit)
false           // bool            (System.Boolean)
true            // bool            (System.Boolean)
```

Simple constant expressions have the corresponding simple type and elaborate to the corresponding simple constant value.

Integer literals with the suffixes `Q`, `R`, `Z`, `I`, `N`, `G` are processed using the following syntactic translation:

```
xxxx<suffix>
```

For `xxxx` = `0`	→ `NumericLiteral<suffix>.FromZero()`
For `xxxx` = `1`	→ `NumericLiteral<suffix>.FromOne()`
For `xxxx` in the `Int32` range	→ `NumericLiteral<suffix>.FromInt32(xxxx)`
For `xxxx` in the `Int64` range	→ `NumericLiteral<suffix>.FromInt64(xxxx)`
For other numbers	→ `NumericLiteral<suffix>.FromString("xxxx")`

For example, defining a module `NumericLiteralZ` as below enables the use of the literal form `32Z` to generate a sequence of 32 ‘`Z`’ characters. No literal syntax is available for numbers outside the range of 32-bit integers.

```
module NumericLiteralZ =
    let FromZero() = ""
    let FromOne() = "Z"
    let FromInt32 n = String.replicate n "Z"
```

F# compilers may optimize on the assumption that calls to numeric literal functions always terminate, are idempotent, and do not have observable side effects.

6.3.2 Tuple Expressions

An expression of the form $expr_1$, `...`, $expr_n$ is a *tuple expression*. For example:

```
let three = (1,2,"3")
let blastoff = (10,9,8,7,6,5,4,3,2,1,0)
```

The expression has the type $(ty_1$ `*` `...` `*` $ty_n)$ for fresh types ty_1 … ty_n, and each individual expression e_i is checked using initial type ty_i.

Tuple types and expressions are translated into applications of a family of F# library types named `System.Tuple`. Tuple types ty_1 `*` `...` `*` ty_n are translated as follows:

- For n <= 7 the elaborated form is $\texttt{Tuple<}ty_1,\ldots,ty_n\texttt{>}$.
- For larger n, tuple types are shorthand for applications of the additional F# library type `System.Tuple<_>` as follows:
  - For n = 8 the elaborated form is $\texttt{Tuple<}ty_1,\ldots,ty_7,\texttt{Tuple<}ty_8\texttt{>>}$.
  - For 9 <= n the elaborated form is $\texttt{Tuple<}ty_1,\ldots,ty_7,ty_B\texttt{>}$ where ty_B is the converted form of the type $(ty_8 * \ldots * ty_n)$.

Tuple expressions $(expr_1,\ldots,expr_n)$ are translated as follows:

- For n <= 7 the elaborated form `new` $\texttt{Tuple<}ty_1,\ldots,ty_n\texttt{>}(expr_1,\ldots,expr_n)$.

- For n = 8 the elaborated form new Tuple<ty_1,…,ty_7,Tuple<ty_8>>($expr_1$,...,$expr_7$, new Tuple<ty_8>($expr_8$).
- For 9 <= n the elaborated form new Tuple<ty_1,...ty_7,ty_{8n}>($expr_1$,..., $expr_7$, new ty_{8n}(e_{8n}) where ty_{8n} is the type (ty_8*...* ty_n) and $expr_{8n}$ is the elaborated form of the expression
  $expr_8$,..., $expr_n$.

When considered as static types, tuple types are distinct from their encoded form. However, the encoded form of tuple values and types is visible in the F# type system through runtime types. For example, typeof<int * int> is equivalent to typeof<System.Tuple<int,int>>, and (1,2) has the runtime type System.Tuple<int,int>. Likewise, (1,2,3,4,5,6,7,8,9) has the runtime type Tuple<int,int,int,int,int,int,int,Tuple<int,int>>.

> Note: The above encoding is invertible and the substitution of types for type variables preserves this inversion. This means, among other things, that the F# reflection library can correctly report tuple types based on runtime System.Type values. The inversion is defined by:
>
> - For the runtime type Tuple<ty_1,...,ty_N> when n <= 7, the corresponding F# tuple type is ty_1 * ... * ty_N
> - For the runtime type Tuple<ty_1,..., Tuple<ty_N>> when n = 8, the corresponding F# tuple type is ty_1 * ... * ty_8
> - For the runtime type Tuple<ty_1,..., ty_7,ty_{Bn}> , if ty_{Bn} corresponds to the F# tuple type ty_8 * ... * ty_N, then the corresponding runtime type is ty_1 * ... * ty_N.
>
> Runtime types of other forms do not have a corresponding tuple type. In particular, runtime types that are instantiations of the eight-tuple type Tuple<_,_,_,_,_,_,_,_> must always have Tuple<_> in the final position. Syntactic types that have some other form of type in this position are not permitted, and if such an instantiation occurs in F# code or CLI library metadata that is referenced by F# code, an F# implementation may report an error.

6.3.3 List Expressions

An expression of the form [$expr_1$;...; $expr_n$] is a *list expression*. The initial type of the expression is asserted to be FSharp.Collections.List<ty> for a fresh type ty.

If ty is a named type, each expression $expr_i$ is checked using a fresh type ty' as its initial type, with the constraint ty' :> ty. Otherwise, each expression $expr_i$ is checked using ty as its initial type.

List expressions elaborate to uses of FSharp.Collections.List<_> as op_Cons($expr_1$,(op_Cons($expr_2$... op_Cons ($expr_n$, op_Nil)...) where op_Cons and op_Nil are the union cases with symbolic names :: and [] respectively.

6.3.4 Array Expressions

An expression of the form `[|`$expr_1$`;...;` $expr_n$ `|]` is an *array expression*. The initial type of the expression is asserted to be `ty[]` for a fresh type `ty`.

If this assertion determines that `ty` is a named type, each expression $expr_i$ is checked using a fresh type `ty'` as its initial type, with the constraint `ty' :> ty`. Otherwise, each expression $expr_i$ is checked using `ty` as its initial type.

Array expressions are a primitive elaborated form.

> Note: The F# implementation ensures that large arrays of constants of type `bool`, `char`, `byte`, `sbyte`, `int16`, `uint16`, `int32`, `uint32`, `int64`, and `uint64` are compiled to an efficient binary representation based on a call to `System.Runtime.CompilerServices.RuntimeHelpers.InitializeArray`.

6.3.5 Record Expressions

An expression of the form `{` $field\text{-}initializer_1$ `; … ;` $field\text{-}initializer_n$ `}` is a *record construction expression*. For example:

```
type Data = { Count : int; Name : string }
let data1 = { Count = 3; Name = "Hello"; }
let data2 = { Name = "Hello"; Count= 3 }
```

In the following example, `data4` uses a long identifier to indicate the relevant field:

```
module M =
    type Data = { Age : int; Name : string; Height : float }

let data3 = { M.Age = 17; M.Name = "John"; M.Height = 186.0 }
let data4 = { data3 with M.Name = "Bill"; M.Height = 176.0  }
```

Fields may also be referenced by using the name of the containing type:

```
module M2 =
    type Data = { Age : int; Name : string; Height : float }

let data5 = { M2.Data.Age = 17; M2.Data.Name = "John"; M2.Data.Height =
186.0 }
let data6 = { data5 with M2.Data.Name = "Bill"; M2.Data.Height=176.0  }

open M2
let data7 = { Data.Age = 17; Data.Name = "John"; Data.Height = 186.0 }
let data8 = { data5 with Data.Name = "Bill"; Data.Height=176.0  }
```

Each $field\text{-}initializer_i$ has the form $field\text{-}label_i$ `=` $expr_i$. Each $field\text{-}label_i$ is a *long-ident*, which must resolve to a field F_i in a unique record type *R* as follows:

- If $field\text{-}label_i$ is a single identifier *fld* and the initial type is known to be a record type `R<_,...,_>` that has field F_i with name *fld*, then the field label resolves to F_i.

- If *field-label*$_i$ is not a single identifier or if the initial type is a variable type, then the field label is resolved by performing *Field Label Resolution* (see §14.1) on *field-label*$_i$. This procedure results in a set of fields *FSet*$_i$. Each element of this set has a corresponding record type, thus resulting in a set of record types *RSet*$_i$. The intersection of all *RSet*$_i$ must yield a single record type *R*, and each field then resolves to the corresponding field in *R*.

  The set of fields must be complete. That is, each field in record type *R* must have exactly one field definition. Each referenced field must be accessible (see §10.5), as must the type *R*.

After all field labels are resolved, the overall record expression is asserted to be of type $R\langle ty_1,...,ty_N\rangle$ for fresh types $ty_1,...,ty_N$. Each $expr_i$ is then checked in turn. The initial type is determined as follows:

1. Assume the type of the corresponding field F_i in $R\langle ty_1,...,ty_N\rangle$ is fty_i
2. If the type of F_i prior to taking into account the instantiation $\langle ty_1,...,ty_N\rangle$ is a named type, then the initial type is a fresh type inference variable fty'_i with a constraint $fty'_i :> fty_i$.
3. Otherwise the initial type is fty_i.

Primitive record constructions are an elaborated form in which the fields appear in the same order as in the record type definition. Record expressions themselves elaborate to a form that may introduce local value definitions to ensure that expressions are evaluated in the same order that the field definitions appear in the original expression. For example:

```
type R = {b : int; a : int }
{ a = 1 + 1; b = 2 }
```

The expression on the last line elaborates to `let v = 1 + 1 in { b = 2; a = v }`.

Records expressions are also used for object initializations in additional object constructor definitions (§8.6.3). For example:

```
type C =
    val x : int
    val y : int
    new() = { x = 1; y = 2 }
```

> Note: The following record initialization form is deprecated:
>
> `{ new type with Field1 = expr1 and … and Fieldn = exprn }`
>
> The F# implementation allows the use of this form only with uppercase identifiers.
>
> F# code should not use this expression form. A future version of the F# language will issue a deprecation warning.

6.3.6 Copy-and-update Record Expressions

A *copy-and-update record expression* has the following form:

```
{ expr with field-initializers }
```

where *field-initializers* is of the following form:

$field\text{-}label_1$ = $expr_1$; … ; $field\text{-}label_n$ = $expr_n$

Each $field\text{-}label_i$ is a *long-ident*. In the following example, `data2` is defined by using such an expression:

```
type Data = { Age : int; Name : string; Height : float }
let data1 = { Age = 17; Name = "John"; Height = 186.0 }
let data2 = { data1 with Name = "Bill"; Height = 176.0  }
```

The expression *expr* is first checked with the same initial type as the overall expression. Next, the field definitions are resolved by using the same technique as for record expressions. Each field label must resolve to a field F_i in a single record type *R*, all of whose fields are accessible. After all field labels are resolved, the overall record expression is asserted to be of type $R<ty_1,...,ty_N>$ for fresh types $ty_1,...,ty_N$. Each $expr_i$ is then checked in turn with initial type that results from the following procedure:

1. Assume the type of the corresponding field F_i in $R<ty_1,...,ty_N>$ is fty_i.

2. If the type of F_i before considering the instantiation $<ty_1,...,ty_N>$ is a named type, then the initial type is a fresh type inference variable fty'_i with a constraint fty'_i :> fty_i.

3. Otherwise, the initial type is fty_i.

A copy-and-update record expression elaborates as if it were a record expression written as follows:

let *v* = *expr* in { $field\text{-}label_1$ = $expr_1$; … ; $field\text{-}label_n$ = $expr_n$; F_1 = $v.F_1$; ... ; F_M = $v.F_M$ }

where F_1 ... F_M are the fields of R that are not defined in *field-initializers* and *v* is a fresh variable.

6.3.7 Function Expressions

An expression of the form `fun` pat_1 ... pat_n -> *expr* is a *function expression*. For example:

```
(fun x -> x + 1)
(fun x y -> x + y)
(fun [x] -> x) // note, incomplete match
(fun (x,y) (z,w) -> x + y + z + w)
```

Function expressions that involve only variable patterns are a primitive elaborated form. Function expressions that involve non-variable patterns elaborate as if they had been written as follows:

```
fun v1 ... vn ->
    let pat1 = v1
    ...
    let patn = vn
    expr
```

No pattern matching is performed until all arguments have been received. For example, the following does not raise a `MatchFailureException` exception:

```
let f = fun [x] y -> y
let g = f []  // ok
```

However, if a third line is added, a `MatchFailureException` exception is raised:

```
let z = g 3 // MatchFailureException is raised
```

6.3.8 Object Expressions

An expression of the following form is an *object expression*:

```
{ new ty0 args-expr_opt object-members
  interface ty1 object-members1
  …
  interface tyn object-membersn }
```

In the case of the interface declarations, the *`object-members`* are optional and are considered empty if absent. Each set of *`object-members`* has the form:

```
with member-defns end_opt
```

Lexical filtering inserts simulated `$end` tokens when lightweight syntax is used.

Each member of an object expression members can use the keyword `member`, `override`, or `default`. The keyword `member` can be used even when overriding a member or implementing an interface.

For example:

```
let obj1 =
   { new System.Collections.Generic.IComparer<int> with
        member x.Compare(a,b) = compare (a % 7) (b % 7) }

let obj2 =
   { new System.Object() with
         member x.ToString () = "Hello" }

let obj3 =
   { new System.Object() with
         member x.ToString () = "Hello, base.ToString() = " +
base.ToString() }

let obj4 =
   { new System.Object() with
         member x.Finalize() = printfn "Finalize";
     interface System.IDisposable with
         member x.Dispose() = printfn "Dispose";  }
```

An object expression can specify additional interfaces beyond those required to fulfill the abstract slots of the type being implemented. For example, `obj4` in the preceding examples has static type `System.Object` but the object additionally implements the interface `System.IDisposable`. The additional interfaces are not part of the static type of the overall expression, but can be revealed through type tests.

Object expressions are statically checked as follows.

1. First, ty_0 to ty_n are checked to verify that they are named types. The overall type of the expression is ty_0 and is asserted to be equal to the initial type of the expression. However, if ty_0 is type equivalent to `System.Object` and ty_1 exists, then the overall type is instead ty_1.

2. The type ty_0 must be a class or interface type. The base construction argument *args-expr* must appear if and only if ty_0 is a class type. The type must have one or more accessible constructors; the call to these constructors is resolved and elaborated using *Method Application Resolution* (see §14.4). Except for ty_0, each ty_i must be an interface type.

3. The F# compiler attempts to associate each member with a unique *dispatch slot* by using *dispatch slot inference* (§14.7). If a unique matching dispatch slot is found, then the argument types and return type of the member are constrained to be precisely those of the dispatch slot.

4. The arguments, patterns, and expressions that constitute the bodies of all implementing members are next checked one by one to verify the following:

   - For each member, the “this” value for the member is in scope and has type ty_0.
   - Each member of an object expression can initially access the protected members of ty_0.
   - If the variable *base-ident* appears, it must be named `base`, and in each member a base variable with this name is in scope. Base variables can be used only in the member implementations of an object expression, and are subject to the same limitations as byref values described in §14.9.

The object must satisfy *dispatch slot checking* (§14.8) which ensures that a one-to-one mapping exists between dispatch slots and their implementations.

Object expressions elaborate to a primitive form. At execution, each object expression creates an object whose runtime type is compatible with all of the ty_i that have a dispatch map that is the result of *dispatch slot checking* (§14.8).

The following example shows how to both implement an interface and override a method from `System.Object`. The overall type of the expression is `INewIdentity`.

```
type public INewIdentity =
    abstract IsAnonymous : bool

let anon =
    { new System.Object() with
        member i.ToString() = "anonymous"
      interface INewIdentity with
        member i.IsAnonymous = true }
```

6.3.9 Delayed Expressions

An expression of the form `lazy` *`expr`* is a *delayed expression*. For example:

```
lazy (printfn "hello world")
```

is syntactic sugar for

```
new System.Lazy (fun () -> expr)
```

The behavior of the `System.Lazy` library type ensures that expression *`expr`* is evaluated on demand in response to a `.Value` operation on the lazy value.

6.3.10 Computation Expressions

The following expression forms are all *computation expressions*:

```
expr { for ... }
expr { let ... }
expr { let! ... }
expr { use ... }
expr { while ... }
expr { yield ... }
expr { yield! ... }
expr { try ... }
expr { return ... }
expr { return! ... }
```

More specifically, computation expressions have the following form:

```
builder-expr { cexpr }
```

where `cexpr` is, syntactically, the grammar of expressions with the additional constructs that are defined in `comp-expr`. Computation expressions are used for sequences and other non-standard interpretations of the F# expression syntax. For a fresh variable `b`, the expression

```
builder-expr { cexpr }
```

translates to

```
let b = builder-expr in {| cexpr |}C
```

The type of `b` must be a named type after the checking of `builder-expr`. The subscript indicates that custom operations (C) are acceptable but are not required.

If the inferred type of `b` has one or more of the `Run`, `Delay`, or `Quote` methods when `builder-expr` is checked, the translation involves those methods. For example, when all three methods exist, the same expression translates to:

```
let b = builder-expr in b.Run (<@ b.Delay(fun () -> {| cexpr |}C) >@)
```

If a **Run** method does not exist on the inferred type of `b`, the call to **`Run`** is omitted. Likewise, if no **`Delay`** method exists on the type of `b`, that call and the inner lambda are omitted, so the expression translates to the following:

```
let b = builder-expr in b.Run (<@ {| cexpr |}C >@)
```

Similarly, if a `Quote` method exists on the inferred type of `b`, at-signs `<@ @>` are placed around `{| cexpr |}C` or `b.Delay(fun () -> {| cexpr |}C)` if a `Delay` method also exists.

The translation `{| cexpr |}C` , which rewrites computation expressions to core language expressions, is defined recursively according to the following rules:

```
{| cexpr |}C ≡ T (cexpr, [], fun v -> v, true)
```

During the translation, we use the helper function `{| cexpr |}0` to denote a translation that does not involve custom operations:

```
{| cexpr |}0 ≡ T (cexpr, [], fun v -> v, false)
```

```
T(e, V, C, q) where e : the computation expression being translated
                    V : a set of scoped variables
                    C : continuation (or context where “e” occurs,
                        up to a hole to be filled by the result of
translating “e”)
                    q : Boolean that indicates whether a custom
operator is allowed
```

Then, T is defined for each computation expression `e`:

```
T(let p = e in ce, V, C, q) = T(ce, V ⊕ var(p), λv.C(let p = e in v), q)

T(let! p = e in ce, V, C, q) =  T(ce, V ⊕ var(p), λv.C(b.Bind(src(e),fun p
-> v), q)

T(yield e, V, C, q) = C(b.Yield(e))

T(yield! e, V, C, q) = C(b.YieldFrom(src(e)))

T(return e, V, C, q) = C(b.Return(e))

T(return! e, V, C, q) = C(b.ReturnFrom(src(e)))

T(use p = e in ce, V, C, q) = C(b.Using(e, fun p -> {| ce |}0))

T(use! p = e in ce, V, C, q) = C(b.Bind(src(e), fun p -> b.Using(p, fun p
-> {| ce |}0))

T(match e with pi -> cei, V, C, q) = C(match e with pi -> {| cei |}0)
```

```
T(while e do ce, V, C, q) = T(ce, V, λv.C(b.While(fun () -> e, b.Delay(fun () -> v))), q)

T(try ce with pi -> cei, V, C, q) =
    Assert(not q); C(b.TryWith(b.Delay(fun () -> {| ce |}0), fun pi -> {| cei |}0))

T(try ce finally e, V, C, q) =
    Assert(not q); C(b.TryFinally(b.Delay(fun () -> {| ce |}0), fun () -> e))

T(if e then ce, V, C, q) = T(ce, V, λv.C(if e then v else b.Zero()), q)

T(if e then ce1 else ce2, V, C, q) = Assert(not q); C(if e then  {| ce1 |}0) else  {| ce2 |}0)

T(for x = e1 to e2 do ce, V, C, q) = T(for x in e1 .. e2 do ce, V, C, q)

T(for p1 in e1 do joinOp p2 in e2 onWord (e3 eop e4) ce, V, C, q) =
    Assert(q); T(for pat(V) in b.Join(src(e1), src(e2), λp1.e3, λp2.e4,
    λp1. λp2.(p1,p2)) do ce, V , C, q)

T(for p1 in e1 do groupJoinOp p2 in e2 onWord (e3 eop e4) into p3 ce, V, C, q) =
    Assert(q); T(for pat(V) in b.GroupJoin(src(e1),
    src(e2), λp1.e3, λp2.e4, λp1. λp3.(p1,p3)) do ce, V , C, q)

T(for x in e do ce, V, C, q) = T(ce, V ⊕ {x}, λv.C(b.For(src(e), fun x -> v)), q)

T(do e in ce, V, C, q) = T(ce, V, λv.C(e; v), q)

T(do! e in ce, V, C, q) = T(let! () = e in ce, V, C, q)

T(joinOp p2 in e2 on (e3 eop e4) ce, V, C, q) =
    T(for pat(V) in C({| yield exp(V) |}0) do join p2 in e2 onWord (e3 eop e4) ce, V, λv.v, q)

T(groupJoinOp p2 in e2 onWord (e3 eop e4) into p3 ce, V, C, q) =
    T(for pat(V) in C({| yield exp(V) |}0) do groupJoin p2 in e2 on (e3 eop e4) into p3 ce,
    V, λv.v, q)

T([<CustomOperator("Cop")>]cop arg, V, C, q) = Assert (q); [| cop arg, C(b.Yield exp(V)) |]V

T([<CustomOperator("Cop", MaintainsVarSpaceUsingBind=true)>]cop arg; e, V, C, q) =
    Assert (q); CL (cop arg; e, V, C(b.Return exp(V)), false)
```

```
T([<CustomOperator("Cop")>]cop arg; e, V, C, q) =
    Assert (q); CL (cop arg; e, V, C(b.Yield exp(V)), false)

T(ce1; ce2, V, C, q) = C(b.Combine({| ce1 |}0, b.Delay(fun () -> {| ce2 |}0)))

T(do! e;, V, C, q) = T(let! () = src(e) in b.Return(), V, C, q)

T(e;, V, C, q) = C(e;b.Zero())
```

The following notes apply to the translations:

- The lambda expression (`fun f x -> b`) is represented by `λx.b`.
- The auxiliary function *var*`(p)` denotes a set of variables that are introduced by a pattern `p`. For example:
  `var(x) = {x}`, `var((x,y)) = {x,y}` or `var(S (x,y)) = {x,y}`
  where `S` is a type constructor.
- `⊕` is an update operator for a set `V` to denote extended variable spaces. It updates the existing variables. For example, `{x,y} ⊕ var((x,z))` becomes `{x,y,z}` where the second `x` replaces the first `x`.
- The auxiliary function *pat*`(V)` denotes a pattern tuple that represents a set of variables in `V`. For example, `pat({x,y})` becomes `(x,y)`, where `x` and `y` represent pattern expressions.
- The auxiliary function *exp*`(V)` denotes a tuple expression that represents a set of variables in `V`. For example, *exp*`({x,y})` becomes `(x,y)`, where `x` and `y` represent variable expressions.
- The auxiliary function *src*`(e)` denotes `b.Source(e)` if the innermost ForEach is from the user code instead of generated by the translation, and a builder `b` contains a `Source` method. Otherwise, *src*`(e)` denotes `e`.
- `Assert()` checks whether a custom operator is allowed. If not, an error message is reported. Custom operators may not be used within `try/with`, `try/finally`, `if/then/else`, `use`, `match`, or sequential execution expressions such as `(e1;e2)`. For example, you cannot use `if/then/else` in any computation expressions for which a builder defines any custom operators, even if the custom operators are not used.
- The operator `eop` denotes one of `=`, `?=`, `=?` or `?=?`.
- `joinOp` and `onWord` represent keywords for join-like operations that are declared in CustomOperationAttribute. For example, `[<CustomOperator("join", IsLikeJoin=true, JoinConditionWord="on")>]` declares "`join`" and "`on`".
- Similarly, `groupJoinOp` represents a keyword for groupJoin-like operations, declared in CustomOperationAttribute. For example, `[<CustomOperator("groupJoin", IsLikeGroupJoin=true, JoinConditionWord="on")>]` declares "`groupJoin`" and "`on`".
- The auxiliary translation **CL** is defined as follows:

```
CL (e1, V, e2, bind) where e1: the computation expression being
translated
                           V: a set of scoped variables
                           e2: the expression that will be translated
after e1 is done
```

```
                           bind: indicator if it is for Bind (true)
or iterator (false).
```

The following shows translations for the uses of `CL` in the preceding computation expressions:

```
CL (cop arg, V, e’, bind) = [| cop arg, e’ |]V

CL ([<MaintainsVariableSpaceUsingBind=true>]cop arg into p; e, V,
e’, bind) =
    T(let! p = e’ in e, [], λv.v, true)

CL (cop arg into p; e, V, e’, bind) = T(for p in e’ do e, [], λv.v,
true)

CL ([<MaintainsVariableSpace=true>]cop arg; e, V, e’, bind) =
    CL (e, V, [| cop arg, e’ |]V, true)

CL ([<MaintainsVariableSpaceUsingBind=true>]cop arg; e, V, e’, bind)
=
    CL (e, V, [| cop arg, e’ |]V, true)

CL (cop arg; e, V, e’, bind) = CL (e, [], [| cop arg, e’ |]V, false)

CL (e, V, e’, true) = T(let! pat(V) = e’ in e, V, λv.v, true)

CL (e, V, e’, false) = T(for pat(V) in e’ do e, V, λv.v, true)
```

- The auxiliary translation `[| e1, e2 |]V` is defined as follows:

```
[|[ e1, e2 |]V where e1: the custom operator available in a build
                  e2: the context argument that will be passed to a
custom operator
                  V: a list of bound variables
```

```
[|[<CustomOperator(" Cop")>] cop [<ProjectionParameter>] arg, e |]V =

      b.Cop (e, fun pat(V) -> arg)

[|[<CustomOperator("Cop")>] cop arg, e |]V = b.Cop (e, arg)
```

- The final two translation rules (for `do! e;` and `do! e;`) apply only for the final expression in the computation expression. The semicolon (`;`) can be omitted.

The following attributes specify custom operations:

- `CustomOperationAttribute` indicates that a member of a builder type implements a custom operation in a computation expression. The attribute has one parameter: the name of the custom operation. The operation can have the following properties:
    - `MaintainsVariableSpace` indicates that the custom operation maintains the variable space of a computation expression.

- `MaintainsVariableSpaceUsingBind` indicates that the custom operation maintains the variable space of a computation expression through the use of a bind operation.
- `AllowIntoPattern` indicates that the custom operation supports the use of 'into' immediately following the operation in a computation expression to consume the result of the operation.
- `IsLikeJoin` indicates that the custom operation is similar to a join in a sequence computation, which supports two inputs and a correlation constraint.
- `IsLikeGroupJoin` indicates that the custom operation is similar to a group join in a sequence computation, which support two inputs and a correlation constraint, and generates a group.
- `JoinConditionWord` indicates the names used for the 'on' part of the custom operator for join-like operators.

- `ProjectionParameterAttribute` indicates that, when a custom operation is used in a computation expression, a parameter is automatically parameterized by the variable space of the computation expression.

The following examples show how the translation works. Assume the following simple sequence builder:

```
type SimpleSequenceBuilder() =
    member __.For (source : seq<'a>, body : 'a -> seq<'b>) =
           seq { for v in source do yield! body v }
    member __.Yield (item:'a) : seq<'a> = seq { yield item }

let myseq = SimpleSequenceBuilder()
```

Then, the expression

```
myseq {
    for i in 1 .. 10 do
    yield i*i
    }
```

translates to

```
let b = myseq
b.For([1..10], fun i ->
   b.Yield(i*i))
```

`CustomOperationAttribute` allows us to define custom operations. For example, the simple sequence builder can have a custom operator, "where":

```
type SimpleSequenceBuilder() =
    member __.For (source : seq<'a>, body : 'a -> seq<'b>) =
           seq { for v in source do yield! body v }
    member __.Yield (item:'a) : seq<'a> = seq { yield item }
    [<CustomOperation("where")>]
    member __.Where (source : seq<'a>, f: 'a -> bool) : seq<'a> =
```

```
    Seq.filter f source

let myseq = SimpleSequenceBuilder()
```

Then, the expression

```
mysq {
    for i in 1 .. 10 do
    where (fun x -> x > 5)
    }
```

translates to

```
let b = myseq
b.Where(
   b.For([1..10], fun i ->
         b.Yield (i)),
   fun x -> x > 5)
```

`ProjectionParameterAttribute` automatically adds a parameter from the variable space of the computation expression. For example, `ProjectionParameterAttribute` can be attached to the second argument of the `where` operator:

```
type SimpleSequenceBuilder() =
    member __.For (source : seq<'a>, body : 'a -> seq<'b>) =
           seq { for v in source do yield! body v }
    member __.Yield (item:'a) : seq<'a> = seq { yield item }
    [<CustomOperation("where")>]
    member __.Where (source: seq<'a>, [<ProjectionParameter>]f: 'a ->
bool) : seq<'a> =
           Seq.filter f source

let myseq = SimpleSequenceBuilder()
```

Then, the expression

```
myseq {
    for i in 1 .. 10 do
    where (i > 5)
    }
```

translates to

```
let b = myseq
b.Where(
```

```
    b.For([1..10], fun i ->
          b.Yield (i)),
    fun i -> i > 5)
```

`ProjectionParameterAttribute` is useful when a let binding appears between ForEach and the custom operators. For example, the expression

```
myseq {
    for i in 1 .. 10 do
    let j = i * i
    where (i > 5 && j < 49)
    }
```

translates to

```
let b = myseq
b.Where(
   b.For([1..10], fun i ->
         let j = i * i
         b.Yield (i,j)),
   fun (i,j) -> i > 5 && j < 49)
```

Without `ProjectionParameterAttribute`, a user would be required to write "`fun (i,j) ->`" explicitly.

Now, assume that we want to write the condition "`where (i > 5 && j < 49)`" in the following syntax:

```
   where (i > 5)
   where (j < 49)
```

To support this style, the `where` custom operator should produce a computation that has the same variable space as the input computation. That is, `j` should be available in the second `where`. The following example uses the `MaintainsVariableSpace` property on the custom operator to specify this behavior:

```
type SimpleSequenceBuilder() =
    member __.For (source : seq<'a>, body : 'a -> seq<'b>) =
           seq { for v in source do yield! body v }
    member __.Yield (item:'a) : seq<'a> = seq { yield item }
    [<CustomOperation("where", MaintainsVariableSpace=true)>]
    member __.Where (source: seq<'a>, [<ProjectionParameter>]f: 'a ->
bool) : seq<'a> =
           Seq.filter f source

let myseq = SimpleSequenceBuilder()
```

Then, the expression

```
mysEq {
    for i in 1 .. 10 do
    let j = i * i
    where (i > 5)
    where (j < 49)
    }
```

translates to

```
let b = myseq
b.Where(
      b.Where(
      b.For([1..10], fun i ->
         let j = i * i
         b.Yield (i,j)),
      fun (i,j) -> i > 5),
   fun (i,j) -> j < 49)
```

When we may not want to produce the variable space but rather want to explicitly express the chain of the `where` operator, we can design this simple sequence builder in a slightly different way. For example, we can express the same expression in the following way:

```
myseq {
    for i in 1 .. 10 do
    where (i > 5) into j
    where (j*j < 49)
    }
```

In this example, instead of having a let-binding (for `j` in the previous example) and passing variable space (including `j`) down to the chain, we can introduce a special syntax that captures a value into a pattern variable and passes only this variable down to the chain, which is arguably more readable. For this case, `AllowIntoPattern` allows the custom operation to have an `into` syntax:

```
type SimpleSequenceBuilder() =
    member __.For (source : seq<'a>, body : 'a -> seq<'b>) =
           seq { for v in source do yield! body v }
    member __.Yield (item:'a) : seq<'a> = seq { yield item }

    [<CustomOperation("where", AllowIntoPattern=true)>]
    member __.Where (source: seq<'a>, [<ProjectionParameter>]f: 'a ->
bool) : seq<'a> =
        Seq.filter f source


let myseq = SimpleSequenceBuilder()
```

Then, the expression

```
myseq {
    for i in 1 .. 10 do
    where (i > 5) into j
    where (j*j < 49)
    }
```

translates to

```
let b = myseq
b.Where(
   b.For(
         b.Where(
               b.For([1..10], fun i -> b.Yield (i))
               fun i -> i>5),
         fun j -> b.Yield (j)),
   fun j -> j*j < 49)
```

Note that the `into` keyword is not customizable, unlike `join` and `on`.

In addition to `MaintainsVariableSpace`, `MaintainsVariableSpaceUsingBind` is provided to pass variable space down to the chain in a different way. For example:

```
type SimpleSequenceBuilder() =
    member __.For (source : seq<'a>, body : 'a -> seq<'b>) =
           seq { for v in source do yield! body v }
    member __.Return (item:'a) : seq<'a> = seq { yield item }
    member __.Bind (value , cont) = cont value

    [<CustomOperation("where", MaintainsVariableSpaceUsingBind=true,
AllowIntoPattern=true)>]
    member __.Where (source: seq<'a>, [<ProjectionParameter>]f: 'a ->
bool) : seq<'a> =
        Seq.filter f source

let myseq = SimpleSequenceBuilder()
```

The presence of `MaintainsVariableSpaceUsingBindAttribute` requires `Return` and `Bind` methods during the translation.

Then, the expression

```
myseq {
    for i in 1 .. 10 do
```

```
        where (i > 5 && i*i < 49) into j
        return j
        }
```

translates to

```
    let b = myseq
    b.Bind(
       b.Where(B.For([1..10], fun i -> b.Return (i)),
             fun i -> i > 5 && i*i < 49),
       fun j -> b.Return (j))
```

where `Bind` is called to capture the pattern variable `j`. Note that `For` and `Yield` are called to capture the pattern variable when `MaintainsVariableSpace` is used.

Certain properties on the `CustomOperationAttribute` introduce join-like operators. The following example shows how to use the `IsLikeJoin` property.

```
    type SimpleSequenceBuilder() =
        member __.For (source : seq<'a>, body : 'a -> seq<'b>) =
               seq { for v in source do yield! body v }
        member __.Yield (item:'a) : seq<'a> = seq { yield item }

        [<CustomOperation("merge", IsLikeJoin=true,
    JoinConditionWord="whenever")>]
        member __.Merge (src1:seq<'a>, src2:seq<'a>, ks1, ks2, ret) =
                  seq { for a in src1 do
                        for b in src2 do
                        if ks1 a = ks2 b then yield((ret a ) b)
                  }

    let myseq = SimpleSequenceBuilder()
```

`IsLikeJoin` indicates that the custom operation is similar to a join in a sequence computation; that is, it supports two inputs and a correlation constraint.

The expression

```
    myseq {
        for i in 1 .. 10 do
        merge j in [5 .. 15] whenever (i = j)
        yield j
        }
```

translates to

```
    let b = myseq
    b.For(
       b.Merge([1..10], [5..15],
```

```
          fun i -> i, fun j -> j,
        fun i -> fun j -> (i,j)),
fun j -> b.Yield (j))
```

This translation implicitly places type constraints on the expected form of the builder methods. For example, for the `async` builder found in the `FSharp.Control` library, the translation phase corresponds to implementing a builder of a type that has the following member signatures:

```
type AsyncBuilder with
    member For: seq<'T> * ('T -> Async<unit>) -> Async<unit>
    member Zero : unit -> Async<unit>
    member Combine : Async<unit> * Async<'T> -> Async<'T>
    member While : (unit -> bool) * Async<unit> -> Async<unit>
    member Return : 'T -> Async<'T>
    member Delay : (unit -> Async<'T>) -> Async<'T>
    member Using: 'T * ('T -> Async<'U>) -> Async<'U>
                           when 'U :> System.IDisposable
    member Bind: Async<'T> * ('T -> Async<'U>) -> Async<'U>
    member TryFinally: Async<'T> * (unit -> unit) -> Async<'T>
    member TryWith: Async<'T> * (exn -> Async<'T>) -> Async<'T>
```

The following example shows a common approach to implementing a new computation expression builder for a monad. The example uses computation expressions to define computations that can be partially run by executing them step-by-step, for example, up to a time limit.

```
/// Computations that can cooperatively yield by returning a
continuation
type Eventually<'T> =
    | Done of 'T
    | NotYetDone of (unit -> Eventually<'T>)

[<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix
)>]
module Eventually =

    /// The bind for the computations. Stitch 'k' on to the end of the
computation.
    /// Note combinators like this are usually written in the reverse
way,
    /// for example,
    ///     e |> bind k
    let rec bind k e =
        match e with
        | Done x -> NotYetDone (fun () -> k x)
        | NotYetDone work -> NotYetDone (fun () -> bind k (work()))

    /// The return for the computations.
    let result x = Done x
```

```
type OkOrException<'T> =
    | Ok of 'T
    | Exception of System.Exception

/// The catch for the computations. Stitch try/with throughout
/// the computation and return the overall result as an
OkOrException.
let rec catch e =
    match e with
    | Done x -> result (Ok x)
    | NotYetDone work ->
        NotYetDone (fun () ->
            let res = try Ok(work()) with | e -> Exception e
            match res with
            | Ok cont -> catch cont // note, a tailcall
            | Exception e -> result (Exception e))

/// The delay operator.
let delay f = NotYetDone (fun () -> f())

/// The stepping action for the computations.
let step c =
    match c with
    | Done _ -> c
    | NotYetDone f -> f ()

// The rest of the operations are boilerplate.

/// The tryFinally operator.
/// This is boilerplate in terms of "result", "catch" and "bind".
let tryFinally e compensation =
    catch (e)
    |> bind (fun res ->  compensation();
                         match res with
                         | Ok v -> result v
                         | Exception e -> raise e)

/// The tryWith operator.
/// This is boilerplate in terms of "result", "catch" and "bind".
let tryWith e handler =
    catch e
    |> bind (function Ok v -> result v | Exception e -> handler e)

/// The whileLoop operator.
/// This is boilerplate in terms of "result" and "bind".
let rec whileLoop gd body =
    if gd() then body |> bind (fun v -> whileLoop gd body)
    else result ()
```

```
    /// The sequential composition operator
    /// This is boilerplate in terms of "result" and "bind".
    let combine e1 e2 =
        e1 |> bind (fun () -> e2)

    /// The using operator.
    let using (resource: #System.IDisposable) f =
        tryFinally (f resource) (fun () -> resource.Dispose())

    /// The forLoop operator.
    /// This is boilerplate in terms of "catch", "result" and "bind".
    let forLoop (e:seq<_>) f =
        let ie = e.GetEnumerator()
        tryFinally (whileLoop (fun () -> ie.MoveNext())
                              (delay (fun () -> let v = ie.Current in f
v)))
                   (fun () -> ie.Dispose())


// Give the mapping for F# computation expressions.
type EventuallyBuilder() =
    member x.Bind(e,k)                    = Eventually.bind k e
    member x.Return(v)                    = Eventually.result v
    member x.ReturnFrom(v)                = v
    member x.Combine(e1,e2)               = Eventually.combine e1 e2
    member x.Delay(f)                     = Eventually.delay f
    member x.Zero()                       = Eventually.result ()
    member x.TryWith(e,handler)           = Eventually.tryWith e handler
    member x.TryFinally(e,compensation) = Eventually.tryFinally e
compensation
    member x.For(e:seq<_>,f)              = Eventually.forLoop e f
    member x.Using(resource,e)            = Eventually.using resource e

let eventually = new EventuallyBuilder()
```

After the computations are defined, they can be built by using `eventually { ... }`:

```
let comp =
    eventually { for x in 1 .. 2 do
                     printfn " x = %d" x
                 return 3 + 4 }
```

These computations can now be stepped. For example:

```
let step x = Eventually.step x
comp |> step
   // returns "NotYetDone <closure>"

comp |> step |> step
```

```
    // prints "x = 1"
    // returns "NotYetDone <closure>"

comp |> step |> step |> step |> step |> step |> step
    // prints "x = 1"
    // prints "x = 2"
    // returns "NotYetDone <closure>"

comp |> step |> step |> step |> step |> step |> step |> step |> step
    // prints "x = 1"
    // prints "x = 2"
    // returns "Done 7"
```

6.3.11 Sequence Expressions

An expression in one of the following forms is a *sequence expression*:

```
seq { comp-expr }
seq { short-comp-expr }
```

For example:

```
seq { for x in [ 1; 2; 3 ] do for y in [5; 6] do yield x + y }
seq { for x in [ 1; 2; 3 ] do yield x + x }
seq { for x in [ 1; 2; 3 ] -> x + x }
```

Logically speaking, sequence expressions can be thought of as computation expressions with a builder of type `FSharp.Collections.SeqBuilder`. This type can be considered to be defined as follows:

```
type SeqBuilder() =
    member x.Yield (v) = Seq.singleton v
    member x.YieldFrom (s:seq<_>) = s
    member x.Return (():unit) = Seq.empty
    member x.Combine (xs1,xs2) = Seq.append xs1 xs2
    member x.For (xs,g) = Seq.collect f xs
    member x.While (guard,body) =
SequenceExpressionHelpers.EnumerateWhile guard body
    member x.TryFinally (xs,compensation) =
        SequenceExpressionHelpers.EnumerateThenFinally xs compensation
    member x.Using (resource,xs) =
SequenceExpressionHelpers.EnumerateUsing resource xs
```

However, this builder type is not actually defined in the F# library. Instead, sequence expressions are elaborated directly as follows:

```
{| yield expr |}                      → Seq.singleton expr
{| yield! expr |}                     → expr
{| expr1 ; expr2 |}               → Seq.append {| expr1 |} {| expr2 |}
{| for pat in expr1 -> expr2 |}       → Seq.map (fun pat -> {| expr2 |})
expr1
{| for pat in expr1 do expr2 |}       → Seq.collect (fun pat -> {| expr2 |})
expr1
```

```
{| while expr₁ do expr₂ |}              → RuntimeHelpers.EnumerateWhile
                                             (fun () -> expr₁)
                                           {| expr₂ |})
{| try expr₁ finally expr₂ |}  → RuntimeHelpers.EnumerateThenFinally
                                             (| expr₁ |})
                                            (fun () -> expr₂)
{| use v = expr₁ in expr₂ |}   → let v = expr₁ in
                                   RuntimeHelpers.EnumerateUsing v {| expr₂
|}
{| let v = expr₁ in expr₂ |}   → let v = expr₁ in {| expr₂ |}
{| match expr with patᵢ -> exprᵢ |}   →.match expr with patᵢ -> {| cexprᵢ |}
{| expr₁ |}                    → expr₁ ; Seq.empty
{| if expr then expr₀ |}C              → if expr then {| expr₀ |}C else
Seq.empty
{| if expr then expr₀ else expr₁ |}    → if expr then {| expr₀ |}C else {|
expr₁ |}C
```

Here the use of `Seq` and `RuntimeHelpers` refers to the corresponding functions in `FSharp.Collections.Seq` and `FSharp.Core.CompilerServices.RuntimeHelpers` respectively. This means that a sequence expression generates an object of type `System.Collections.Generic.IEnumerable<ty>` for some type *ty*. Such an object has a `GetEnumerator` method that returns a `System.Collections.Generic.IEnumerator<ty>` whose `MoveNext`, `Current` and `Dispose` methods implement an on-demand evaluation of the sequence expressions.

6.3.12 Range Expressions

Expressions of the following forms are *range expressions*.

```
{ e1 .. e2 }
{ e1 .. e2 .. e3 }
seq { e1 .. e2 }
seq { e1 .. e2 .. e3 }
```

Range expressions generate sequences over a specified range. For example:

```
seq { 1 .. 10 } // 1; 2; 3; 4; 5; 6; 7; 8; 9; 10
seq { 1 .. 2 .. 10 } // 1; 3; 5; 7; 9
```

Range expressions involving $expr_1$ `..` $expr_2$ are translated to uses of the `(..)` operator, and those involving $expr_1$ `..` $expr_1$ `..` $expr_3$ are translated to uses of the `(.. ..)` operator:

```
seq { e1 .. e2 }     → (..) e₁ e₂
seq { e1 .. e2 .. e3 } → (.. ..) e₁ e₂ e₃
```

The default definition of these operators is in `FSharp.Core.Operators`. The `(..)` operator generates an `IEnumerable<_>` for the range of values between the start ($expr_1$) and finish ($expr_2$) values, using an increment of 1 (as defined by `FSharp.Core.LanguagePrimitives.GenericOne`). The `(.. ..)` operator generates an `IEnumerable<_>` for the range of values between the start ($expr_1$) and finish ($expr_3$) values, using an increment of $expr_2$.

The `seq` keyword, which denotes the type of computation expression, can be omitted for simple range expressions, but this is not recommended and might be deprecated in a future release. It is always preferable to explicitly mark the type of a computation expression.

Range expressions also occur as part of the translated form of expressions, including the following:

- `[ expr1 .. expr2 ]`
- `[| expr1 .. expr2 |]`
- `for var in expr1 .. expr2 do expr3`

A sequence iteration expression of the form `for var in expr1 .. expr2 do expr3 done` is sometimes elaborated as a simple for loop-expression (§6.5.7).

6.3.13 Lists via Sequence Expressions

A *list sequence expression* is an expression in one of the following forms

```
[ comp-expr ]
[ short-comp-expr ]
[ range-expr ]
```

In all cases `[ cexpr ]` elaborates to `FSharp.Collections.Seq.toList(seq { cexpr })`.

For example:

```
let x2 = [ yield 1; yield 2 ]

let x3 = [ yield 1
           if System.DateTime.Now.DayOfWeek = System.DayOfWeek.Monday
then
               yield 2]
```

6.3.14 Arrays Sequence Expressions

An expression in one of the following forms is an *array sequence expression*:

```
[| comp-expr |]
[| short-comp-expr |]
[| range-expr |]
```

In all cases `[| cexpr |]` elaborates to `FSharp.Collections.Seq.toArray(seq { cexpr })`.

For example:

```
let x2 = [| yield 1; yield 2 |]
let x3 = [| yield 1
            if System.DateTime.Now.DayOfWeek = System.DayOfWeek.Monday
then
                yield 2 |]
```

6.3.15 Null Expressions

An expression in the form `null` is a *null expression*. A null expression imposes a nullness constraint (§5.2.2, §5.4.8) on the initial type of the expression. The constraint ensures that the type directly supports the value `null`.

Null expressions are a primitive elaborated form.

6.3.16 'printf' Formats

Format strings are strings with `%` markers as format placeholders. Format strings are analyzed at compile time and annotated with static and runtime type information as a result of that analysis. They are typically used with one of the functions `printf`, `fprintf`, `sprintf`, or `bprintf` in the `FSharp.Core.Printf` module. Format strings receive special treatment in order to type check uses of these functions more precisely.

More concretely, a constant string is interpreted as a printf-style format string if it is expected to have the type `FSharp.Core.PrintfFormat<'Printer,'State,'Residue,'Result,'Tuple>`. The string is statically analyzed to resolve the generic parameters of the `PrintfFormat` type, of which `'Printer` and `'Tuple` are the most interesting:

- `'Printer` is the function type that is generated by applying a `printf`-like function to the format string.
- `'Tuple` is the type of the tuple of values that are generated by treating the string as a generator (for example, when the format string is used with a function similar to `scanf` in other languages).

A format placeholder has the following shape:

```
%[flags][width][.precision][type]
```

where:

`flags`

Are `0`, `-`, `+`, and the space character. The `#` flag is invalid and results in a compile-time error.

`width`

Is an integer that specifies the minimum number of characters in the result.

`precision`

Is the number of digits to the right of the decimal point for a floating-point type. .

`type`

Is as shown in the following table.

Placeholder string	Type
`%b`	`bool`
`%s`	`string`

Placeholder string	Type
`%c`	`char`
`%d`, `%i`	One of the basic integer types: `byte`, `sbyte`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `nativeint`, or `unativeint`
`%u`	Basic integer type formatted as an unsigned integer
`%x`	Basic integer type formatted as an unsigned hexadecimal integer with lowercase letters a through f.
`%X`	Basic integer type formatted as an unsigned hexadecimal integer with uppercase letters A through F.
`%o`	Basic integer type formatted as an unsigned octal integer.
`%e`, `%E`, `%f`, `%F`, `%g`, `%G`	`float` or `float32`
`%M`	`System.Decimal`
`%O`	`System.Object`
`%A`	Fresh variable type `'T`
`%a`	Formatter of type `'State -> 'T -> 'Residue` for a fresh variable type `'T`
`%t`	Formatter of type `'State -> 'Residue`

For example, the format string `"%s %d %s"` is given the type `PrintfFormat<(string -> int -> string -> 'd), 'b, 'c, 'd,(string * int * string)>` for fresh variable types `'b, 'c, 'd`. Applying `printf` to it yields a function of type `string -> int -> string -> unit`.

6.4 Application Expressions

6.4.1 Basic Application Expressions

Application expressions involve variable names, dot-notation lookups, function applications, method applications, type applications, and item lookups, as shown in the following table.

Expression	Description
long-ident-or-op	Long-ident lookup expression
expr '.' *long-ident-or-op*	Dot lookup expression
expr expr	Function or member application expression
expr(expr)	High precedence function or member application expression
`expr<types>`	Type application expression
`expr< >`	Type application expression with an empty type list
type expr	Simple object expression

The following are examples of application expressions:

```
System.Math.PI
System.Math.PI.ToString()
(3 + 4).ToString()
System.Environment.GetEnvironmentVariable("PATH").Length
System.Console.WriteLine("Hello World")
```

Application expressions may start with object construction expressions that do not include the new keyword:

```
System.Object()
System.Collections.Generic.List<int>(10)
System.Collections.Generic.KeyValuePair(3,"Three")
System.Object().GetType()
System.Collections.Generic.Dictionary<int,int>(10).[1]
```

If the *long-ident-or-op* starts with the special pseudo-identifier keyword `global`, F# resolves the identifier with respect to the global namespace—that is, ignoring all `open` directives (see §14.2). For example:

```
global.System.Math.PI
```

is resolved to `System.Math.PI` ignoring all `open` directives.

The checking of application expressions is described in detail as an algorithm in §14.2. To check an application expression, the expression form is repeatedly decomposed into a *lead* expression *expr* and a list of projections `projs` through the use of *Unqualified Lookup* (§14.2.1). This in turn uses procedures such as *Expression-Qualified Lookup* and *Method Application Resolution*.

As described in §14.2, checking an application expression results in an elaborated expression that contains a series of lookups and method calls. The elaborated expression may include:

- Uses of named values
- Uses of union cases
- Record constructions
- Applications of functions
- Applications of methods (including methods that access properties)
- Applications of object constructors
- Uses of fields, both static and instance
- Uses of active pattern result elements

Additional constructs may be inserted when resolving method calls into simpler primitives:

- The use of a method or value as a first-class function may result in a function expression.

  For example, `System.Environment.GetEnvironmentVariable` elaborates to:
  `(fun v -> System.Environment.GetEnvironmentVariable(v))`
  for some fresh variable `v`.

- The use of post-hoc property setters results in the insertion of additional assignment and sequential execution expressions in the elaborated expression.

  For example, `new System.Windows.Forms.Form(Text="Text")` elaborates to
  `let v = new System.Windows.Forms.Form() in v.set_Text("Text"); v`
  for some fresh variable `v`.

- The use of optional arguments results in the insertion of `Some(_)` and `None` data constructions in the elaborated expression.

For uses of active pattern results (see §10.2.4), for result *i* in an active pattern that has *N* possible results of types *types*, the elaborated expression form is a union case `ChoiceNOfi` of type `FSharp.Core.Choice<types>`.

6.4.2 Object Construction Expressions

An expression of the following form is an *object construction expression*:

```
new ty(e1 ... en)
```

An object construction expression constructs a new instance of a type, usually by calling a constructor method on the type. For example:

```
new System.Object()
new System.Collections.Generic.List<int>()
new System.Windows.Forms.Form (Text="Hello World")
new 'T()
```

The initial type of the expression is first asserted to be equal to *ty*. The type `ty` must not be an array, record, union or tuple type. If `ty` is a named class or struct type:

- *ty* must not be abstract.
- If *ty* is a struct type, *n* is 0, and *ty* does not have a constructor method that takes zero arguments, the expression elaborates to the default "zero-bit pattern" value for *ty*.
- Otherwise, the type must have one or more accessible constructors. The overloading between these potential constructors is resolved and elaborated by using *Method Application Resolution* (see §14.4).

If *ty* is a delegate type the expression is a *delegate implementation expression*.

- If the delegate type has an `Invoke` method that has the following signature `Invoke(`ty_1`,...,`ty_n`) ->` rty_A,

  then the overall expression must be in this form:

  `new ty(expr)` where *expr* has type ty_1 `-> ... ->` ty_n `->` rty_B

  If type rty_A is a CLI `void` type, then rty_B is `unit`, otherwise it is rty_A.

- If any of the types ty_i is a byref-type then an explicit function expression must be specified. That is, the overall expression must be of the form `new ty(fun` pat_1 … pat_n `->` $expr_{body}$`)`.

If *ty* is a type variable:

- There must be no arguments (that is, `n = 0`).
- The type variable is constrained as follows:

  ```
  ty : (new : unit -> ty)  -- CLI default constructor constraint
  ```

- The expression elaborates to a call to `FSharp.Core.LanguagePrimitives.IntrinsicFunctions.CreateInstance<ty>()`, which in turn calls `System.Activator.CreateInstance<ty>()`, which in turn uses CLI reflection to find and call the null object constructor method for type *ty*. On return from this function, any exceptions are wrapped by using `System.TargetInvocationException`.

6.4.3 Operator Expressions

Operator expressions are specified in terms of their shallow syntactic translation to other constructs. The following translations are applied in order:

```
infix-or-prefix-op e1  → (~infix-or-prefix-op) e1
prefix-op e1          → (prefix-op) e1
e1 infix-op e2          → (infix-op) e1 e2
```

> Note: When an operator that may be used as either an infix or prefix operator is used in prefix position, a tilde character ~ is added to the name of the operator during the translation process.

These rules are applied after applying the rules for dynamic operators (§6.4.4).

The parenthesized operator name is then treated as an identifier and the standard rules for unqualified name resolution (§14.1) in expressions are applied. The expression may resolve to a specific definition of a user-defined or library-defined operator. For example:

```
let (+++) a b = (a,b)
3 +++ 4
```

In some cases, the operator name resolves to a standard definition of an operator from the F# library. For example, in the absence of an explicit definition of (+),

```
3 + 4
```

resolves to a use of the infix operator `FSharp.Core.Operators.(+)`.

Some operators that are defined in the F# library receive special treatment in this specification. In particular:

- The *&expr* and *&&expr* address-of operators (§6.4.5)
- The *expr && expr* and *expr || expr* shortcut control flow operators (§6.5.4)
- The *%expr* and *%%expr* expression splice operators in quotations (§6.8.3)
- The library-defined operators, such as `+`, `-`, `*`, `/`, `%`, `**`, `<<<`, `>>>`, `&&&`, `|||`, and `^^^` (§18.2).

If the operator does not resolve to a user-defined or library-defined operator, the name resolution rules (§14.1) ensure that the operator resolves to an expression that implicitly uses a static member invocation expression (§0) that involves the types of the operands. This means that the effective behavior of an operator that is not defined in the F# library is to require a static member that has the same name as the operator, on the type of one of the operands of the operator. In the following code, the otherwise undefined operator `-->` resolves to the static member on the `Receiver` type, based on a type-directed resolution:

```
type Receiver(latestMessage:string) =
    static member (<--) (receiver:Receiver,message:string) =
        Receiver(message)

    static member (-->) (message,receiver:Receiver) =
        Receiver(message)

let r = Receiver "no message"

r <-- "Message One"

"Message Two" --> r
```

6.4.4 Dynamic Operator Expressions

Expressions of the following forms are *dynamic operator expressions:*

```
expr1 ? expr2
expr1 ? expr2 <- expr3
```

These expressions are defined by their syntactic translation:

```
expr ? ident                 → (?) expr "ident"
expr1 ? (expr2)              → (?) expr1 expr2
expr1 ? ident <- expr2       → (?<-) expr1 "ident" expr2
expr1 ? (expr2) <- expr3     → (?<-) expr1 expr2 expr3
```

Here `"ident"` is a string literal that contains the text of `ident`.

> Note: The F# core library `FSharp.Core.dll` does not define the `(?)` and `(?<-)` operators. However, user code may define these operators. For example, it is common to define the operators to perform a dynamic lookup on the properties of an object by using reflection.

This syntactic translation applies regardless of the definition of the `(?)` and `(?<-)` operators. However, it does not apply to uses of the parenthesized operator names, as in the following:

```
(?) x y
```

6.4.5 The AddressOf Operators

Under default definitions, expressions of the following forms are *address-of expressions,* called *byref-address-of expression* and *nativeptr-address-of expression,* respectively:

```
&expr
&&expr
```

Such expressions take the address of a mutable local variable, byref-valued argument, field, array element, or static mutable global variable.

For `&expr` and `&&expr` , the initial type of the overall expression must be of the form `byref<ty>` and `nativeptr<ty>` respectively, and the expression `expr` is checked with initial type `ty`.

The overall expression is elaborated recursively by taking the address of the elaborated form of `expr`, written *AddressOf*(`expr`, `DefinitelyMutates`), defined in §6.9.4.

Use of these operators may result in unverifiable or invalid common intermediate language (CIL) code; when possible, a warning or error is generated. In general, their use is recommended only:

- To pass addresses where `byref` or `nativeptr` parameters are expected.
- To pass a byref parameter on to a subsequent function.
- When required to interoperate with native code.

Addresses that are generated by the `&&` operator must not be passed to functions that are in tail call position. The F# compiler does not check for this.

Direct uses of `byref` types, `nativeptr` types, or values in the `FSharp.NativeInterop` module may result in invalid or unverifiable CIL code. In particular, `byref` and `nativeptr` types may NOT be used within named types such as tuples or function types.

When calling an existing CLI signature that uses a CLI pointer type `ty*`, use a value of type `nativeptr<ty>`.

> Note: The rules in this section apply to the following prefix operators, which are defined in the F# core library for use with one argument.
>
> `FSharp.Core.LanguagePrimitives.IntrinsicOperators.(~&)`
>
> `FSharp.Core.LanguagePrimitives.IntrinsicOperators.(~&&)`
>
> Other uses of these operators are not permitted.

6.4.6 Lookup Expressions

Lookup expressions are specified by syntactic translation:

```
e1.[eargs]                        → e1.get_Item(eargs)
e1.[eargs] <- e3                  → e1.set_Item(eargs, e3)
```

In addition, for the purposes of resolving expressions of this form, array types of rank 1, 2, 3, and 4 are assumed to support a type extension that defines an `Item` property that has the following signatures:

```
type 'T[] with
    member arr.Item : int -> 'T

type 'T[,] with
    member arr.Item : int * int -> 'T

type 'T[,,] with
    member arr.Item : int * int * int -> 'T

type 'T[,,,] with
    member arr.Item : int * int * int * int -> 'T
```

In addition, if type checking determines that the type of e_1 is a named type that supports the `DefaultMember` attribute, then the member name identified by the `DefaultMember` attribute is used instead of `Item`.

6.4.7 Slice Expressions

Slice expressions are defined by syntactic translation:

```
e1.[sliceArg1, ,,, sliceArgN]                → e1.GetSlice( args1,…,argsN)

e1.[sliceArg1, ,,, sliceArgN] <- expr        → e1.SetSlice( args1,…,argsN,
expr)
```

where each sliceArgN is one of the following and translated to argsN (giving one or two args) as indicated

```
*           → None, None
e1..        → Some e1, None
..e2        → None, Some e2
e1..e2      → Some e1, Some e2
idx         → idx
```

Because this is a shallow syntactic translation, the `GetSlice` and `SetSlice` name may be resolved by any of the relevant *Name Resolution* (§14.1) techniques, including defining the method as a type extension for an existing type.

For example, if a matrix type has the appropriate overloads of the GetSlice method (see below), it is possible to do the following:

```
matrix.[1..,*] -- get rows 1.. from a matrix (returning a matrix)
matrix.[1..3,*] -- get rows 1..3 from a matrix (returning a matrix)
matrix.[*,1..3] -- get columns 1..3from a matrix (returning a matrix)
matrix.[1..3,1,.3] -- get a 3x3 sub-matrix (returning a matrix)
matrix.[3,*] -- get row 3 from a matrix as a vector
matrix.[*,3] -- get column 3 from a matrix as a vector
```

In addition, CIL array types of rank 1 to 4 are assumed to support a type extension that defines a method `GetSlice` that has the following signature:

```
type 'T[] with
    member arr.GetSlice : ?start1:int * ?end1:int -> 'T[]

type 'T[,] with
    member arr.GetSlice : ?start1:int * ?end1:int * ?start2:int *
?end2:int -> 'T[,]
    member arr.GetSlice : idx1:int * ?start2:int * ?end2:int -> 'T[]
    member arr.GetSlice : ?start1:int * ?end1:int * idx2:int  -> 'T[]

type 'T[,,] with
    member arr.GetSlice : ?start1:int * ?end1:int * ?start2:int *
?end2:int *
```

```
                              ?start3:int * ?end3:int
                                 -> 'T[,,]

type 'T[,,,] with
    member arr.GetSlice : ?start1:int * ?end1:int * ?start2:int *
?end2:int *
                          ?start3:int * ?end3:int * ?start4:int *
?end4:int
                             -> 'T[,,,]
```

In addition, CIL array types of rank 1 to 4 are assumed to support a type extension that defines a method `SetSlice` that has the following signature:

```
type 'T[] with
    member arr.SetSlice : ?start1:int * ?end1:int * values:T[] -> unit

type 'T[,] with
    member arr.SetSlice : ?start1:int * ?end1:int * ?start2:int *
?end2:int *
                          values:T[,] -> unit
    member arr.SetSlice : idx1:int * ?start2:int * ?end2:int *
values:T[] -> unit
    member arr.SetSlice : ?start1:int * ?end1:int * idx2:int *
values:T[] -> unit

type 'T[,,] with
    member arr.SetSlice : ?start1:int * ?end1:int * ?start2:int *
?end2:int *
                          ?start3:int * ?end3:int  * values:T[,,] ->
unit

type 'T[,,,] with
    member arr.SetSlice : ?start1:int * ?end1:int * ?start2:int *
?end2:int *
                          ?start3:int * ?end3:int * ?start4:int *
?end4:int *
                          values:T[,,,] -> unit
```

6.4.8 Member Constraint Invocation Expressions

An expression of the following form is a member constraint invocation expression:

```
(static-typars : (member-sig) expr)
```

Type checking proceeds as follows:

1. The expression is checked with initial type *ty*.
2. A statically resolved member constraint is applied (§5.2.3): *static-typars* : (*member-sig*)
3. *ty* is asserted to be equal to the return type of the constraint.
4. *expr* is checked with an initial type that corresponds to the argument types of the constraint.

The elaborated form of the expression is a member invocation. For example:

```
let inline speak (a: ^a) =
    let x = (^a : (member Speak: unit -> string) (a))
    printfn "It said: %s" x
    let y = (^a : (member MakeNoise: unit -> string) (a))
    printfn "Then it went: %s" y

type Duck() =
    member x.Speak() = "I'm a duck"
    member x.MakeNoise() = "quack"
type Dog() =
    member x.Speak() = "I'm a dog"
    member x.MakeNoise() = "grrrr"

let x = new Duck()
let y = new Dog()
speak x
speak y
```

Outputs:

```
It said: I'm a duck
Then it went: quack
It said: I'm a dog
Then it went: grrrr
```

6.4.9 Assignment Expressions

An expression of the following form is an *assignment expression*:

```
expr1 <- expr2
```

A modified version of *Unqualified Lookup* (§14.2.1) is applied to the expression *expr*$_1$ using a fresh expected result type *ty*, thus producing an elaborate expression *expr*$_1$. The last qualification for *expr*$_1$ must resolve to one of the following constructs:

- An invocation of a property with a setter method. The property may be an indexer.

  Type checking incorporates *expr*$_2$ as the last argument in the method application resolution for the setter method. The overall elaborated expression is a method call to this setter property and includes the last argument.

- A mutable value *path* of type *ty*.

  Type checking of *expr*$_2$ uses the expected result type *ty* and generates an elaborated expression *expr*$_2$. The overall elaborated expression is an assignment to a value reference &*path* <-$_{\text{stobj}}$ *expr*$_2$.

- A reference to a value *path* of type `byref<`*ty*`>`.

Type checking of *expr₂* uses the expected result type `ty` and generates an elaborated expression *expr₂*. The overall elaborated expression is an assignment to a value reference `path <-stobj expr2`.

- A reference to a mutable field `expr1a.field` with the actual result type `ty`.

  Type checking of $expr_2$ uses the expected result type `ty` and generatesan elaborated expression $expr_2$. The overall elaborated expression is an assignment to a field (see §6.9.4):

  ```
  AddressOf(expr1a.field, DefinitelyMutates) <-stobj expr2
  ```

- A array lookup `expr1a.[expr1b]` where $expr_{1a}$ has type `ty[]`.

  Type checking of $expr_2$ uses the expected result type `ty` and generates thean elaborated expression $expr_2$. The overall elaborated expression is an assignment to a field (see §6.9.4):

  ```
  AddressOf(expr1a.[expr1b] , DefinitelyMutates) <-stobj expr2
  ```

Note: Because assignments have the preceding interpretations, local values must be mutable so that primitive field assignments and array lookups can mutate their immediate contents. In this context, "immediate" contents means the contents of a mutable value type. For example, given

```
[<Struct>]
type SA =
    new(v) = { x = v }
    val mutable x : int

[<Struct>]
type SB =
    new(v) = { sa = v }
    val mutable sa : SA

let s1 = SA(0)
let mutable s2 = SA(0)
let s3 = SB(0)
let mutable s4 = SB(0)
```

Then these are not permitted:

```
s1.x <- 3
s3.sa.x <- 3
```

and these are:

```
s2.x <- 3
s4.sa.x <- 3
s4.sa <- SA(2)
```

6.5 Control Flow Expressions

6.5.1 Parenthesized and Block Expressions

A *parenthesized expression* has the following form:

```
(expr)
```

A *block expression* has the following form:

```
begin expr end
```

The expression *expr* is checked with the same initial type as the overall expression.

The elaborated form of the expression is simply the elaborated form of *expr*.

6.5.2 Sequential Execution Expressions

A *sequential execution expression* has the following form:

```
expr1; expr2
```

For example:

```
printfn "Hello"; printfn "World"; 3
```

The `;` token is optional when both of the following are true:

- The expression $expr_2$ occurs on a subsequent line that starts in the same column as $expr_1$.
- The current pre-parse context that results from the syntax analysis of the program text is a **SeqBlock** (§15).

When the semicolon is optional, parsing inserts a `$sep` token automatically and applies an additional syntax rule for lightweight syntax (§15.1.1). In practice, this means that code can omit the `;` token for sequential execution expressions that implement functions or immediately follow tokens such as `begin` and `(`.

The expression $expr_1$ is checked with an arbitrary initial type *ty*. After checking $expr_1$, *ty* is asserted to be equal to `unit`. If the assertion fails, a warning rather than an error is reported. The expression $expr_2$ is then checked with the same initial type as the overall expression.

Sequential execution expressions are a primitive elaborated form.

6.5.3 Conditional Expressions

A *conditional expression* has the following form:s

```
if expr1a then expr1b
elif expr3a then expr2b
…
elif exprna then exprnb
else exprlast
```

The *elif* and *else* branches may be omitted. For example:

```
if (1 + 1 = 2) then "ok" else "not ok"
if (1 + 1 = 2) then printfn "ok"
```

Conditional expressions are equivalent to pattern matching on Boolean values. For example, the following expression forms are equivalent:

```
if expr1 then expr2 else expr3
match (expr1:bool) with true -> expr2 | false -> expr3
```

If the *else* branch is omitted, the expression is a *sequential conditional expression* and is equivalent to:

```
match (expr1:bool) with true -> expr2 | false -> ()
```

with the exception that the initial type of the overall expression is first asserted to be `unit`.

6.5.4 Shortcut Operator Expressions

Under default definitions, expressions of the following form are respectively an *shortcut and expression* and a *shortcut or expression*:

```
expr && expr
expr || expr
```

These expressions are defined by their syntactic translation:

```
expr1 && expr2                   → if expr1 then expr2 else false
expr1 || expr2                   → if expr1 then true else expr2
```

Note: The rules in this section apply when the following operators, as defined in the F# core library, are applied to two arguments.

```
FSharp.Core.LanguagePrimitives.IntrinsicOperators.(&&)
FSharp.Core.LanguagePrimitives.IntrinsicOperators.(||)
```

If the operator is not immediately applied to two arguments, it is interpreted as a strict function that evaluates both its arguments before use.

6.5.5 Pattern-Matching Expressions and Functions

A *pattern-matching expression*has the following form:

```
match expr with rules
```

Pattern matching is used to evaluate the given expression and select a rule (§7). For example:

```
match (3, 2) with
  | 1, j -> printfn "j = %d" j
  | i, 2 -> printfn "i = %d" i
  | _    -> printfn "no match"
```

A *pattern-matching function* is an expression of the following form:

```
function rules
```

A pattern-matching function is syntactic sugar for a single-argument function expression that is followed by immediate matches on the argument. For example:

```
function
  | 1, j -> printfn "j = %d" j
  | _    -> printfn "no match"
```

is syntactic sugar for the following, where `x` is a fresh variable:

```
fun x ->
  match x with
  | 1, j -> printfn "j = %d" j
  | _    -> printfn "no match"
```

6.5.6 Sequence Iteration Expressions

An expression of the following form is a *sequence iteration expression*:

```
for pat in expr1 do expr2 done
```

The `done` token is optional if *expr₂* appears on a later line and is indented from the column position of the `for` token. In this case, parsing inserts a `$done` token automatically and applies an additional syntax rule for lightweight syntax (§15.1.1).

For example:

```
for x, y in [(1, 2); (3, 4)] do
    printfn "x = %d, y = %d" x y
```

The expression $expr_1$ is checked with a fresh initial type ty_{expr}, which is then asserted to be a subtype of type `IEnumerable<ty>`, for a fresh type *ty*. If the assertion succeeds, the expression elaborates to the following, where *v* is of type `IEnumerator<ty>` and *pat* is a pattern of type *ty*:

```
let v = expr1.GetEnumerator()
try
    while (v.MoveNext()) do
        match v.Current with
        | pat -> expr2
        | _ -> ()
finally
    match box(v) with
    | :? System.IDisposable as d -> d.Dispose()
    | _ -> ()
```

If the assertion fails, the type ty_{expr} may also be of any static type that satisfies the "collection pattern" of CLI libraries. If so, the *enumerable extraction* process is used to enumerate the type. In particular, ty_{expr} may be any type that has an accessible `GetEnumerator` method that accepts zero arguments and returns a value that has accessible `MoveNext` and `Current` properties. The type of *pat* is the same as the return type of the `Current` property on the enumerator value. However, if the `Current` property has return type `obj` and the collection type *ty* has an `Item` property with a more specific (non-object) return type ty_2, type ty_2 is used instead, and a dynamic cast is inserted to convert `v.Current` to ty_2.

A sequence iteration of the form

```
for var in expr1 .. expr2 do expr3 done
```

where the type of $expr_1$ or $expr_2$ is equivalent to `int`, is elaborated as a simple for-loop expression (§6.5.7)

6.5.7 Simple for-Loop Expressions

An expression of the following form is a *simple for loop expression*:

```
for var = expr1 to expr2 do expr3 done
```

The `done` token is optional when `e2` appears on a later line and is indented from the column position of the `for` token. In this case, a `$done` token is automatically inserted, and an additional syntax rule for lightweight syntax applies (§15.1.1). For example:

```
for x = 1 to 30 do
    printfn "x = %d, x^2 = %d" x (x*x)
```

The bounds $expr_1$ and $expr_2$ are checked with initial type `int`. The overall type of the expression is `unit`. A warning is reported if the body $expr_3$ of the `for` loop does not have static type `unit`.

The following shows the elaborated form of a simple for-loop expression for fresh variables `start` and `finish`:

```
let start = expr1 in
let finish = expr2 in
for var = start to finish do expr3 done
```

For-loops over ranges that are specified by variables are a primitive elaborated form. When executed, the iterated range includes both the starting and ending values in the range, with an increment of 1.

An expression of the form

```
for var in expr1 .. expr2 do expr3 done
```

is always elaborated as a simple for-loop expression whenever the type of $expr_1$ or $expr_2$ is equivalent to `int`.

6.5.8 While Expressions

A *while loop expression* has the following form:

```
while expr1 do expr2 done
```

The done token is optional when $expr_2$ appears on a subsequent line and is indented from the column position of the while. In this case, a $done token is automatically inserted, and an additional syntax rule for lightweight syntax applies (§15.1.1).

For example:

```
while System.DateTime.Today.DayOfWeek = System.DayOfWeek.Monday do
    printfn "I don't like Mondays"
```

The overall type of the expression is unit. The expression $expr_1$ is checked with initial type bool. A warning is reported if the body $expr_2$ of the while loop cannot be asserted to have type unit.

6.5.9 Try-with Expressions

A *try-with expression* has the following form:

```
try expr with rules
```

For example:

```
try "1" with _ -> "2"

try
    failwith "fail"
with
   | Failure msg -> "caught"
   | :? System.InvalidOperationException -> "unexpected"
```

Expression *expr* is checked with the same initial type as the overall expression. The pattern matching clauses are then checked with the same initial type and with input type System.Exception.

Try-with expressions are a primitive elaborated form.

6.5.10 Reraise Expressions

A *reraise expression* is an application of the reraise F# library function. This function must be applied to an argument and can be used only on the immediate right-hand side of *rules* in a try-with expression.

```
try
    failwith "fail"
with e -> printfn "Failing"; reraise()
```

Note: The rules in this section apply to any use of the function FSharp.Core.Operators.reraise, which is defined in the F# core library.

When executed, reraise() continues exception processing with the original exception information.

6.5.11 Try-finally Expressions

A *try-finally expression* has the following form:

```
try expr1 finally expr2
```

For example:

```
try "1" finally printfn "Finally!"

try
    failwith "fail"
finally
    printfn "Finally block"
```

Expression $expr_1$ is checked with the initial type of the overall expression. Expression $expr_2$ is checked with arbitrary initial type, and a warning occurs if this type cannot then be asserted to be equal to `unit`.

Try-finally expressions are a primitive elaborated form.

6.5.12 Assertion Expressions

An *assertion expression* has the following form:

```
assert expr
```

The expression `assert expr` is syntactic sugar for `System.Diagnostics.Debug.Assert(expr)`

> Note: `System.Diagnostics.Debug.Assert` is a conditional method call. This means that assertions are triggered only if the DEBUG conditional compilation symbol is defined.

6.6 Definition Expressions

A *definition expression* has one of the following forms:

```
let function-defn in expr
let value-defn in expr
let rec function-or-value-defns in expr
use ident = expr1 in expr
```

Such an expression establishes a local function or value definition within the lexical scope of *expr* and has the same overall type as *expr*.

In each case, the `in` token is optional if *expr* appears on a subsequent line and is aligned with the token `let`. In this case, a `$in` token is automatically inserted, and an additional syntax rule for lightweight syntax applies (§15.1.1)

For example:

```
let x = 1
x + x
```

and

```
let x, y = ("One", 1)
x.Length + y
```

and

```
let id x = x in (id 3, id "Three")
```

and

```
let swap (x, y) = (y,x)
List.map swap [ (1, 2); (3, 4) ]
```

and

```
let K x y = x in List.map (K 3) [ 1; 2; 3; 4 ]
```

Function and value definitions in expressions are similar to function and value definitions in class definitions (§8.6.1.3), modules (§10.2.1), and computation expressions (§6.3.10), with the following exceptions:

- Function and value definitions in expressions may not define explicit generic parameters (§5.3). For example, the following expression is rejected:

  ```
  let f<'T> (x:'T) = x in f 3
  ```

- Function and value definitions in expressions are not public and are not subject to arity analysis (§14.10).
- Any custom attributes that are specified on the declaration, parameters, and/or return arguments are ignored and result in a warning. As a result, function and value definitions in expressions may not have the `ThreadStatic` or `ContextStatic` attribute.

6.6.1 Value Definition Expressions

A value definition expression has the following form:

```
let value-defn in expr
```

where *value-defn* has the form:

$\texttt{mutable}_{opt}\ \textit{access}_{opt}\ \textit{pat}\ \textit{typar-defns}_{opt}\ \textit{return-type}_{opt}\ \texttt{=}\ \textit{rhs-expr}$

Checking proceeds as follows:

1. Check the *value-defn* (§14.6), which defines a group of identifiers $ident_j$ with inferred types ty_j
2. Add the identifiers $ident_j$ to the name resolution environment, each with corresponding type ty_j.
3. Check the body *expr* against the initial type of the overall expression.

In this case, the following rules apply:

- If *pat* is a single value pattern `ident`, the resulting elaborated form of the entire expression is

```
let ident1 <typars1> = expr1 in
body-expr
```

  where $ident_1$, $typars_1$ and $expr_1$ are defined in §14.6.

- Otherwise, the resulting elaborated form of the entire expression is

```
let tmp <typars1… typarsn> = expr in
let ident1 <typars1> = expr1 in
…
let identn <typarsn> = exprn in
body-expr
```

  where *tmp* is a fresh identifier and $ident_i$, $typars_i$, and $expr_i$ all result from the compilation of the pattern *pat* (§7) against the input *tmp*.

Value definitions in expressions may be marked as `mutable`. For example:

```
let mutable v = 0
while v < 10 do
    v <- v + 1
    printfn "v = %d" v
```

Such variables are under the same restrictions as values of type `byref<_>` (§14.9), and are implicitly dereferenced each time they are used.

6.6.2 Function Definition Expressions

A function definition expression has the form:

```
let function-defn in expr
```

where *function-defn* has the form:

```
inline_opt access_opt ident-or-op typar-defns_opt pat1 ... patn return-type_opt
= rhs-expr
```

Checking proceeds as follows:

1. Check the *function-defn* (§14.6), which defines $ident_1$, ty_1, $typars_1$ and $expr_1$

2. Add the identifier $ident_1$ to the name resolution environment, each with corresponding type ty_1.

3. Check the body *expr* against the initial type of the overall expression.

The resulting elaborated form of the entire expression is

```
let ident1 <typars1> = expr1 in
expr
```

where *ident*$_1$, *typars*$_1$ and *expr*$_1$ are as defined in §14.6.

6.6.3 Recursive Definition Expressions

An expression of the following form is a *recursive definition expression*:

```
let rec function-or-value-defns in expr
```

The defined functions and values are available for use within their own definitions—that is can be used within any of the expressions on the right-hand side of `function-or-value-defns`. Multiple functions or values may be defined by using `let rec … and …`. For example:

```
let test() =
    let rec twoForward count =
        printfn "at %d, taking two steps forward" count
        if count = 1000 then "got there!"
        else oneBack (count + 2)
    and oneBack count =
        printfn "at %d, taking one step back " count
        twoForward (count - 1)

    twoForward 1

test()
```

In the example, the expression defines a set of recursive functions. If one or more recursive values are defined, the recursive expressions are analyzed for safety (§14.6.6). This may result in warnings (including some reported as compile-time errors) and runtime checks.

6.6.4 Deterministic Disposal Expressions

A *deterministic disposal expression* has the form:

```
use ident = expr1 in expr2
```

For example:

```
use inStream = System.IO.File.OpenText "input.txt"
let line1 = inStream.ReadLine()
let line2 = inStream.ReadLine()
(line1,line2)
```

The expression is first checked as an expression of form `let ident = expr1 in expr2` (§**Error! Reference source not found.**), which results in an elaborated expression of the following form:

```
let ident1 : ty1 = expr1 in expr2.
```

Only one value may be defined by a deterministic disposal expression, and the definition is not generalized (§14.6.7). The type *ty*$_1$, is then asserted to be a subtype of `System.IDisposable`. If the dynamic value of the expression after coercion to type `obj` is non-null, the `Dispose` method is called on the value when the value goes out of scope. Thus the overall expression elaborates to this:

```
let ident1 : ty1 = expr1
try expr2
finally (match (ident :> obj) with
          | null -> ()
          | _ -> (ident :> System.IDisposable).Dispose())
```

6.7 Type-Related Expressions

6.7.1 Type-Annotated Expressions

A *type-annotated expression* has the following form, where *ty* indicates the static type of *expr*:

```
expr : ty
```

For example:

```
(1 : int)
let f x = (x : string) + x
```

When checked, the initial type of the overall expression is asserted to be equal to *ty*. Expression *expr* is then checked with initial type *ty*. The expression elaborates to the elaborated form of *expr*. This ensures that information from the annotation is used during the analysis of `expr` itself.

6.7.2 Static Coercion Expressions

A *static coercion expression*—also called a flexible type constraint—has the following form:

```
expr :> ty
```

The expression `upcast` *expr* is equivalent to *expr* `:> _`, so the target type is the same as the initial type of the overall expression. For example:

```
(1 :> obj)
("Hello" :> obj)
([1;2;3] :> seq<int>).GetEnumerator()
(upcast 1 : obj)
```

The initial type of the overall expression is *ty*. Expression *expr* is checked using a fresh initial type ty_e, with constraint ty_e `:>` *ty*. Static coercions are a primitive elaborated form.

6.7.3 Dynamic Type-Test Expressions

A dynamic type-test expression has the following form:

```
expr :? ty
```

For example:

```
((1 :> obj) :? int)
((1 :> obj) :? string)
```

The initial type of the overall expression is `bool`. Expression *expr* is checked using a fresh initial type ty_e. After checking:

- The type ty_e must not be a variable type.
- A warning is given if the type test will always be true and therefore is unnecessary.
- The type ty_e must not be sealed.
- If type *ty* is sealed, or if *ty* is a variable type, or if type ty_e is not an interface type, then *ty* `:>` ty_e is asserted.

Dynamic type tests are a primitive elaborated form.

6.7.4 Dynamic Coercion Expressions

A dynamic coercion expression has the following form:

```
expr :?> ty
```

The expression `downcast` *e1* is equivalent to *expr* `:?> _`, so the target type is the same as the initial type of the overall expression. For example:

```
let obj1 = (1 :> obj)
(obj1 :?> int)
(obj1 :?> string)
(downcast obj1 : int)
```

The initial type of the overall expression is *ty*. Expression *expr* is checked using a fresh initial type ty_e. After these checks:

- The type ty_e must not be a variable type.
- A warning is given if the type test will always be true and therefore is unnecessary.
- The type ty_e must not be sealed.
- If type *ty* is sealed, or if *ty* is a variable type, or if type ty_e is not an interface type, then *ty* `:>` ty_e is asserted.

Dynamic coercions are a primitive elaborated form.

6.8 Quoted Expressions

An expression in one of these forms is a quoted expression:

```
<@ expr @>

<@@ expr @@>
```

The former is a *strongly typed quoted expression*, and the latter is a *weakly typed quoted expression*. In both cases, the expression forms capture the enclosed expression in the form of a typed abstract syntax tree.

The exact nodes that appear in the expression tree are determined by the elaborated form of *expr* that type checking produces.

For details about the nodes that may be encountered, see the documentation for the `FSharp.Quotations.Expr` type in the F# core library. In particular, quotations may contain:

- References to module-bound functions and values, and to type-bound members. For example:

  ```
  let id x = x
  let f (x : int) = <@ id 1 @>
  ```

  In this case the value appears in the expression tree as a node of kind `FSharp.Quotations.Expr.Call`.

- A type, module, function, value, or member that is annotated with the `ReflectedDefinition` attribute. If so, the expression tree that forms its definition may be retrieved dynamically using the `FSharp.Quotations.Expr.TryGetReflectedDefinition`.

  If the `ReflectedDefinition` attribute is applied to a type or module, it will be recursively applied to all members, too.

- References to defined values, such as the following:

  ```
  let f (x : int) = <@ x + 1 @>
  ```

  Such a value appears in the expression tree as a node of kind `FSharp.Quotations.Expr.Value`.

- References to generic type parameters or uses of constructs whose type involves a generic parameter, such as the following:

  ```
  let f (x:'T) = <@ (x, x) : 'T * 'T @>
  ```

  In this case, the actual value of the type parameter is implicitly substituted throughout the type annotations and types in the generated expression tree.

As of F# 3.1, the following limitations apply to quoted expressions:

- Quotations may not use object expressions.
- Quotations may not define expression-bound functions that are themselves inferred to be generic. Instead, expression-bound functions should either include type annotations to refer to a specific type or should be written by using module-bound functions or class-bound members.

6.8.1 Strongly Typed Quoted Expressions

A strongly typed quoted expression has the following form:

```
<@ expr @>
```

For example:

```
<@ 1 + 1 @>
```

```
<@ (fun x -> x + 1) @>
```

In the first example, the type of the expression is `FSharp.Quotations.Expr<int>`. In the second example, the type of the expression is `FSharp.Quotations.Expr<int -> int>`.

When checked, the initial type of a strongly typed quoted expression `<@ expr @>` is asserted to be of the form `FSharp.Quotations.Expr<ty>` for a fresh type `ty`. The expression `expr` is checked with initial type `ty`.

6.8.2 Weakly Typed Quoted Expressions

A *weakly typed quoted expression* has the following form:

```
<@@ expr @@>
```

Weakly typed quoted expressions are similar to strongly quoted expressions but omit any type annotation. For example:

```
<@@ 1 + 1 @@>

<@@ (fun x -> x + 1) @@>
```

In both these examples, the type of the expression is `FSharp.Quotations.Expr`.

When checked, the initial type of a weakly typed quoted expression `<@@ expr @@>` is asserted to be of the form `FSharp.Quotations.Expr`. The expression `expr` is checked with fresh initial type `ty`.

6.8.3 Expression Splices

Both strongly typed and weakly typed quotations may contain expression splices in the following forms:

```
%expr
%%expr
```

These are respectively strongly typed and weakly typed splicing operators.

6.8.3.1 Strongly Typed Expression Splices

An expression of the following form is a *strongly typed expression splice*:

```
%expr
```

For example, given

```
open FSharp.Quotations
let f1 (v:Expr<int>) = <@ %v + 1 @>
let expr = f1 <@ 3 @>
```

the identifier `expr` evaluates to the same expression tree as `<@ 3 + 1 @>`. The expression tree for `<@ 3 @>` replaces the splice in the corresponding expression tree node.

A strongly typed expression splice may appear only in a quotation. Assuming that the splice expression `%expr` is checked with initial type `ty`, the expression `expr` is checked with initial type `FSharp.Quotations.Expr<ty>`.

> Note: The rules in this section apply to any use of the prefix operator `FSharp.Core.ExtraTopLevelOperators.(~%)`. Uses of this operator must be applied to an argument and may only appear in quoted expressions.

6.8.3.2 Weakly Typed Expression Splices

An expression of the following form is a *weakly typed expression splice*:

```
%%expr
```

For example, given

```
open FSharp.Quotations
let f1 (v:Expr) = <@ %%v + 1 @>
let tree = f1 <@@ 3 @@>
```

the identifier `tree` evaluates to the same expression tree as `<@ 3 + 1 @>`. The expression tree replaces the splice in the corresponding expression tree node.

A weakly typed expression splice may appear only in a quotation. Assuming that the splice expression `%%expr` is checked with initial type `ty`, then the expression `expr` is checked with initial type `FSharp.Quotations.Expr`. No additional constraint is placed on `ty`.

Additional type annotations are often required for successful use of this operator.

> Note: The rules in this section apply to any use of the prefix operator `FSharp.Core.ExtraTopLevelOperators.(~%%)`, which is defined in the F# core library. Uses of this operator must be applied to an argument and may only occur in quoted expressions.

6.9 Evaluation of Elaborated Forms

At runtime, execution evaluates expressions to values. The evaluation semantics of each expression form are specified in the subsections that follow.

6.9.1 Values and Execution Context

The execution of elaborated F# expressions results in values. Values include:

- Primitive constant values
- The special value `null`
- References to object values in the global heap of object values
- Values for value types, containing a value for each field in the value type
- Pointers to mutable locations (including static mutable locations, mutable fields and array elements)

Evaluation assumes the following evaluation context:

- A global heap of object values. Each object value contains:
    - A runtime type and dispatch map
    - A set of fields with associated values
    - For array objects, an array of values in index order
    - For function objects, an expression which is the body of the function
    - An optional *union case label*, which is an identifier
    - A closure environment that assigns values to all variables that are referenced in the method bodies that are associated with the object
- A global environment that maps runtime-type/name pairs to values.Each name identifies a static field in a type definition or a value in a module.
- A local environment mapping names of variables to values.
- A local stack of active exception handlers, made up of a stack of try/with and try/finally handlers.

Evaluation may also raise an exception. In this case, the stack of active exception handlers is processed until the exception is handled, in which case additional expressions may be executed (for try/finally handlers), or an alternative expression may be evaluated (for try/with handlers), as described below.

6.9.2 Parallel Execution and Memory Model

In a concurrent environment, evaluation may involve both multiple active computations (multiple concurrent and parallel threads of execution) and multiple pending computations (pending callbacks, such as those activated in response to an I/O event).

If multiple active computations concurrently access mutable locations in the global environment or heap, the atomicity, read, and write guarantees of the underlying CLI implementation apply. The guarantees are related to the logical sizes and characteristics of values, which in turn depend on their type:

- F# reference types are guaranteed to map to CLI reference types. In the CLI memory model, reference types have atomic reads and writes.
- F# value types map to a corresponding CLI value type that has corresponding fields. Reads and writes of sizes less than or equal to one machine word are atomic.

The `VolatileField` attribute marks a mutable location as volatile in the compiled form of the code.

Ordering of reads and writes from mutable locations may be adjusted according to the limitations specified by the CLI memory model. The following example shows situations in which changes to read and write order can occur, with annotations about the order of reads:

```
type ClassContainingMutableData() =
    let value = (1, 2)
```

```
    let mutable mutableValue = (1, 2)
    [<VolatileField>]
    let mutable volatileMutableValue = (1, 2)
    member x.ReadValues() =
        // Two reads on an immutable value
        let (a1, b1) = value

        // One read on mutableValue, which may be duplicated according
        // to ECMA CLI spec.
        let (a2, b2) = mutableValue

        // One read on volatileMutableValue, which may not be
duplicated.
        let (a3, b3) = volatileMutableValue

        a1, b1, a2, b2, a3, b3

    member x.WriteValues() =
        // One read on mutableValue, which may be duplicated according
        // to ECMA CLI spec.
        let (a2, b2) = mutableValue

        // One write on mutableValue.
        mutableValue <- (a2 + 1, b2 + 1)

        // One read on volatileMutableValue, which may not be
duplicated.
        let (a3, b3) = volatileMutableValue

        // One write on volatileMutableValue.
        volatileMutableValue <- (a3 + 1, b3 + 1)

let obj = ClassContainingMutableData()
Async.Parallel [ async { return obj.WriteValues() };
                 async { return obj.WriteValues() };
                 async { return obj.ReadValues() };
                 async { return obj.ReadValues() } ]
```

6.9.3 Zero Values

Some types have a *zero value*. The zero value is the“default” value for the type in the CLI execution environment. The following types have the following zero values:

- For reference types, the `null` value.
- For value types, the value with all fields set to the zero value for the type of the field. The zero value is also computed by the F# library function `Unchecked.defaultof<`*`ty`*`>`.

6.9.4 Taking the Address of an Elaborated Expression

When the F# compiler determines the elaborated forms of certain expressions, it must compute a "reference" to an elaborated expression *expr*, written *AddressOf*(*expr*, *mutation*). The *AddressOf* operation is used internally within this specification to indicate the elaborated forms of address-of expressions, assignment expressions, and method and property calls on objects of variable and value types.

The *AddressOf* operation is computed as follows:

- If *expr* has form *path* where *path* is a reference to a value with type `byref<ty>`, the elaborated form is `&path`.
- If *expr* has form $expr_a$`.field` where *field* is a mutable, non-readonly CLI field, the elaborated form is `&(`*AddressOf*($expr_a$)`.field)`.
- If *expr* has form $expr_a$`.[`$expr_b$`]` where the operation is an array lookup, the elaborated form is `&(`*AddressOf*($expr_a$)`.[`$expr_b$`])`.
- If *expr* has any other form, the elaborated form is `&v`,where *v* is a fresh mutable local value that is initialized by adding `let v = expr` to the overall elaborated form for the entire assignment expression. This initialization is known as a *defensive copy* of an immutable value. If *expr* is a struct, *expr* is copied each time the *AddressOf* operation is applied, which results in a different address each time. To keep the struct in place, the field that contains it should be marked as `mutable.`

The *AddressOf* operation is computed with respect to *mutation*, which indicates whether the relevant elaborated form uses the resulting pointer to change the contents of memory. This assumption changes the errors and warnings reported.

- If *mutation* is `DefinitelyMutates`, then an error is given if a defensive copy must be created.
- If *mutation* is `PossiblyMutates`, then a warning is given if a defensive copy arises.

An F# compiler can optionally upgrade `PossiblyMutates` to `DefinitelyMutates` for calls to property setters and methods named `MoveNext` and `GetNextArg`, which are the most common cases of struct-mutators in CLI library design. This is done by the F# compiler.

> Note:In F#, the warning "copy due to possible mutation of value type" is a level 4 warning and is not reported when using the default settings of the F# compiler. This is because the majority of value types in CLI libraries are immutable. This is warning number 52 in the F# implementation.
>
> CLI libraries do not include metadata to indicate whether a particular value type is immutable. Unless a value is held in arrays or locations marked mutable, or a value type is known to be immutable to the F# compiler, F# inserts copies to ensure that inadvertent mutation does not occur.

6.9.5 Evaluating Value References

At runtime, an elaborated value reference `v` is evaluated by looking up the value of `v` in the local environment.

6.9.6 Evaluating Function Applications

At runtime, an elaborated application of a function $f\ e_1 \ldots\ e_n$ is evaluated as follows:

- The expressions f and $e_1 \ldots\ e_n$, are evaluated.
- If f evaluates to a function value with closure environment **E**, arguments $v_1 \ldots\ v_m$, and body *expr*, where $m <= n$, then **E** is extended by mapping $v_1 \ldots\ v_m$ to the argument values for $e_1 \ldots\ e_m$. The expression *expr* is then evaluated in this extended environment and any remaining arguments applied.
- If f evaluates to a function value with more than n arguments, then a new function value is returned with an extended closure mapping n additional formal argument names to the argument values for $e_1 \ldots\ e_m$.

The result of calling the `obj.GetType()` method on the resulting object is under-specified (see §6.9.24).

6.9.7 Evaluating Method Applications

At runtime an elaborated application of a method is evaluated as follows:

- The elaborated form is e_0`.M(`$e_1,\ldots,e_n$`)` for an instance method or `M(`$e_1,\ldots,e_n$`)` for a static method.
- The (optional) e_0 and $e_1,\ldots,e_n$ are evaluated in order.
- If e_0 evaluates to `null`, a `NullReferenceException` is raised.
- If the method is declared `abstract`—that is, if it is a virtual dispatch slot—then the body of the member is chosen according to the dispatch maps of the value of e_0 (§14.8).
- The formal parameters of the method are mapped to corresponding argument values. The body of the method member is evaluated in the resulting environment .

6.9.8 Evaluating Union Cases

At runtime, an elaborated use of a union case *Case*`(`$e_1,\ldots,e_n$`)` for a union type *ty* is evaluated as follows:

- The expressions $e_1,\ldots,e_n$ are evaluated in order.
- The result of evaluation is an object value with union case label *Case* and fields given by the values of $e_1,\ldots,e_n$.
- If the type *ty* uses `null` as a representation (§5.4.8) and *Case* is the single union case without arguments, the generated value is `null`.
- The runtime type of the object is either *ty* or an internally generated type that is compatible with *ty*.

6.9.9 Evaluating Field Lookups

At runtime, an elaborated lookup of a CLI or F# fields is evaluated as follows:

- The elaborated form is *expr*`.`*F* for an instance field or *F* for a static field.

- The(optional) *expr* is evaluated.
- If *expr* evaluates to `null`, a `NullReferenceException` is raised.
- The value of the field is read from either the global field table or the local field table associated with the object.

6.9.10 Evaluating Array Expressions

At runtime, an elaborated array expression `[| e1; ... ; en |]ty` is evaluated as follows:

- Each expression e_1 ... e_n is evaluated in order.
- The result of evaluation is a new array of runtime type `ty[]` that contains the resulting values in order.

6.9.11 Evaluating Record Expressions

At runtime, an elaborated record construction `{ field1 = e1; ... ; fieldn = en }ty` is evaluated as follows:

- Each expression e_1 ... e_n is evaluated in order.
- The result of evaluation is an object of type *ty* with the given field values

6.9.12 Evaluating Function Expressions

At runtime, an elaborated function expression `(fun v1 ... vn -> expr)` is evaluated as follows:

- The expression evaluates to a function object with a closure that assigns values to all variables that are referenced in *expr* and a function body that is *expr*.
- The values in the closure are the current values of those variables in the execution environment.
- The result of calling the `obj.GetType()` method on the resulting object is under-specified (see §6.9.24).

6.9.13 Evaluating Object Expressions

At runtime, elaborated object expressions

```
{ new ty0 args-expropt object-members
      interface ty1 object-members1
      ...
      interface tyn object-membersn }
```

is evaluated as follows:

- The expression evaluates to an object whose runtime type is compatible with all of the ty_i and which has the corresponding dispatch map (§14.8). If present, the base construction expression `ty0 (args-expr)` is executed as the first step in the construction of the object.
- The object is given a closure that assigns values to all variables that are referenced in *expr*.
- The values in the closure are the current values of those variables in the execution environment.

The result of calling the `obj.GetType()` method on the resulting object is under-specified (see §6.9.24).

6.9.14 Evaluating Definition Expressions

At runtime, each elaborated definition *pat* `=` *expr* is evaluated as follows:

- The expression *expr* is evaluated.
- The expression is then matched against *pat* to produce a value for each variable pattern (§7.2) in *pat*.
- These mappings are added to the local environment.

6.9.15 Evaluating Integer For Loops

At runtime, an integer for loop `for` *var* `=` $expr_1$ `to` $expr_2$ `do` $expr_3$ `done` is evaluated as follows:

- Expressions $expr_1$ and $expr_2$ are evaluated once to values v_1 and v_2.
- The expression $expr_3$ is evaluated repeatedly with the variable *var* assigned successive values in the range of v_1 up to v_2.
- If v_1 is greater than v_2, then $expr_3$ is never evaluated.

6.9.16 Evaluating While Loops

As runtime, while-loops `while` $expr_1$ `do` $expr_2$ `done` are evaluated as follows:

- Expression $expr_1$ is evaluated to a value v_1.
- If v_1 is `true`, expression $expr_2$ is evaluated, and the expression `while` $expr_1$ `do` $expr_2$ `done` is evaluated again.
- If v_1 is `false`, the loop terminates and the resulting value is `null` (the representation of the only value of type `unit`)

6.9.17 Evaluating Static Coercion Expressions

At runtime, elaborated static coercion expressions of the form *expr* `:>` *ty* are evaluated as follows:

- Expression *expr* is evaluated to a value *v*.
- If the static type of *e* is a value type, and *ty* is a reference type, *v* is *boxed*; that is, *v* is converted to an object on the heap with the same field assignments as the original value. The expression evaluates to a reference to this object.
- Otherwise, the expression evaluates to *v*.

6.9.18 Evaluating Dynamic Type-Test Expressions

At runtime, elaborated dynamic type test expressions *expr* `:?` *ty* are evaluated as follows:

1. Expression *expr* is evaluated to a value *v*.

2. If *v* is `null`, then:

- If ty_e uses `null` as a representation (§5.4.8), the result is `true`.
- Otherwise the expression evaluates to `false`.

3. If *v* is not `null` and has runtime type *vty* which dynamically converts to *ty* (§5.4.10), the expression evaluates to `true`. However, if *ty* is an enumeration type, the expression evaluates to `true` if and only if *ty* is precisely *vty*.

6.9.19 Evaluating Dynamic Coercion Expressions

At runtime, elaborated dynamic coercion expressions *expr* :?> *ty* are evaluated as follows:

1. Expression *expr* is evaluated to a value *v*.

2. If *v* is `null`:

- If ty_e uses `null` as a representation (§5.4.8), the result is the `null` value.
- Otherwise a `NullReferenceException` is raised.

3. If *v* is not `null`:

- If *v* has dynamic type *vty* which *dynamically converts to* *ty* (§5.4.10), the expression evaluates to the dynamic conversion of *v* to *ty*.
  - If *vty* is a reference type and *ty* is a value type, then *v* is *unboxed*; that is, *v* is converted from an object on the heap to a struct value with the same field assignments as the object. The expression evaluates to this value.
  - Otherwise, the expression evaluates to *v*.
- Otherwise an `InvalidCastException` is raised.

Expressions of the form *expr* :?> *ty* evaluate in the same way as the F# library function unbox<*ty*>(*expr*).

> Note: Some F# types—most notably the `option<_>` type—use `null` as a representation for efficiency reasons (§5.4.8),. For these types, boxing and unboxing can lose type distinctions. For example, contrast the following two examples:
>
> ```
> > (box([]:string list) :?> int list);;
> System.InvalidCastException…
>
> > (box(None:string option) :?> int option);;
> val it : int option = None
> ```
>
> In the first case, the conversion from an empty list of strings to an empty list of integers (after first boxing) fails. In the second case, the conversion from a string option to an integer option (after first boxing) succeeds.

6.9.20 Evaluating Sequential Execution Expressions

At runtime, elaborated sequential expressions $expr_1$; $expr_2$ are evaluated as follows:

- The expression $expr_1$ is evaluated for its side effects and the result is discarded.
- The expression $expr_2$ is evaluated to a value v_2 and the result of the overall expression is v_2.

6.9.21 Evaluating Try-with Expressions

At runtime, elaborated try-with expressions `try expr1 with rules` are evaluated as follows:

- The expression $expr_1$ is evaluated to a value v_1.
- If no exception occurs, the result is the value v_1.
- If an exception occurs, the pattern rules are executed against the resulting exception value.
    - If no rule matches, the exception is reraised.
    - If a rule `pat -> expr2` matches, the mapping `pat = v1` is added to the local environment, and $expr_2$ is evaluated.

6.9.22 Evaluating Try-finally Expressions

At runtime, elaborated try-finally expressions `try expr1 finally expr2` are evaluated as follows:

- The expression $expr_1$ is evaluated.
    - If the result of this evaluation is a value v , then $expr_2$ is evaluated.
        1) If this evaluation results in an exception, then the overall result is that exception.
        2) If this evaluation does not result in an exception, then the overall result is v.
    - If the result of this evaluation is an exception, then $expr_2$ is evaluated.
        3) If this evaluation results in an exception, then the overall result is that exception.
        4) If this evaluation does not result in an exception, then the original exception is re-raised.

6.9.23 Evaluating AddressOf Expressions

At runtime, an elaborated address-of expression is evaluated as follows. First, the expression has one of the following forms:

- `&path` where `path` is a static field.
- `&(expr.field)`
- `&(expra.[exprb])`
- `&v` where `v` is a local mutable value.

The expression evaluates to the address of the referenced local mutable value, mutable field, or mutable static field.

Note: The underlying CIL execution machinery that F# uses supports covariant arrays, as evidenced by the fact that the type `string[]` *dynamically converts to* `obj[]` (§5.4.10). Although this feature is rarely used in F#, its existence means that array assignments and taking the address of array elements may fail at runtime with a `System.ArrayTypeMismatchException` if the runtime type of the target array does not match the runtime type of the element being assigned. For example, the following code fails at runtime:

```
let F(x: byref<obj>) = ()

let a = Array.zeroCreate<obj> 10
let b = Array.zeroCreate<string> 10
F(&a.[0])
let bb = ((b :> obj) :?> obj[])
// The next line raises a System.ArrayTypeMismatchException
exception.
F(&bb.[1])
```

6.9.24 Values with Underspecified Object Identity and Type Identity

The CLI and F# support operations that detect object identity—that is, whether two object references refer to the same "physical" object. For example, `System.Object.ReferenceEquals(obj₁, obj₂)` returns `true` if the two object references refer to the same object. Similarly, `System.Runtime.CompilerServices.RuntimeHelpers.GetHashCode()` returns a hash code that is partly based on physical object identity, and the `AddHandler` and `RemoveHandler` operations (which register and unregister event handlers) are based on the object identity of delegate values.

The results of these operations are underspecified when used with values of the following F# types:

- Function types
- Tuple types
- Immutable record types
- Union types
- Boxed immutable value types

For two values of such types, the results of `System.Object.ReferenceEquals` and `System.Runtime.CompilerServices.RuntimeHelpers.GetHashCode` are underspecified; however, the operations terminate and do not raise exceptions. An implementation of F# is not required to define the results of these operations for values of these types.

For function values and objects that are returned by object expressions, the results of the following operations are underspecified in the same way:

- `Object.GetHashCode()`
- `Object.GetType()`

For union types the results of the following operations are underspecified in the same way:

- `Object.GetType()`

7. Patterns

Patterns are used to perform simultaneous case analysis and decomposition on values together with the `match`, `try...with`, `function`, `fun`, and `let` expression and declaration constructs. Rules are attempted in order from top to bottom and left to right. The syntactic forms of patterns are shown in the subsequent table.

```
rule :=
    pat pattern-guard_opt -> expr    -- pattern, optional guard and
action

pattern-guard := when expr

pat :=
    const                      -- constant pattern
    long-ident pat-param_opt pat_opt    -- named pattern
    _                                   -- wildcard pattern
    pat as ident               -- "as" pattern
    pat '|' pat                -- disjunctive pattern
    pat '&' pat                -- conjunctive pattern
    pat :: pat                          -- "cons" pattern
    pat : type                          -- pattern with type constraint
    pat,...,pat                -- tuple pattern
    (pat)                      -- parenthesized pattern
    list-pat                   -- list pattern
    array-pat                           -- array pattern
    record-pat                          -- record pattern
    :? atomic-type             -- dynamic type test pattern
    :? atomic-type as ident-- dynamic type test pattern
    null                       -- null-test pattern
    attributes pat             -- pattern with attributes

list-pat :=
    [ ]
    [ pat ; ... ; pat ]

array-pat :=
    [| |]
    [| pat ; ... ; pat |]

record-pat :=
    { field-pat ; ... ; field-pat }

atomic-pat :=
    pat :     one of
          const  long-ident  list-pat  record-pat  array-pat  (pat)
          :? atomic-type
          null  _

field-pat := long-ident = pat
```

```
pat-param :=
      | const
      | long-ident
      | [ pat-param ; ... ; pat-param ]
      | ( pat-param, ..., pat-param )
      | long-ident pat-param
      | pat-param : type
      | <@ expr @>
      | <@@ expr @@>
      | null

pats :=  pat , ... , pat
field-pats := field-pat ; ... ; field-pat
rules := '|'opt rule '|' ... '|' rule
```

Patterns are elaborated to expressions through a process called *pattern match compilation*. This reduces pattern matching to *decision trees* which operate on an input value, called the *pattern input*. The decision tree is composed of the following constructs:

- Conditionals on integers and other constants
- Switches on union cases
- Conditionals on runtime types
- Null tests
- Value definitions
- An array of pattern-match targets referred to by index

7.1 Simple Constant Patterns

The pattern `const` is a *constant pattern* which matches values equal to the given constant. For example:

```
let rotate3 x =
   match x with
   | 0 -> "two"
   | 1 -> "zero"
   | 2 -> "one"
   | _ -> failwith "rotate3"
```

In this example, the constant patterns are 0, 1, and 2. Any constant listed in §6.3.1 may be used as a constant pattern except for integer literals that have the suffixes `Q`, `R`, `Z`, `I`, `N`, `G`.

Simple constant patterns have the corresponding simple type. Such patterns elaborate to a call to the F# structural equality function `FSharp.Core.Operators.(=)` with the pattern input and the constant as arguments. The match succeeds if this call returns `true`; otherwise, the match fails.

Note: The use of `FSharp.Core.Operators.(=)` means that CLI floating-point equality is used to match floating-point values, and CLI ordinal string equality is used to match strings.

7.2 Named Patterns

Patterns in the following forms are *named patterns*:

```
Long-ident
Long-ident pat
Long-ident pat-params pat
```

If `Long-ident` is a single identifier that does not begin with an uppercase character, it is interpreted as a *variable pattern*. During checking, the variable is assigned the same value and type as the pattern input.

If `Long-ident` is more than one-character long or begins with an uppercase character (that is, if `System.Char.IsUpperInvariant` is `true` and `System.Char.IsLowerInvariant` is `false` on the first character), it is resolved by using *Name Resolution in Patterns* (§14.1.6). This algorithm produces one of the following:

- A union case
- An exception label
- An active pattern case name
- A literal value

Otherwise, `Long-ident` must be a single uppercase identifier `ident`. In this case, `pat` is a variable pattern. An F# implementation may optionally generate a warning if the identifier is uppercase. Such a warning is recommended if the length of the identifier is greater than two.

After name resolution, the subsequent treatment of the named pattern is described in the following sections.

7.2.1 Union Case Patterns

If `Long-ident` from §7.2 resolves to a union case, the pattern is a union case pattern. If `Long-ident` resolves to a union case `Case`, then `Long-ident` and `Long-ident pat` are patterns that match pattern inputs that have union case label `Case`. The `Long-ident` form is used if the corresponding case takes no arguments, and the `Long-ident pat` form is used if it takes arguments.

At runtime, if the pattern input is an object that has the corresponding union case label, the data values carried by the union are matched against the given argument patterns.

For example:

```
type Data =
    | Kind1 of int * int
```

```
        | Kind2 of string * string

let data = Kind1(3, 2)

let result =
    match data with
    | Kind1 (a, b) -> a + b
    | Kind2 (s1, s2) -> s1.Length + s2.Length
```

In this case, result is given the value 5.

When a union case has named fields, these names may be referenced in a union case pattem. When using pattern matching with multiple fields, semicolons are used to delimit the named fields. For example

```
type Shape =
    | Rectangle of width: float * height: float
    | Square of width: float

let getArea (s: Shape) =
    match s with
    | Rectangle (width = w; height = h) -> w*h
    | Square (width = w) -> w*w
```

7.2.2 Literal Patterns

If *long-ident* from §7.2 resolves to a literal value, the pattern is a literal pattern. The pattern is equivalent to the corresponding constant pattern.

In the following example, the `Literal` attribute (§10.2.2) is first used to define two literals, and these literals are used as identifiers in the match expression:

```
[<Literal>]
let Case1 = 1

[<Literal>]
let Case2 = 100

let result =
    match 100 with
    | Case1 -> "Case1"
    | Case2 -> "Case2"
    | _ -> "Some other case"
```

In this case, `result` is given the value "Case2”.

7.2.3 Active Patterns

If *Long-ident* from §7.2 resolves to an *active pattern case name* $CaseName_i$ then the pattern is an active pattern. The rules for name resolution in patterns (§14.1.6) ensure that $CaseName_i$ is associated with an *active pattern function* *f* in one of the following forms:

- `(|CaseName|) inp`

  Single case. The function accepts one argument (the value being matched) and can return any type.

- `(|CaseName|_|) inp`

  Partial. The function accepts one argument (the value being matched) and must return a value of type `FSharp.Core.option<_>`

- `(|`$CaseName_1$`| ...|`$CaseName_n$`|) inp`

  Multi-case. The function accepts one argument (the value being matched), and must return a value of type `FSharp.Core.Choice<_,...,_>` based on the number of case names. In F#, the limitation $n \leq 7$ applies.

- `(|CaseName|)` arg_1 `...` arg_n `inp`

  Single case with parameters. The function accepts $n+1$ arguments, where the last argument (*inp*) is the value to match, and can return any type.

- `(|CaseName|_|)` arg_1 `...` arg_n `inp`

  Partial with parameters. The function accepts $n+1$ arguments, where the last argument (*inp*) is the value to match, and must return a value of type `FSharp.Core.option<_>.`

Other active pattern functions are not permitted. In particular, multi-case, partial functions such as the following are not permitted:

```
(|CaseName1| ... |CaseNamen|_|)
```

When an active pattern function takes arguments, the *pat-params* are interpreted as expressions that are passed as arguments to the active pattern function. The *pat-params* are converted to the syntactically identical corresponding expression forms and are passed as arguments to the active pattern function *f*.

At runtime, the function *f* is applied to the pattern input, along with any parameters. The pattern matches if the active pattern function returns *v*, `ChoicekOfN` *v*, or `Some` *v*, respectively, when applied to the pattern input. If the pattern argument *pat* is present, it is then matched against *v*.

The following example shows how to define and use a partial active pattern function:

```
let (|Positive|_|) inp = if inp > 0 then Some(inp) else None
let (|Negative|_|) inp = if inp < 0 then Some(-inp) else None

match 3 with
| Positive n -> printfn "positive, n = %d" n
| Negative n -> printfn "negative, n = %d" n
| _          -> printfn "zero"
```

The following example shows how to define and use a multi-case active pattern function:

```
let (|A|B|C|) inp = if inp < 0 then A elif inp = 0 then B else C

match 3 with
| A -> "negative"
| B -> "zero"
| C -> "positive"
```

The following example shows how to define and use a parameterized active pattern function:

```
let (|MultipleOf|_|) n inp = if inp%n = 0 then Some (inp / n) else None

match 16 with
| MultipleOf 4 n -> printfn "x = 4*%d" n
| _ -> printfn "not a multiple of 4"
```

An active pattern function is executed only if a left-to-right, top-to-bottom reading of the entire pattern indicates that execution is required. For example, consider the following active patterns:

```
let (|A|_|) x =
    if x = 2 then failwith "x is two"
    elif x = 1 then Some()
    else None

let (|B|_|) x =
    if x=3 then failwith "x is three" else None

let (|C|) x = failwith "got to C"

let f x =
    match x with
    | 0 -> 0
    | A -> 1
    | B -> 2
    | C -> 3
    | _ -> 4
```

These patterns evaluate as follows:

```
f 0 // 0
f 1 // 1
f 2 // failwith "x is two"
f 3 // failwith "x is three"
f 4 // failwith "got to C"
```

An active pattern function may be executed multiple times against the same pattern input during resolution of a single overall pattern match. The precise number of times that the active pattern function is executed against a particular pattern input is implementation-dependent.

7.3 “As” Patterns

An “as” pattern is of the following form:

```
pat as ident
```

The “as” pattern defines *ident* to be equal to the pattern input and matches the pattern input against *pat*. For example:

```
let t1 = (1, 2)
let (x, y) as t2 = t1
printfn "%d-%d-%A" x y t2  // 1-2-(1, 2)
```

This example binds the identifiers `x`, `y`, and `t1` to the values `1`, `2`, and `(1,2)`, respectively.

7.4 Wildcard Patterns

The pattern _ is a wildcard pattern and matches any input. For example:

```
let categorize x =
    match x with
    | 1 -> 0
    | 0 -> 1
    | _ -> 0
```

In the example, if `x` is 0, the match returns 1. If `x` has any other value, the match returns 0.

7.5 Disjunctive Patterns

A disjunctive pattern matches an input value against one or the other of two patterns:

```
pat | pat
```

At runtime, the patterm input is matched against the first pattern. If that fails, the pattern input is matched against the second pattern. Both patterns must bind the same set of variables with the same types. For example:

```
type Date = Date of int * int * int

let isYearLimit date =
    match date with
    | (Date (year, 1, 1) | Date (year, 12, 31)) -> Some year
    | _ -> None

let result = isYearLimit (Date (2010,12,31))
```

In this example, `result` is given the value `true`, because the pattern input matches the second pattern.

7.6 Conjunctive Patterns

A conjunctive pattern matches the pattern input against two patterns.

pat_1 `&` pat_2

For example:

```
let (|MultipleOf|_|) n inp = if inp%n = 0 then Some (inp / n) else None

let result =
    match 56 with
    | MultipleOf 4 m & MultipleOf 7 n -> m + n
    | _ -> false
```

In this example, `result` is given the value `22` (= `16` + `8`), because the pattern input match matches both patterns.

7.7 List Patterns

The pattern *pat* `::` *pat* is a union case pattern that matches the "cons" union case of F# list values.

The pattern `[]` is a union case pattern that matches the "nil" union case of F# list values.

The pattern `[`pat_1 `; ... ;` pat_n`]` is shorthand for a series of `::` and empty list patterns pat_1 `::` … `::` pat_n `::` `[]`.

For example:

```
let rec count x =
    match x with
    | [] -> 0
    | h :: t -> h + count t

let result1 = count [1;2;3]
let result2 =
    match [1;2;3] with
    | [a;b;c] -> a + b + c
    | _ -> 0
```

In this example, both `result1` and `result2` are given the value `6`.

7.8 Type-Annotated Patterns

A *type-annotated pattern* specifies the type of the value to match to a pattern.

pat `:` *type*

For example:

```
let rec sum xs =
    match xs with
    | [] -> 0
    | (h : int) :: t -> h + sum t
```

In this example, the initial type of `h` is asserted to be equal to `int` before the pattern `h` is checked. Through type inference, this in turn implies that `xs` and `t` have static type `int list`, and `sum` has static type
`int list -> int`.

7.9 Dynamic Type-Test Patterns

Dynamic type-test patterns have the following two forms:

```
:? type
:? type as ident
```

A dynamic type-test pattern matches any value whose runtime type is *type* or a subtype of *type*. For example:

```
let message (x : System.Exception) =
    match x with
    | :? System.OperationCanceledException -> "cancelled"
    | :? System.ArgumentException -> "invalid argument"
    | _ -> "unknown error"
```

If the type-test pattern is of the form `:? type as ident`, then the value is coerced to the given type and *`ident`* is bound to the result. For example:

```
let findLength (x : obj) =
    match x with
    | :? string as s -> s.Length
    | _ -> 0
```

In the example, the identifier `s` is bound to the value `x` with type `string`.

If the pattern input has type ty_{in}, pattern checking uses the same conditions as both a dynamic type-test expression `e :? type` and a dynamic coercion expression `e :?> type` where *e* has type ty_{in}. An error occurs if *type* cannot be statically determined to be a subtype of the type of the pattern input. A warning occurs if the type test will always succeed based on *type* and the static type of the pattern input.

A warning is issued if an expression contains a redundant dynamic type-test pattern, after any coercion is applied. For example:

```
match box "3" with
| :? string -> 1
```

```
| :? string -> 1  // a warning is reported that this rule is "never matched"
| _ -> 2

match box "3" with
| :? System.IComparable -> 1
| :? string -> 1  // a warning is reported that this rule is "never matched"
| _ -> 2
```

At runtime, a dynamic type-test pattern succeeds if and only if the corresponding dynamic type-test expression *e* `:?` *ty* would return `true` where *e* is the pattern input. The value of the pattern is bound to the results of a dynamic coercion expression *e* `:?>` *ty*.

7.10 Record Patterns

The following is a *record pattern*:

$$\{\ long\text{-}ident_1 = pat_1;\ \ldots\ ;\ long\text{-}ident_n = pat_n\}$$

For example:

```
type Data = { Header:string; Size: int; Names: string list }

let totalSize data =
    match data with
    | { Header = "TCP"; Size = size; Names = names } -> size + names.Length * 12
    | { Header = "UDP"; Size = size } -> size
    | _ -> failwith "unknown header"
```

The $long\text{-}ident_i$ are resolved in the same way as field labels for record expressions and must together identify a single, unique F# record type. Not all record fields for the type need to be specified in the pattern.

7.11 Array Patterns

An *array pattern* matches an array of a partciular length:

```
[|pat ; ... ; pat|]
```

For example:

```
let checkPackets data =
    match data with
    | [| "HeaderA"; data1; data2 |] -> (data1, data2)
    | [| "HeaderB"; data2; data1 |] -> (data1, data2)
    | _ -> failwith "unknown packet"
```

7.12 Null Patterns

The *null pattern* null matches values that are represented by the CLI value null. For example:

```
let path =
    match System.Environment.GetEnvironmentVariable("PATH") with
    | null -> failwith "no path set!"
    | res -> res
```

Most F# types do not use null as a representation; consequently, the null pattern is generally used to check values passed in by CLI method calls and properties. For a list of F# types that use null as a representation, see §5.4.8.

7.13 Guarded Pattern Rules

Guarded pattern rules have the following form:

```
pat when expr
```

For example:

```
let categorize x =
    match x with
    | _ when x < 0 -> -1
    | _ when x < 0 -> 1
    | _ -> 0
```

The guards on a rule are executed only after the match value matches the corresponding pattern. For example, the following evaluates to 2 with no output.

```
match (1, 2) with
| (3, x) when (printfn "not printed"; true) -> 0
| (_, y) -> y
```

8. Type Definitions

Type definitions define new named types. The grammar of type definitions is shown below.

```
type-defn :=
    abbrev-type-defn
    record-type-defn
    union-type-defn
    anon-type-defn
    class-type-defn
    struct-type-defn
    interface-type-defn
    enum-type-defn
    delegate-type-defn
    type-extension

type-name :=
    attributes_opt access_opt ident typar-defns_opt

abbrev-type-defn :=
    type-name = type

union-type-defn :=
    type-name '=' union-type-cases type-extension-elements_opt

union-type-cases   :=
    '|'_opt union-type-case '|' ... '|' union-type-case

union-type-case :=
    attributes_opt union-type-case-data

union-type-case-data :=
    ident                      -- null union case
    ident of union-type-field * ... * union-type-field -- n-ary union
case
    ident : uncurried-sig   -- n-ary union case

union-type-field :=
    type                       -- unnamed union fiels
    ident : type                     -- named union field

record-type-defn :=
    type-name = '{' record-fields '}' type-extension-elements_opt

record-fields :=
    record-field ; ... ; record-field ;_opt

record-field :=
    attributes_opt mutable_opt access_opt ident : type

anon-type-defn :=
```

```
    type-name primary-constr-args_opt object-val_opt '=' begin class-
type-body end

class-type-defn :=
    type-name primary-constr-args_opt object-val_opt '=' class class-
type-body end

as-defn := as ident

class-type-body :=
    class-inherits-decl_opt class-function-or-value-defns_opt type-defn-
elements_opt

class-inherits-decl := inherit type expr_opt

class-function-or-value-defn :=
    attributes_opt static_opt let rec_opt function-or-value-defns
    attributes_opt static_opt do expr

struct-type-defn :=
    type-name primary-constr-args_opt as-defn_opt '=' struct struct-type-
body end

struct-type-body := type-defn-elements

interface-type-defn :=
    type-name '=' interface interface-type-body end

interface-type-body := type-defn-elements

exception-defn :=
    attributes_opt exception union-type-case-data    -- exception
definition
    attributes_opt exception ident = long-ident      -- exception
abbreviation

enum-type-defn :=
    type-name '=' enum-type-cases

enum-type-cases =
    '|'_opt enum-type-case '|' ... '|' enum-type-case

enum-type-case :=
    ident '=' const -- enum constant definition

delegate-type-defn :=
    type-name '=' delegate-sig

delegate-sig :=
    delegate of uncurried-sig      -- CLI delegate definition

type-extension :=
    type-name type-extension-elements
```

```
type-extension-elements := with type-defn-elements end

type-defn-element :=
    member-defn
    interface-impl
    interface-spec

type-defn-elements :=  type-defn-element ... type-defn-element

primary-constr-args :=
    attributesopt accessopt (simple-pat, ... , simple-pat)

simple-pat :=
    | ident
    | simple-pat : type


additional-constr-defn :=
    attributesopt accessopt new pat as-defn = additional-constr-expr

additional-constr-expr :=
    stmt ';' additional-constr-expr        -- sequence construction
(after)
    additional-constr-expr then expr       -- sequence construction
(before)
    if expr then additional-constr-expr else  additional-constr-expr
    let function-or-value-defn in  additional-constr-expr
    additional-constr-init-expr

additional-constr-init-expr :=
    '{' class-inherits-decl field-initializers '}'-- explicit
construction
    new type expr                          -- delegated construction

member-defn :=
    attributesopt staticopt member accessopt method-or-prop-defn      --
concrete member
    attributesopt abstract memberopt accessopt member-sig -- abstract
member
    attributesopt override accessopt method-or-prop-defn      --
override member
    attributesopt default accessopt method-or-prop-defn       --
override member
    attributesopt staticopt val mutableopt accessopt ident : type      --
value member
    additional-constr-defn -- additional constructor

method-or-prop-defn :=
    ident.opt function-defn         -- method definition
    ident.opt value-defn                   -- property definition
    ident.opt ident with function-or-value-defns    -- property
definition via get/set methods
```

```
    member ident = exp                     -- auto-implemented
property definition
    member ident = exp with get            -- auto-implemented
property definition
    member ident = exp with set            -- auto-implemented
property definition
    member ident = exp with get,set        -- auto-implemented
property definition
    member ident = exp with set,get        -- auto-implemented
property definition

member-sig :=
    ident typar-defnsopt : curried-sig               -- method or
property signature
    ident typar-defnsopt : curried-sig with get     -- property
signature
    ident typar-defnsopt : curried-sig with set     -- property
signature
    ident typar-defnsopt : curried-sig with get,set-- property
signature
    ident typar-defnsopt : curried-sig with set,get-- property
signature

curried-sig :=
    args-spec -> ... -> args-spec -> type

uncurried-sig :=
    args-spec -> type

args-spec :=
    arg-spec * ... * arg-spec

arg-spec :=
    attributesopt arg-name-specopt type

arg-name-spec :=
    ?opt ident :

interface-spec :=
    interface type
```

For example:

```
type int = System.Int32
type Color = Red | Green | Blue
type Map<'T> = { entries: 'T[] }
```

Type definitions can be declared in:

- Module definitions
- Namespace declaration groups

F# supports the following kinds of type definitions:

- Type abbreviations (§8.3)
- Record type definitions (§8.4)
- Union type definitions (§8.5)
- Class type definitions (§8.6)
- Interface type definitions (§8.7)
- Struct type definitions (§8.8)
- Enum type definitions (§8.9)
- Delegate type definitions (§8.10)
- Exception type definitions (§8.11)
- Type extension definitions (§8.12)
- Measure type definitions (§9.4)

With the exception of type abbreviations and type extension definitions, type definitions define fresh, named types that are distinct from other types.

A *type definition group* defines several type definitions or extensions simultaneously:

```
type ... and ...
```

For example:

```
type RowVector(entries: seq<int>) =
    let entries = Seq.toArray entries
    member x.Length = entries.Length
    member x.Permute = ColumnVector(entries)

and ColumnVector(entries: seq<int>) =
    let entries = Seq.toArray entries
    member x.Length = entries.Length
    member x.Permute = RowVector(entries)
```

A type definition group can include any type definitions except for exception type definitions and module definitions.

Most forms of type definitions may contain both *static* elements and *instance* elements. Static elements are accessed by using the type definition. Within a `static` definition, only the `static` elements are in scope. Most forms of type definitions may contain *members* (§8.13).

Custom attributes may be placed immediately before a type definition group, in which case they apply to the first type definition, or immediately before the name of the type definition:

```
[<Obsolete>] type X1() = class end

type [<Obsolete>] X2() = class end
and [<Obsolete>] Y2() = class end
```

8.1 Type Definition Group Checking and Elaboration

F# checks type definition groups by determining the basic shape of the definitions and then filling in the details. In overview, a type definition group is checked as follows:

1. For each type definition:
   - Determine the generic arguments, accessibility and kind of the type definition
   - Determine whether the type definition supports equality and/or comparison
   - Elaborate the explicit constraints for the generic parameters.
2. For each type definition:
   - Establish type abbreviations
   - Determine the base types and implemented interfaces of each new type definition
   - Detect any cyclic abbreviations
   - Verify the consistency of types in fields, union cases, and base types.
3. For each type definition:
   - Determine the union cases, fields, and abstract members (§8.14) of each new type definition.
   - Check the union cases, fields, and abstract members themselves, as described in the corresponding sections of this chapter.
4. For each member, add items that represent the members to the environment as a recursive group.
5. Check the members, function, and value definitions in order and apply incremental generalization.

In the context in which type definitions are checked, the type definition itself is in scope, as are all members and other accessible functionality of the type. This context enables recursive references to the accessible static content of a type. It also enables recursive references to the accessible properties of any object that has the same type as the type definition or a related type.

In more detail, given an initial environment *env*, a type definition group is checked as described in the following paragraphs.

First, check the individual type definitions. For each type definition:

1. Determine the number, names, and sorts of generic arguments of the type definition.
   - For each generic argument, if a `Measure` attribute is present, mark the generic argument as a measure parameter. The generic arguments are initially inference parameters, and additional constraints may be inferred for these parameters.
   - For each type definition *T*, the subsequent steps use an environment env_T that is produced by adding the type definitions themselves and the generic arguments for *T* to *env*.

2. Determine the accessibility of the type definition.

3. Determine and check the basic kind of the type definition, using *Type Kind Inference* if necessary (§8.2).

4. Mark the type definition as a measure type definition if a `Measure` attribute is present.

5. If the type definition is generic, infer whether the type definition supports equality and/or comparison.

6. Elaborate and add the explicit constraints for the generic parameters of the type definition, and then generalize the generic parameters. Inference of additional constraints is not permitted.

7. If the type definition is a type abbreviation, elaborate and establish the type being abbreviated.

8. Check and elaborate any base types and implemented interfaces.

9. If the type definition is a type abbreviation, check that the type abbreviation is not cyclic.

10. Check whether the type definition has a single, zero-argument constructor, and hence forms a type that satisfies the default constructor constraint.

11. Recheck the following to ensure that constraints are consist:

    - The type being abbreviated, if any.
    - The explicit constraints for any generic parameters, if any.
    - The types and constraints occurring in the base types and implemented interfaces, if any.

12. Determine the union cases, fields, and abstract members, if any, of the type definition. Check and elaborate the types that the union cases, fields, and abstract members include.

13. Make additional checks as defined elsewhere in this chapter. For example, check that the `AbstractClass` attribute does not appear on a union type.

14. For each type definition that is a struct, class, or interface, check that the inheritance graph and the struct-inclusion graph are not cyclic. This check ensures that a struct does not contain itself and that a class or interface does not inherit from itself. This check includes the following steps:

    a) Create a graph with one node for each type definition.

    b) Close the graph under edges.

        - (T, base-type-definition)
        - (T, interface-type-definition)
        - (T_1, T_2) where T_1 is a struct and T_2 is a type that would store a value of type T_1 <...> for some instantiation. Here “X storing Y” means that X is Y or is a struct type with an instance field that stores Y.

    c) Check for cycles.

The special case of a struct `S<typars>` storing a static field of type `S<typars>` is allowed.

15. Collectively add the elaborated member items that represent the members for all new type definitions to the environment as a recursive group (§8.13), excluding interface implementation members.

16. If the type definition has a primary constructor, create a member item to represent the primary constructor.

After these steps are complete for each type definition, check the members. For each member:

1. If the member is in a generic type, create a copy of the type parameters for the generic type and add the copy to the environment for that member.

2. If the member has explicit type parameters, elaborate these type parameters and any explicit constraints.

3. If the member is an override, default, or interface implementation member, apply dispatch-slot inference.

4. If the member has syntactic parameters, assign an initial type to the elaborated member item based on the patterns that specify arguments for the members.

5. If the member is an instance member, assign a type to the instance variable.

Finally, check the function, value, and member definitions of each new type definition in order as a recursive group.

8.2 Type Kind Inference

A type that is specified in one of the following ways has an anonymous type kind:

- By using `begin` and `end` on the right-hand side of the `=` token.
- In lightweight syntax, with an implicit `begin`/`end`.

F# infers the kind of an anonymous type by applying the following rules, in order:

1. If the type has a `Class` attribute, `Interface` attribute, or `Struct` attribute, this attribute identifies the kind of the type.

2. If the type has any concrete elements, the type is a class. Concrete elements are primary constructors, additional object constructors, function definitions, value definitions, non-abstract members, and any `inherit` declarations that have arguments.

3. Otherwise, the type is an interface type.

For example:

```
// This is implicitly an interface
```

```
type IName =
    abstract Name : string

// This is implicitly a class, because it has a constructor
type ConstantName(n:string) =
    member x.Name = n

// This is implicitly a class, because it has a constructor
type AbstractName(n:string) =
    abstract Name : string
    default x.Name = "<no-name>"
```

If a type is not an anonymous type, any use of the `Class` attribute, `Interface` attribute, or `Struct` attribute must match the `class`/`end`, `interface`/`end`, and `struct`/`end` tokens, if such tokens are present. These attributes cannot be used with other kinds of type definitions such as type abbreviations, record, union, or enum types.

8.3 Type Abbreviations

Type abbreviations define new names for other types. For example:

```
type PairOfInt = int * int
```

Type abbreviations are expanded and erased during compilation and do not appear in the elaborated form of F# declarations, nor can they be referred to or accessed at runtime.

The process of repeatedly eliminating type abbreviations in favor of their equivalent types must not result in an infinite type derivation. For example, the following are not valid type definitions:

```
type X = option<X>

type Identity<'T> = 'T
and Y = Identity<Y>
```

The constraints on a type abbreviation must satisfy any constraints that the abbreviated type requires.

For example, assuming the following declarations:

```
type IA =
    abstract AbstractMember : int -> int

type IB =
    abstract AbstractMember : int -> int

type C<'T when 'T :> IB>() =
    static member StaticMember(x : 'a) = x.AbstractMember(1)
```

the following is permitted:

```
type D<'T when 'T :> IB> = C<'T>
```

whereas the following is not permitted:

```
type E<'T> = C<'T>  // invalid: missing constraint
```

Type abbreviations can define additional constraints, so the following is permitted:

```
type F<'T when 'T :> IA and 'T :> IB> = C<'T>
```

The right side of a type abbreviation must use all the declared type variables that appear on the left side. For this purpose, the order of type variables that are used on the right-hand side of a type definition is determined by their left-to-right occurrence in the type.

For example, the following is not a valid type abbreviation.

```
type Drop<'T,'U> = 'T * 'T // invalid: dropped type variable
```

> **Note**: This restriction simplifies the process of guaranteeing a stable and consistent compilation to generic CLI code.

Flexible type constraints *#type* may not be used on the right side of a type abbreviation, because they expand to a type variable that has not been named in the type arguments of the type abbreviation. For example, the following type is disallowed:

```
type BadType = #Exception -> int  // disallowed
```

Type abbreviations may be declared `internal` or `private`.

> Note: Private type abbreviations are still, for all purposes, considered equivalent to the abbreviated types.

8.4 Record Type Definitions

A *record type definition* introduces a type in which all the inputs that are used to construct a value are accessible as properties on values of the type. For example:

```
type R1 =
    { x : int;
      y : int }
    member this.Sum = this.x + this.y
```

In this example, the integers x and y can be accessed as properties on values of type R1.

Record fields may be marked mutable. For example:

```
type R2 =
    { mutable x : int;
      mutable y : int }
    member this.Move(dx,dy) =
        this.x <- this.x + dx
```

```
            this.y <- this.y + dy
```

The `mutable` attribute on `x` and `y` makes the assignments valid.

Record types are implicitly sealed and may not be given the `Sealed` attribute. Record types may not be given the `AbstractClass` attribute.

Record types are implicitly marked serializable unless the `AutoSerializable(false)` attribute is used.

8.4.1 Members in Record Types

Record types may declare members (§8.13), overrides, and interface implementations. Like all types with overrides and interface implementations, they are subject to *Dispatch Slot Checking* (§14.8).

8.4.2 Name Resolution and Record Field Labels

For a record type, the record field labels $\texttt{field}_1$ ... $\texttt{field}_N$ are added to the *FieldLabels* table of the current name resolution environmentunless the record type has the `RequireQualifiedAccess` attribute.

Record field labels in the *FieldLabels* table play a special role in *Name Resolution for Members* (§14.1): an expression's type may be inferred from a record label. For example:

```
type R = { dx : int; dy: int }
let f x = x.dx // x is inferred to have type R
```

In this example, the lookup `.dx` is resolved to be a field lookup.

8.4.3 Structural Hashing, Equality, and Comparison for Record Types

Record types implicitly implement the following interfaces and dispatch slots unless they are explicitly implemented as part of the definition of the record type:

```
interface System.Collections.IStructuralEquatable
interface System.Collections.IStructuralComparable
interface System.IComparable
override GetHashCode : unit -> int
override Equals : obj -> bool
```

The implicit implementations of these interfaces and overrides are described in §8.15.

8.4.4 With/End in Record Type Definitions

Record type definitions can include `with/end` tokens, as the following shows:

```
type R1 =
    { x : int;
      y : int }
    with
        member this.Sum = this.x + this.y
    end
```

The with/end tokens can be omitted if the type-defn-elements vertically align with the { in the record-fields. The semicolon (;) tokens can be omitted if the next *record-field* vertically aligns with the previous *record-field*.

8.4.5 CLIMutable Attributes

Adding the CLIMutable attribute to a record type causes it to be compiled to a CLI representation as a plain-old CLR object (POCO) with a default constructor along with property getters and setters. Adding the default constructor and mutable properties makes objects of the record type usable with .NET tools and frameworks such as database queries, serialization frameworks, and data models in XAML programming.

For example, an F# immutable record cannot be serialized because it does not have a constructor. However, if you attach the CLIMutable attribute as in the following example, the XmlSerializer is enable to serialize or deserialize this record type:

```
[<CLIMutable>]
type R1 = { x : string;  y : int }
```

8.5 Union Type Definitions

A *union type definition* is a type definition that includes one or more *union cases*. For example:

```
type Message =
    | Result of string
    | Request of int * string
    member x.Name = match x with Result(nm) -> nm | Request(_,nm) -> nm
```

Union case names must begin with an uppercase letter, which is defined to mean any character for which the CLI library function System.Char.IsUpper returns true and System.Char.IsLower returns false.

The union cases Case1 ... CaseN have module scope and are added to the *ExprItems* and *PatItems* tables in the name resolution environment. This means that their unqualified names can be used to form both expressions and patterns, unless the record type has the RequireQualifiedAccess attribute.

Parentheses are significant in union definitions. Thus, the following two definitions differ:

```
type CType = C of int * int
type CType = C of (int * int)
```

The lack of parentheses in the first example indicates that the union case takes two arguments. The parentheses in the second example indicate that the union case takes one argument that is a first-class tuple value.

Union fields may optionally be named within each case of a union type. For example:

```
type Shape =
    | Rectangle of width: float * length: float
    | Circle of radius: float
    | Prism of width: float * float * height: float
```

The names are referenced when pattern matching on union values of this type. When using pattern matching with multiple fields, semicolons are used to delimit the named fields, e.g. `Prism(width=w; height=h)`.

The following declaration defines a type abbreviation if the named type `A` exists in the name resolution environment. Otherwise it defines a union type.

```
type OneChoice = A
```

To disambiguate this case and declare an explicit union type, use the following:

```
type OneChoice =
    | A
```

Union types are implicitly marked serializable unless the `AutoSerializable(false)` attribute is used.

8.5.1 Members in Union Types

Union types may declare members (§8.13), overrides, and interface implementations. As with all types that declare overrides and interface implementations, they are subject to *Dispatch Slot Checking* (§14.8).

8.5.2 Structural Hashing, Equality, and Comparison for Union Types

Union types implicitly implement the following interfaces and dispatch slots unless they are explicitly implemented as part of the definition of the union type:

```
interface System.Collections.IStructuralEquatable
interface System.Collections.IStructuralComparable
interface System.IComparable
override GetHashCode : unit -> int
override Equals : obj -> bool
```

The implicit implementations of these interfaces and overrides are described in §8.15.

8.5.3 With/End in Union Type Definitions

Union type definitions can include `with/end` tokens, as the following shows:

```
type R1 =
    { x : int;
      y : int }
    with
        member this.Sum = this.x + this.y
    end
```

The `with/end` tokens can be omitted if the `type-defn-elements` vertically align with the `{` in the `record-fields`. The semicolon (`;`) tokens can be omitted if the next *`record-field`* vertically aligns with the previous *`record-field`*.

For union types, the `with/end` tokens can be omitted if the `type-defn-elements` vertically alignwith the first `|` in the `union-type-cases`. However, `with/end` must be present if the `|` tokens align with the `type` token. For example:

```
/// Note: this layout is permitted
type Message =
  | Result of string
  | Request of int * string
  member x.Name = match x with Result(nm) -> nm | Request(_,nm) -> nm

/// Note: this layout is not permitted
type Message =
| Result of string
| Request of int * string
member x.Name = match x with Result(nm) -> nm | Request(_,nm) -> nm
```

8.5.4 Compiled Form of Union Types for Use from Other CLI Languages

A compiled union type `U` has:

- One CLI static getter property `U.C` for each null union case `C`. This property gets a singleton object that represents each such case.
- One CLI nested type `U.C` for each non-null union case `C`. This type has instance properties `Item1`, `Item2`.... for each field of the union case, or a single instance property `Item` if there is only one field. However, a compiled union type that has only one case does not have a nested type. Instead, the union type itself plays the role of the case type.
- One CLI static method `U.NewC` for each non-null union case `C`. This method constructs an object for that case.
- One CLI instance property `U.IsC` for each case `C`. This property returns `true` or `false` for the case.
- One CLI instance property `U.Tag` for each case `C`. This property fetches or computes an integer tag corresponding to the case.
- If `U` has more than one case, it has one CLI nested type `U.Tags`. The `U.Tags` typecontains one integer literal for each case, in increasing order starting from zero.
- A compiled union type has the methods that are required to implement its auto-generated interfaces, in addition to any user-defined properties or methods.

These methods and properties may not be used directly from F#. However, these types have user-facing `List.Empty`, `List.Cons`, `Option.None`, and `Option.Some` properties and/or methods.

A compiled union type may not be used as a base type in another CLI language, because it has at least one assembly-private constructor and no public constructors.

8.6 Class Type Definitions

A *class type definition* encapsulates values that are constructed by using one or more object constructors. Class types have the form:

```
type type-name pat_opt as-defn_opt =
    class
        class-inherits-decl_opt
        class-function-or-value-defns_opt
        type-defn-elements
    end
```

The `class/end` tokens can be omitted, in which case Type Kind Inference (§8.2) is used to determine the kind of the type.

In F#, class types are implicitly marked serializable unless the `AutoSerializable(false)` attribute is present.

8.6.1 Primary Constructors in Classes

An *object constructor* represents a way of initializing an object. Object constructors can create values of the type and can partially initialize an object from a subclass. A class can have an optional *primary constructor* and zero or more *additional object constructors*.

If a type definition has a pattern immediately after the *type-name* and any accessibility annotation, then it has a *primary constructor*. For example, the following type has a primary constructor:

```
type Vector2D(dx : float, dy : float) =
    let length = sqrt(dx*x + dy*dy)
    member v.Length = length
    member v.DX = dx
    member v.DY = dy
```

Class definitions that have a primary constructor may contain function and value definitions, including those that use `let rec`.

The pattern for a primary constructor must have zero or more patterns of the following form:

```
(simple-pat, ..., simple-pat)
```

Each *simple-pat* has this form:

```
simple-pat :=
    | ident
    | simple-pat : type
```

Specifically, nested patterns may not be used in the primary constructor arguments. For example, the following is not permitted because the primary constructor arguments contain a nested tuple pattern:

```
type TwoVectors((px, py), (qx, qy)) =
    member v.Length = sqrt((qx-px)*(qx-px) + (qy-py)*(qy-py))
```

Instead, one or more value definitions should be used to accomplish the same effect:

```
type TwoVectors(pv, qv) =
    let (px, py) = pv
    let (qx, qy) = qv
    member v.Length = sqrt((qx-px)*(qx-px) + (qy-py)*(qy-py))
```

When a primary constructor is evaluated, the inheritance and function and value definitions are evaluated in order.

8.6.1.1 Object References in Primary Constructors

For types that have a primary constructor, the name of the object parameter can be bound and used in the non-static function, value, and member definitions of the type definition as follows:

```
type X(a:int) as x =
    let mutable currentA = a
    let mutable currentB = 0
    do x.B <- x.A + 3
    member self.GetResult()=  currentA + currentB
    member self.A with get() = currentA and set v = currentA <- v
    member self.B with get() = currentB and set v = currentB <- v
```

During construction, no member on the type may be called before the last value or function definition in the type has completed; such a call results in an `InvalidOperationException`. For example, the following code raises this exception:

```
type C() as self =
    let f = (fun (x:C) -> x.F())
    let y = f self
    do printfn "construct"
    member this.F() = printfn "hi, y = %A" y

let r = new C() // raises InvalidOperationException
```

The exception is raised because an attempt may be made to access the value of the field `y` before initialization is complete.

8.6.1.2 Inheritance Declarations in Primary Constructors

An `inherit` declaration specifies that the type being defined is an extension of an existing type. Such declarations have the following form:

```
class-inherits-decl := inherit type expr_opt
```

For example:

```
type MyDerived(...) =
   inherit MyBase(...)
```

If a class definition does not contain an `inherit` declaration, the class inherits from`System.Object` by default.

The `inherit` declaration for a type must have arguments if and only if the type has a primary constructor.

Unlike §8.6.1.2, members of a base type can be accessed during construction of the derived class. For example, the following code does not raise an exception:

```
type B() =
    member this.G() = printfn "hello "

type C() as self =
    inherit B()
    let f = (fun (x:C) -> x.G())
    let y = f self
    do printfn "construct"
    member this.F() = printfn "hi, y = %A" y

let r = new C() // does not raise InvalidOperationException
```

8.6.1.3 Instance Function and Value Definitions in Primary Constructors

Classes that have primary constructors may include function definitions, value definitions, and "do" statements. The following rules apply to these definitions:

- Each definition may be marked `static` (see §8.6.2.1). If the definition is not marked `static`, it is called an instance definition.
- The functions and values defined by instance definitions are lexically scoped (and thus implicitly private) to the object being defined.
- Each value definition may optionally be marked `mutable`.
- A group of function and value definitions may optionally be marked `rec`.
- Function and value definitions are generalized.
- Value definitions that declared in classes are represented in compiled code as follows:
    - If a value definition is not mutable, and is not used in any function or member, then the value is represented as a local value in the object constructor.
    - If a value definition is mutable, or used in any function or member, then the value is represented as an instance field in the corresponding CLI type.
- Function definitions are represented in compiled code as private members of the corresponding CLI type.

  For example, consider this type:

  ```
  type C(x:int,y:int) =
      let z = x + y
      let f w = x + w
      member this.Z = z
      member this.Add(w) = f w
  ```

The input `y` is used only during construction, and no field is stored for it. Likewise the function `f` is represented as a member rather than a field that is a function value.

A value definition is considered a function definition if its immediate right-hand-side is an anonymous function, as in this example:

```
let f = (fun w -> x + w)
```

Function and value definitions may have attributes as follows:

- Value definitions represented as fields may have attributes that target fields.
- Value definitions represented as locals may have attributes that target fields, but these attributes will not be attached to any construct in the resulting CLI assembly.
- Function definitions represented as methods may have attributes that target methods.

For example:

```
type C(x:int) =
    [<System.Obsolete>]
    let unused = x
    member __.P = 1
```

In this example, no field is generated for `unused`, and no corresponding compiled CLI attribute is generated.

8.6.1.4 Static Function and Value Definitions in Primary Constructors

Classes that have primary constructors may have function definitions, value definitions, and "do" statements that are marked as static:

- The values that are defined by static function and value definitions are lexically scoped (and thus implicitly private) to the type being defined.
- Each value definition may optionally be marked `mutable`.
- A group of function and value definitions may optionally be marked `rec`.
- Static function and value definitions are generalized.
- Static function and value definitions are computed once per generic instantiation.
- Static function and value definitions are elaborated to a *static initializer* associated with each generic instantiation of the generated class. Static initializers are executed on demand in the same way as static initializers for implementation files §12.5.
- The compiled representation for static value definitions is as follows:
    - If the value is not used in any function or member then the value is represented as a local value in the CLI class initializer of the type.
    - If the value is used in any function or member, then the value is represented as a static field of the CLI class for the type.
- The compiled representation for a static function definition is a private static member of the corresponding CLI type.

Static function and value definitions may have attributes as follows:

- Static function and value definitions represented as fields may have attributes that target fields.
- Static function and value definitions represented as methods may have attributes that target methods.

For example:

```
type C<'T>() =
    static let mutable v = 2 + 2
    static do v <- 3

    member x.P = v
    static member P2 = v+v

printfn "check: %d = 3" (new C<int>()).P
printfn "check: %d = 3" (new C<int>()).P
printfn "check: %d = 3" (new C<string>()).P
printfn "check: %d = 6" (C<int>.P2)
printfn "check: %d = 6" (C<string>.P2)
```

In this example, the value `v` is represented as a static field in the CLI type for `C`. One instance of this field exists for each generic instantiation of `C`. The output of the program is

```
check: 3 = 3
check: 3 = 3
check: 3 = 3
check: 6 = 6
check: 6 = 6
```

8.6.2 Members in Classes

Class types may declare members (§8.13), overrides, and interface implementations. As with all types that have overrides and interface implementations, such class types are subject to *Dispatch Slot Checking* (§14.8).

8.6.3 Additional Object Constructors in Classes

Although the use of primary object constructors is generally preferable, additional object constructors may also be specified. Additional object constructors are required in two situations:

- To define classes that have more than one constructor.
- To specify explicit `val` fields without the `DefaultValue` attribute.

For example, the following statement adds a second constructor to a class that has a primary constructor:

```
type PairOfIntegers(x:int,y:int) =
    new (x) = PairOfIntegers(x,x)
```

The next example declares a class without a primary constructor:

```
type PairOfStrings =
    val s1 : string
    val s2 : string
    new (s) = { s1 = s; s2 = s }
    new (s1,s2) = { s1 = s1; s2 = s2 }
```

If a primary constructor is present, additional object constructors must call another object constructor in the same type, which may be another additional constructor or the primary constructor.

If no primary constructor is present, additional constructors must initialize any `val` fields of the object that do not have the `DefaultValue` attribute. They must also specify a call to a base class constructor for any inherited class type. A call to a base class constructor is not required if the base class is `System.Object`.

The use of additional object constructors and `val` fields is required if a class has multiple object constructors that must each call different base class constructors. For example:

```
type BaseClass =
    val s1 : string
    new (s) = { s1 = s }
    new () = { s1 = "default" }

type SubClass =
    inherit BaseClass
    val s2 : string
    new (s1,s2) = { inherit BaseClass(s1); s2 = s2 }
    new (s2) = { inherit BaseClass(); s2 = s2 }
```

To implement additional object constructors, F# uses a restricted subset of expressions that ensure that the code generated for the constructor is valid according to the rules of object construction for CLI objects. Note that precisely one *`additional-constr-init-expr`* occurs for each branch of a construction expression.

For classes without a primary constructor, side effects can be performed after the initialization of the fields of the object by using the *`additional-constr-expr`* `then` *`stmt`* form. For example:

```
type PairOfIntegers(x:int,y:int) =
    // This additional constructor has a side effect after
initialization.
    new(x) =
       PairOfIntegers(x, x)
       then
          printfn "Initialized with only one integer"
```

The name of the object parameter can be bound within additional constructors. For example:

```
type X =
    val a : (unit -> string)
    val mutable b : string
    new() as x = { a = (fun () -> x.b); b = "b" }
```

A warning is given if `x` occurs syntactically in or before the *`additional-constr-init-expr`* of the construction expression. If any member is called before the completion of execution of the *`additional-constr-init-expr`* within the *additional-constr-expr* then an `InvalidOperationException` is thrown.

8.6.4 Additional Fields in Classes

Additional field declarations indicate that a value is stored in an object. They are generally used only for classes without a primary constructor, or for mutable fields that use default initialization, and typically occur only in generated code. For example:

```
type PairOfIntegers =
    val x : int
    val y : int
    new(x, y) = {x = x; y = y}
```

The following shows an additional field declaration as a static field in an explicit class type:

```
type TypeWithADefaultMutableBooleanField =
    [<DefaultValue>]
    static val mutable ready : bool
```

At runtime, such a field is initially assigned the zero value for its type (§6.9.3). For example:

```
type MyClass(name:string) =
    // Keep a global count. It is initially zero.
    [<DefaultValue>]
    static val mutable count : int

    // Increment the count each time an object is created
    do MyClass.count <- MyClass.count + 1

    static member NumCreatedObjects = MyClass.count

    member x.Name = name
```

A `val` specification in a type that has a primary constructor must be marked mutable and must have the `DefaultValue` attribute. For example:

```
type X() =
    [<DefaultValue>]
    val mutable x : int
```

The `DefaultValue` attribute takes a `check` parameter, which indicates whether to ensure that the `val` specification does not create unexpected null values. The default value for `check` is `true`. If this parameter is `true`, the type of the field must permit default initialization (§5.4.8). For example, the following type is rejected:

```
type MyClass<'T>() =
    [<DefaultValue>]
    static val mutable uninitialized : 'T
```

The reason is that the type `'T` does not admit default initialization. However, in compiler-generated and hand-optimized code it is sometimes essential to be able to emit fields that are completely uninitialized. In this case, `DefaultValue(false)` can be used. For example:

```
type MyNullable<'T>() =
    [<DefaultValue>]
    static val mutable ready : bool

    [<DefaultValue(false)>]
    static val mutable uninitialized : 'T
```

8.7 Interface Type Definitions

An *interface type definition* represents a contract that an object may implement. Such a type definition containsonly abstract members. For example:

```
type IPair<'T,'U> =
    interface
        abstract First: 'T
        abstract Second: 'U
    end

type IThinker<'Thought> =
    abstract Think: ('Thought -> unit) -> unit
    abstract StopThinking: (unit -> unit)
```

Note: The `interface/end` tokens can be omitted when lightweight syntax is used, in which case Type Kind Inference (§8.2) is used to determine the kind of the type. The presence of any non-abstract members or constructors means a type is not an interface type.

By convention, interface type names start with I, as in IEvent. However, this convention is not followed as strictly in F# as in other CLI languages.

Interface types may be arranged hierarchically by specifying `inherit` declarations. For example:

```
type IA =
    abstract One: int -> int

type IB =
    abstract Two: int -> int

type IC =
```

```
        inherit IA
        inherit IB
        abstract Three: int -> int
```

Each `inherit` declaration must itself be an interface type. Circular references are not allowed among `inherit` declarations. F# uses the named types of the inherited interface types to determine whether references are circular.

8.8 Struct Type Definitions

A *struct type definition* is a type definition whose instances are stored inline inside the stack frame or object of which they are a part. The type is represented as a CLI struct type, also called a *value type*. For example:

```
    type Complex =
        struct
            val real: float;
            val imaginary: float
            member x.R = x.real
            member x.I = x.imaginary
        end
```

> Note: The `struct/end` tokens can be omitted when lightweight syntax is used, in which case Type Kind Inference (§8.2) is used to determine the kind of the type.

Becaues structs undergo type kind inference (§8.2), the following is valid:

```
    [<Struct>]
    type Complex(r:float, i:float) =
        member x.R = r
        member x.I = i
```

Structs may have primary constructors:

```
    [<Struct>]
    type Complex(r : float, I : float) =
        member x.R = r
        member x.I = i
```

Structs that have primary constructors must accept at least one argument.

Structs may have additional constructors. For example:

```
    [<Struct>]
    type Complex(r : float, I : float) =
        member x.R = r
        member x.I = i
        new(r : float) = new Complex(r, 0.0)
```

The fields in a struct may be mutable only if the struct does not have a primary constructor. For example:

```
[<Struct>]
type MutableComplex =
    val mutable real : float;
    val mutable imaginary : float
    member x.R = x.real
    member x.I = x.imaginary
    member x.Change(r, i) = x.real <- r; x.imaginary <- i
    new (r, i) = { real = r; imaginary = i }
```

Struct types may declare members, overrides, and interface implementations. As for all types that declare overrides and interface implementations, struct types are subject to *Dispatch Slot Checking* (§14.8).

Structs may not have `inherit` declarations.

Structs may not have “let” or “do” statements unless they are static. For example, the following is not valid:

```
[<Struct>]
type BadStruct1 (def : int) =
    do System.Console.WriteLine("Structs cannot use 'do'!")
```

Structs may have static “let” or “do” statements. For example, the following is valid:

```
[<Struct>]
type GoodStruct1 (def : int) =
    static do System.Console.WriteLine("Structs can use 'static do'")
```

A struct type must be valid according to the CLI rules for structs; in particular, recursively constructed structs are not permitted. For example, the following type definition is not permitted, because the size of `BadStruct2` would be infinite:

```
[<Struct>]
type BadStruct2 =
    val data : float;
    val rest : BadStruct2
    new (data, rest) = { data = data; rest = rest }
```

Likewise, the implied size of the following struct would be infinite:

```
[<Struct>]
type BadStruct3 (data : float, rest : BadStruct3) =
    member s.Data = data
    member s.Rest = rest
```

If the types of all the fields in a struct type permit default initialization, the struct type has an *implicit default constructor*,which initializes all the fields to the default value. For example, the `Complex` type defined earlier in this section permits default initialization.

```
[<Struct>]
type Complex(r : float, I : float) =
    member x.R = r
    member x.I = i
    new(r : float) = new Complex(r, 0.0)

let zero = Complex()
```

Note: The existence of the implicit default constructor for structs is not recorded in CLI metadata and is an artifact of the CLI specification and implementation itself. A CLI implementation permits default constructors for all struct types, although F# does not permit their direct use for F# struct types unless all field types admit default initialization. This is similar to the way that F# considers some types to have null as an abnormal value.

Public struct types for use from other CLI languages should be designed with the existence of the default zero-initializing constructor in mind.

8.9 Enum Type Definitions

Occasionally the need arises to represent a type that compiles as a CLI enumeration type. An *enum type definition* has values that are represented by integer constants and has a CLI enumeration as its compiled form. Enum type definitions are declared by specifying integer constants in a format that is syntactically similar to a union type definition. For example:

```
type Color =
   | Red = 0
   | Green = 1
   | Blue = 2

let rgb = (Color.Red, Color.Green, Color.Blue)

let show(colorScheme) =
     match colorScheme with
     | (Color.Red, Color.Green, Color.Blue) -> printfn "RGB in use"
     | _ -> printfn "Unknown color scheme in use"
```

The example defines the enum type Color, which has the values Red, Green, and Blue, mapped to the constants 0, 1, and 2 respectively. The values are accessed by their qualified names: Color.Red, Color.Green, and Color.Blue.

Each case must be given a constant value of the same type. The constant values dictate the *underlying type* of the enum, and must be one of the following types:

- `sbyte`, `int16`, `int32`, `int64`, `byte`, `uint16`, `uint32`, `uint64`, `char`

The declaration of an enumeration type in an implementation file has the following effects on the typing environment:

- Brings a named type into scope.
- Adds the named type to the inferred signature of the containing namespace or module.

Enum types coerce to `System.Enum` and satisfy the `enum<`*`underlying-type`*`>` constraint for their underlying type.

Each enum type declaration is implicitly annotated with the `RequiresQualifiedAccess` attribute and does not add the tags of the enumeration to the name environment.

```
type Color =
    | Red = 0
    | Green = 1
    | Blue = 2

let red = Red // not accepted, must use Color.Red
```

Unlike unions, enumeration types are fundamentally "incomplete," because CLI enumerations can be converted to and from their underlying primitive type representation. For example, a `Color` value that is not in the above enumeration can be generated by using the `enum` function from the F# library:

```
let unknownColor : Color = enum<Color>(7)
```

This statement adds the value named unknownColor, equal to the constant 7, to the Color enumeration.

8.10 Delegate Type Definitions

Occasionally the need arises to represent a type that compiles as a CLI delegate type. A *delegate type definition* has as its values functions that are represented as CLI delegate values. A delegate type definition is declared by using the `delegate` keyword with a member signature. For example:

```
type Handler<'T> = delegate of obj * 'T -> unit
```

Delegates are often used when using Platform Invoke (P/Invoke) to interface with CLI libraries, as in the following example:

```
type ControlEventHandler = delegate of int -> bool

[<DllImport("kernel32.dll")>]
extern void SetConsoleCtrlHandler(ControlEventHandler callback, bool
add)
```

8.11 Exception Definitions

An *exception definition* defines a new way of constructing values of type `exn` (a type abbreviation for `System.Exception`). Exception definitions have the form:

```
exception ident of type1 * … * typen
```

An exception definition has the following effect:

- The identifier *ident* can be used to generate values of type exn.
- The identifier *ident* can be used to pattern match on values of type exn.
- The definition generates a type with name *ident* that derives from exn.

For example:

```
exception Error of int * string
raise (Error (3, "well that didn't work did it"))

try
    raise (Error (3, "well that didn't work did it"))
with
    | Error(sev, msg) -> printfn "severity = %d, message = %s" sev msg
```

The type that corresponds to the exception definition can be used as a type in F# code. For example:

```
let exn = Error (3, "well that didn't work did it")
let checkException() =
    if (exn :? Error) then printfn "It is of type Error"
    if (exn.GetType() = typeof<Error>) then printfn "Yes, it really is
of type Error"
```

Exception abbreviations may abbreviate existing exception constructors. For example:

```
exception ThatWentBadlyWrong of string * int
exception ThatWentWrongBadly = ThatWentBadlyWrong

let checkForBadDay() =
    if System.DateTime.Today.DayOfWeek = System.DayOfWeek.Monday then
        raise (ThatWentWrongBadly("yes indeed",123))
```

Exception values may also be generated by defining and using classes that extend System.Exception.

8.12 Type Extensions

A *type extension* associates additional members with an existing type. For example, the following associates the additional member IsLong with the existing type System.String:

```
type System.String with
    member x.IsLong = (x.Length > 1000)
```

Type extensions may be applied to any accessible type definition except those defined by type abbreviations. For example, to add an extension method to a list type, use `'a List` because `'a list` is a type abbreviation of `'a List`. For example:

```
type 'a List with
  member x.GetOrDefault(n) =
    if x.Length > n then x.[n]
    else Unchecked.defaultof<'a>

let intlst = [1; 2; 3]
intlst.GetOrDefault(1) //2
intlst.GetOrDefault(4) //0
```

For an array type, backtick marks can be used to define an extension method to the array type:

```
type 'a ``[]`` with
  member x.GetOrDefault(n) =
    if x.Length > n then x.[n]
    else Unchecked.defaultof<'a>

let arrlist = [| 1; 2; 3 |]
arrlist.GetOrDefault(1) //2
arrlist.GetOrDefault(4) //0
```

A type can have any number of extensions.

If the type extension is in the same module or namespace declaration group as the original type definition, it is called an *intrinsic extension*. Members that are defined in intrinsic extensions follow the same name resolution and other language rules as members that are defined as part of the original type definition.

If the type extension is not intrinsic, it must be in a module, and it is called an *extension member*. Opening a module that contains an extension member extends the name resolution of the dot syntax for the extended type. That is, extension members are accessible only if the module that contains the extension is open.

Name resolution for members that are defined in type extensions behaves as follows:

- In method application resolution (see §14.4), regular members (that is, members that are part of the original definition of a type, plus intrinsic extensions) are preferred to extension members.
- Extension members that are in scope and have the correct name are included in the group of members considered for method application resolution (see §14.4).
- An intrinsic member is always preferred to an extension member. If an extension member has the same name and type signature as a member in the original type definition or an inherited member, then it will be inaccessible.

The following illustrates the definition of one intrinsic and one extension member for the same type:

```
namespace Numbers
  type Complex(r : float, i : float) =
```

```
        member x.R = r
        member x.I = i

    // intrinsic extension
    type Complex with
        static member Create(a, b) = new Complex (a, b)
        member x.RealPart = x.R
        member x.ImaginaryPart = x.I

namespace Numbers

    module ComplexExtensions =

        // extension member
        type Numbers.Complex with
            member x.Magnitude = ...
            member x.Phase = ...
```

Extensions may define both instance members and static members.

Extensions are checked as follows:

- Checking applies to the member definitions in an extension together with the members and other definitions in the group of type definitions of which the extension is a part.
- Two intrinsic extensions may not contain conflicting members because intrinsic extensions are considered part of the definition of the type.
- Extensions may not define fields, interfaces, abstract slots, inherit declarations, or dispatch slot (interface and override) implementations.
- Extension members must be in modules.
- Extension members are compiled as CLI static members with encoded names.
  - The elaborated form of an application of a static extension member `C.M(arg1,…,argn)` is a call to this static member with arguments `arg1,…,argn`..
  - The elaborated form of an application of an instance extension member `obj.M(arg1,…,argn)` is an invocation of the static instance member where the object parameter is supplied as the first argument to the extension member followed by arguments `arg1 … argn`.

8.12.1 Imported CLI C# Extensions Members

The CLI C# language defines an "extension member," which commonly occurs in CLI libraries, along with some other CLI languages. C# limits extension members to instance methods.

C#-defined extension members are made available to F# code in environments where the C#-authored assembly is referenced and an `open` declaration of the corresponding namespace is in effect.

The encoding of compiled names for F# extension members is not compatible with C# encodings of C# extension members. However, for instance extension methods, the naming can be made compatible. For example:

```
open System.Runtime.CompilerServices

[<Extension>]
module EnumerableExtensions =
    [<CompiledName("OutputAll"); Extension>]
    type System.Collections.Generic.IEnumerable<'T> with
        member x.OutputAll (this:seq<'T>) =
            for x in this do
                System.Console.WriteLine (box x)
```

C#-style extension members may also be declared directly in F#. When combined with the “inline” feature of F#, this allows the definition of generic, constrained extension members that are not otherwise definable in C# or F#.

```
[<Extension>]
type ExtraCSharpStyleExtensionMethodsInFSharp () =
    [<Extension>]
    static member inline Sum(xs: seq<'T>) = Seq.sum xs
```

Such an extension member can be used as follows:

```
let listOfIntegers = [ 1 .. 100 ]
let listOfBigIntegers = [ 1I .. 100I ]
listOfIntegers.Sum()
listOfBigIntegers.Sum()
```

8.13 Members

Member definitions describe functions that are associated with type definitions and/or values of particular types. Member definitions can be used in type definitions. Members can be classified as follows:

- Property members
- Method members

A *static member* is prefixed by `static` and is associated with the type, rather than with any particular object. Here are some examples of static members:

```
type MyClass() =
    static let mutable adjustableStaticValue = "3"
    static let staticArray = [| "A"; "B" |]
    static let staticArray2 = [|[| "A"; "B" |]; [| "A"; "B" |] |]

    static member StaticMethod(y:int) = 3 + 4 + y
```

```
    static member StaticProperty = 3 + staticArray.Length

    static member StaticProperty2
        with get() = 3 + staticArray.Length

    static member MutableStaticProperty
        with get()         = adjustableStaticValue
        and  set(v:string) = adjustableStaticValue <- v

    static member StaticIndexer
        with get(idx) = staticArray.[idx]

    static member StaticIndexer2
        with get(idx1,idx2) = staticArray2.[idx1].[idx2]

    static member MutableStaticIndexer
        with get (idx1) = staticArray.[idx1]
        and  set (idx1) (v:string) = staticArray.[idx1] <- v
```

An *instance member* is a member without `static`. Here are some examples of instance members:

```
type MyClass() =
    let mutable adjustableInstanceValue = "3"
    let instanceArray = [| "A"; "B" |]
    let instanceArray2 = [| [| "A"; "B" |]; [| "A"; "B" |] |]

    member x.InstanceMethod(y:int) = 3 + y + instanceArray.Length

    member x.InstanceProperty = 3 + instanceArray.Length

    member x.InstanceProperty2
        with get () = 3 + instanceArray.Length

    member x.InstanceIndexer
        with get (idx) = instanceArray.[idx]

    member x.InstanceIndexer2
        with get (idx1,idx2) = instanceArray2.[idx1].[idx2]

    member x.MutableInstanceProperty
        with get ()          = adjustableInstanceValue
        and  set (v:string) = adjustableInstanceValue <- v

    member x.MutableInstanceIndexer
        with get (idx1) = instanceArray.[idx1]
        and  set (idx1) (v:string) = instanceArray.[idx1] <- v
```

Members from a set of mutually recursive type definitions are checked as a single mutually recursive group. As with collections of recursive functions, recursive calls to potentially-generic methods may result in inconsistent type constraints:

```
type Test() =
   static member Id x = x
   member t.M1 (x: int) = Test.Id(x)
   member t.M2 (x: string) = Test.Id(x) // error, x has type 'string'
not 'int'
```

A target method that has a full type annotation is eligible for early generalization (§14.6.7).

```
type Test() =
   static member Id<'T> (x:'T) : 'T = x
   member t.M1 (x: int) = Test.Id(x)
   member t.M2 (x: string) = Test.Id(x)
```

8.13.1 Property Members

A *property member* is a *method-or-prop-defn* in one of the following forms:

```
static_opt member ident._opt ident = expr
static_opt member ident._opt ident with get pat = expr
static_opt member ident._opt ident with set pat_opt pat= expr
static_opt member ident._opt ident with get pat = expr and set pat_opt pat =
expr
static_opt member ident._opt ident with set pat_opt pat = expr and get pat =
expr
```

A property member in the form

```
static_opt member ident._opt ident with get pat1 = expr1 and set pat2a pat2b
opt = expr2
```

is equivalent to two property members of the form:

```
static_opt member ident._opt ident with get pat1 = expr1
static_opt member ident._opt ident with set pat2a pat2b opt = expr2
```

Furthermore, the following two members are equivalent:

```
static_opt member ident._opt ident = expr
static_opt member ident._opt ident with get () = expr
```

These two are also equivalent:

```
static_opt member ident._opt ident with set pat = expr2
static_opt member ident._opt ident with set () pat = expr
```

Thus, property members may be reduced to the following two forms:

```
static_opt member ident._opt ident with get pat_idx = expr
static_opt member ident._opt ident with set pat_idx pat = expr
```

The *ident*.$_{opt}$ must be present if and only if the property member is an instance member. When evaluated, the identifier *ident* is bound to the “this” or “self” object parameter that is associated with the object within the expression *expr*.

A property member is an *indexer property* if *pat*$_{idx}$ is not the unit pattern `()`. Indexer properties called `Item` are special in the sense that they are accessible via the `.[]` notation. An `Item` property that takes one argument is accessed by using `x.[i]`; with two arguments by `x.[i,j]`, and so on. Setter properties must return type `unit`.

> **Note**: As of F# 3.1, the special `.[]` notation for Item properties is available only for instance members. A static indexer property cannot be accessible by using the `.[]` notation.

Property members may be declared `abstract`. If a property has both a getter and a setter, then both must be abstract or neither must be abstract.

Each property member has an implied property type. The property type is the type of the value that the getter property returns or the setter property accepts. If a property member has both a getter and a setter, and neither is an indexer property, the signatures of both the getter and the setter must imply the same property type.

Static and instance property members are evaluated every time the member is invoked. For example, in the following, the body of the member is evaluated each time `C.Time` is evaluated:

```
type C () =
    static member Time = System.DateTime.Now
```

Note that a static property member may also be written with an explicit `get` method:

```
static member ComputerName
    with get() = System.Environment.GetEnvironmentVariable("COMPUTERNAME")
```

Property members that have the same name may not appear in the same type definition even if their signatures are different. For example:

```
type C () =
    static member P = false // error: Duplicate property.
    member this.P = true
```

However, methods that have the same name can be overloaded when their signatures are different.

8.13.2 Auto-implemented Properties

Properties can be declared in two ways: either explicitly specified with the underlying value or automatically generated by the compiler. The compiler creates a backing field automatically if all of the following are true for the declaration:

- The declaration uses the `member val` keywords.
- The declaration omits the self-identifier.
- The declaration includes an expression to initialize the property.

To create a mutable property, include `with get`, `with set`,or both:

```
static_opt member val access_opt ident : ty_opt = expr
static_opt member val access_opt ident : ty_opt = expr with get
static_opt member val access_opt ident : ty_opt = expr with set
static_opt member val access_opt ident : ty_opt = expr with get, set
```

Automatically implemented properties are part of the initialization of a type, so they must be included before any other member definitions, in the same way as let bindings and do bindings in a type definition. The expression that initializes an automatically implemented property is evaluated only at initialization, and not every time the property is accessed. This behavior is different from the behavior of an explicitly implemented property.

For example, the following class type includes two automatically implemented properties. `Property1` is read-only and is initialized to the argument provided to the primary constructor and `Property2` is a settable property that is initialized to an empty string:

```
type D (x:int) =
    member val Property1 = x
    member val Property2 = "" with get, set
```

Auto-implemented properties can also be used to implement default or override properties:

```
type MyBase () =
   abstract Property : string with get, set
   default val Property = "default" with get, set

type MyDerived() =
   inherit MyBase()
    override val Property = "derived" with get, set
```

The following example shows how to use an auto-implemented property to implement an interface:

```
type MyInterface () =
   abstract Property : string with get, set

type MyImplementation () =
   interface MyInterface with
      member val Property = "implemented" with get, set
```

8.13.3 Method Members

A *method member* is of the form:

```
static_opt member ident._opt ident pat_1 ... pat_n = expr
```

The *ident*.$_{opt}$ can be present if and only if the property member is an instance member. In this case, the identifier *ident* corresponds to the “this” (or “self”) variable associated with the object on which the member is being invoked.

Arity analysis (§14.10) applies to method members. This is because F# members must compile to CLI methods, which accept only a single fixed collection of arguments.

8.13.4 Curried Method Members

Methods that take multiple arguments may be written in iterated ("curried") form. For example:

```
static member StaticMethod2 s1 s2 =
    sprintf "In StaticMethod(%s,%s)" s1 s2
```

The rules of arity analysis (§14.10) determine the compiled form of these members.

The following limitations apply to curried method members:

- Additional argument groups may not include optional or byref parameters.
- When the member is called, additional argument groups may not use named arguments(§8.13.5).
- Curried members may not be overloaded.

The compiled representation of a curried method member is a .NET method in which the arguments are concatenated into a single argument group.

> **Note**: It is recommended that curried argument members do not appear in the public API of an F# assembly that is designed for use from other .NET languages. Information about the currying order is not visible to these languages.

8.13.5 Named Arguments to Method Members

Calls to methods—but not to let-bound functions or function values—may use named arguments. For example:

```
System.Console.WriteLine(format = "Hello {0}", arg0 = "World")
System.Console.WriteLine("Hello {0}", arg0 = "World")
System.Console.WriteLine(arg0 = "World", format = "Hello {0}")
```

The argument names that are associated with a method declaration are derived from the names that appear in the first pattern of a member definition, or from the names used in the signature for a method member. For example:

```
type C() =
    member x.Swap(first, second) = (second, first)

let c = C()
c.Swap(first = 1,second = 2)  // result is '(2,1)'
c.Swap(second = 1,first = 2)  // result is '(1,2)'
```

Named arguments may be used only with the arguments that correspond to the arity of the member. That is, because members have an arity only up to the first set of tupled arguments, named arguments may not be used with subsequent curried arguments of the member.

The resolution of calls that use named arguments is specified in *Method Application Resolution* (see §14.4). The rules in that section describe how resolution matches a named argument with either a formal parameter of the same name or a “settable” return property of the same name. For example, the following code resolves the named argument to a settable property:

```
System.Windows.Forms.Form(Text = "Hello World")
```

If an ambiguity exists, assigning the named argument is assigned to a formal parameter rather than to a settable return property.

The *Method Application Resolution* (§14.4) rules ensure that:

- Named arguments must appear after all other arguments, including optional arguments that are matched by position.

After named arguments have been assigned, the remaining required arguments are called the *required unnamed arguments*. The required unnamed arguments must precede the named arguments in the argument list. The *n* unnamed arguments are matched to the first *n* formal parameters; the subsequent named arguments must include only the remaining formal parameters. In addition, the arguments must appear in the correct sequence.

For example, the following code is invalid:

```
// error: unnamed args after named
System.Console.WriteLine(arg0 = "World", "Hello {0}")
```

Similarly, the following code is invalid:

```
type Foo() =
    static member M (arg1, arg2, arg3) = 1
// error: arg1, arg3 not a prefix of the argument list
Foo.M(1, 2, arg2 = 3)
```

The following code is valid:

```
type Foo() =
    static member M (arg1, arg2, arg3) = 1

Foo.M (1, 2, arg3 = 3)
```

The names of arguments to members may be listed in member signatures. For example, in a signature file:

```
type C =
    static member ThreeArgs : arg1:int * arg2:int * arg3:int -> int
    abstract TwoArgs : arg1:int * arg2:int -> int
```

8.13.6 Optional Arguments to Method Members

Method members—but not functions definitions—may have optional arguments. Optional arguments must appear at the end of the argument list. An optional argument is marked with a `?`

before its name in the method declaration. Inside the member, the argument has type `option<argType>`.

The following example declares a method member that has two optional arguments:

```
let defaultArg x y = match x with None -> y | Some v -> v

type T() =
    static member OneNormalTwoOptional (arg1, ?arg2, ?arg3) =
        let arg2 = defaultArg arg2 3
        let arg3 = defaultArg arg3 10
        arg1 + arg2 + arg3
```

Optional arguments may be used in interface and abstract members. In a signature, optional arguments appear as follows:

```
static member OneNormalTwoOptional : arg1:int * ?arg2:int * ?arg3:int -
> int
```

Callers may specify values for optional arguments in the following ways:

- By name, such as `arg2 = 1`.
- By propagating an existing optional value by name, such as `?arg2=None` or `?arg2=Some(3)` or `?arg2=arg2`. This can be useful when building a method that passes optional arguments on to another method.
- By using normal, unnamed arguments that are matched by position.

For example:

```
T.OneNormalTwoOptional(3)
T.OneNormalTwoOptional(3, 2)
T.OneNormalTwoOptional(arg1 = 3)
T.OneNormalTwoOptional(arg1 = 3, arg2 = 1)
T.OneNormalTwoOptional(arg2 = 3, arg1 = 0)
T.OneNormalTwoOptional(arg2 = 3, arg1 = 0, arg3 = 11)
T.OneNormalTwoOptional(0, 3, 11)
T.OneNormalTwoOptional(0, 3, arg3 = 11)
T.OneNormalTwoOptional(arg1 = 3, ?arg2 = Some 1)
T.OneNormalTwoOptional(arg2 = 3, arg1 = 0, arg3 = 11)
T.OneNormalTwoOptional(?arg2 = Some 3, arg1 = 0, arg3 = 11)
T.OneNormalTwoOptional(0, 3, ?arg3 = Some 11)
```

The resolution of calls that use optional arguments is specified in *Method Application Resolution* (see §14.4).

Optional arguments may not be used in member constraints.

Note: Imported CLI metadata may specify arguments as optional and may additionally specify a default value for the argument. These are treated as F# optional arguments. CLI optional arguments can propagate an existing optional value by name; for example, `?ValueTitle = Some (…)`.

For example, here is a fragment of a call to a Microsoft Excel COM automation API that uses named and optional arguments.

```
chartobject.Chart.ChartWizard(Source = range5,
                              Gallery = XlChartType.xl3DColumn,
                              PlotBy = XlRowCol.xlRows,
                              HasLegend = true,
                              Title = "Sample Chart",
                              CategoryTitle = "Sample Category
Type",
                              ValueTitle = "Sample Value Type")
```

CLI optional arguments are not passed as values of type `Option<_>`. If the optional argument is present, its value is passed. If the optional argument is omitted, the default value from the CLI metadata is supplied instead. The value `System.Reflection.Missing.Value` is supplied for any CLI optional arguments of type `System.Object` that do not have a corresponding CLI default value, and the default (zero-bit pattern) value is supplied for other CLI optional arguments of other types that have no default value.

The compiled representation of members varies as additional optional arguments are added. The addition of optional arguments to a member signature results in a compiled form that is not binary-compatible with the previous compiled form.

Marking an argument as optional is equivalent to adding the `FSharp.Core.OptionalArgument` attribute (§17.1) to a required argument. This attribute is added implicitly for optional arguments. Adding the `[<OptionalArgument>]` attribute to a parameter of type `'a option` in a virtual method signature is equivalent to using the `(?x:'a)` syntax in a method definition. If the attribute is applied to an argument of a method, it should also be applied to all subsequent arguments of the method. Otherwise, it has no effect and callers must provide all of the arguments.

8.13.7 Type-directed Conversions at Member Invocations

As described in *Method Application Resolution* (see §14.4), two type-directed conversions are applied at method invocations.

The first type-directed conversion converts anonymous function expressions and other function-valued arguments to delegate types. Given:

- A formal parameter of delegate type *D*
- An actual argument *farg* of known type ty_1 `->` ... `->` ty_n `->` *rty*
- Precisely *n* arguments to the `Invoke` method of delegate type *D*

Then:

- The parameter is interpreted as if it were written:

  ```
  new D(fun arg1 ... argn -> farg arg1 ... argn)
  ```

If the type of the formal parameter is a variable type, then F# uses the known inferred type of the argument including instantiations to determine whether a formal parameter has delegate type. For example, if an explicit type instantiation is given that instantiates a generic type parameter to a delegate type, the following conversion can apply:

```
type GenericClass<'T>() =
    static member M(arg: 'T) = ()

GenericClass<System.Action>.M(fun () -> ())  // allowed
```

The second type-directed conversion enables an F# reference cell to be passed where a `byref<ty>` is expected. Given:

- A formal out parameter of type `byref<ty>`
- An actual argument that is not a byref type

Then:

- The actual parameter is interpreted as if it had type `ref<ty>`.

For example:

```
type C() =
    static member M1(arg: System.Action) = ()
    static member M2(arg: byref<int>) = ()

C.M1(fun () -> ())                    // allowed
let f = (fun () -> ()) in C.M1(f)    // not allowed

let result = ref 0
C.M2(result)    // allowed
```

> Note: These type-directed conversions are primarily for interoperability with existing member-based .NET libraries and do not apply at invocations of functions defined in modules or bound locally in expressions.

A value of type `ref<ty>` may be passed to a function that accepts a byref parameter. The interior address of the heap-allocated cell that is associated with such a parameter is passed as the pointer argument.

For example, consider the following C# code:

```
public class C
{
    static public void IntegerOutParam(out int x) { x = 3; }
}
public class D
```

```
{
    virtual public void IntegerOutParam(out int x) { x = 3; }
}
```

This C# code can be called by the following F# code:

```
let res1 = ref 0
C.IntegerOutParam(res1)
// res1.contents now equals 3
```

Likewise, the abstract signature can be implemented as follows:

```
let x = {new D() with IntegerOutParam(res : byref<int>) = res <- 4}
let res2 = ref 0
x.IntegerOutParam(res2);
// res2.contents now equals 4
```

8.13.8 Overloading of Methods

Multiple methods that have the same name may appear in the same type definition or extension. For example:

```
type MyForm() =
    inherit System.Windows.Forms.Form()

    member x.ChangeText(text: string) =
        x.Text <- text

    member x.ChangeText(text: string, reason: string) =
        x.Text <- text
        System.Windows.Forms.MessageBox.Show ("changing text due to " +
reason)
```

Methods must be distinct based on their name and fully inferred types, after erasure of type abbreviations and unit-of-measure annotations.

Methods that take curried arguments may not be overloaded.

8.13.9 Naming Restrictions for Members

A member in a record type may not have the same name as a record field in that type.

A member may not have the same name and signature as another method in the type. This check ignores return types except for members that are named `op_Implicit` or `op_Explicit`.

8.13.10 Members Represented as Events

Events are the CLI notion of a "listening point"—that is, a configurable object that holds a set of callbacks, which can be triggered, often by some external action such as a mouse click or timer tick.

In F#, events are first-class values; that is, they are objects that mediate the addition and removal of listeners from a backing list of listeners. The F# library supports the type `FSharp.Control.IEvent<_,_>` and the module `FSharp.Control.Event`, which contains operations to map, fold, create, and compose events. The type is defined as follows:

```
type IDelegateEvent<'del when 'del :> System.Delegate > =
    abstract AddHandler : 'del -> unit
    abstract RemoveHandler : 'del -> unit

type IEvent<'Del,'T when 'Del : delegate<'T,unit> and 'del :>
System.Delegate > =
    abstract Add : event : ('T -> unit) -> unit
    inherit IDelegateEvent<'del>

type Handler<'T> =  delegate of sender : obj * 'T -> unit

type IEvent<'T> = IEvent<Handler<'T>, 'T>
```

The following shows a sample use of events:

```
open System.Windows.Forms

type MyCanvas() =
    inherit Form()
    let event = new Event<PaintEventArgs>()
    member x.Redraw = event.Publish
    override x.OnPaint(args) = event.Trigger(args)

let form = new MyCanvas()
form.Redraw.Add(fun args -> printfn "OnRedraw")
form.Activate()
Application.Run(form)
```

Events from CLI languages are revealed as object properties of type `FSharp.Control.IEvent<`$ty_{delegate}$`, `ty_{args}`>`. The F# compiler determines the type arguments, which are derived from the CLI delegate type that is associated with the event.

Event declarations are not built into the F# language, and `event` is not a keyword. However, property members that are marked with the `CLIEvent` attribute and whose type coerces to `FSharp.Control.IDelegateEvent<`$ty_{delegate}$`>` are compiled to include extra CLI metadata and methods that mark the property name as a CLI event. For example, in the following code, the ChannelChanged property is currently compiled as a CLI event:

```
type ChannelChangedHandler = delegate of obj * int -> unit

type C() =
    let channelChanged = new Event<ChannelChangedHandler,_>()
    [<CLIEvent>]
    member self.ChannelChanged = channelChanged.Publish
```

Similarly, the following shows the definition and implementation of an abstract event:

```
type I =
    [<CLIEvent>]
    abstract ChannelChanged : IEvent<ChannelChanged,int>

type ImplI() =
    let channelChanged = new Event<ChannelChanged,_>()
    interface I with
        [<CLIEvent>]
        member self.ChannelChanged = channelChanged.Publish
```

8.13.11 Members Represented as Static Members

Most members are represented as their corresponding CLI method or property. However, in certain situations an instance member may be compiled as a static method. This happens when either of the following is true:

- The type definition uses `null` as a representation by placing the `CompilationRepresentation(CompilationRepresentationFlags.UseNullAsTrueValue)` attribute on the type that declares the member.
- The member is an extension member.

Compilation of an instance member as a static method can affect the view of the type when seen from other languages or from `System.Reflection.` A member that might otherwise have a static representation can be reverted to an instance member representation by placing the attribute `CompilationRepresentation(CompilationRepresentationFlags.Instance)` on the member.

For example, consider the following type:

```
[<CompilationRepresentation(CompilationRepresentationFlags.UseNullAsTru
eValue)>]
type option<'T> =
    | None
    | Some of 'T

    member x.IsNone = match x with None -> true | _ -> false
    member x.IsSome = match x with Some _ -> true | _ -> false


[<CompilationRepresentation(CompilationRepresentationFlags.Instance)>]
    member x.Item =
        match x with
        | Some x -> x
        | None -> failwith "Option.Item"
```

The `IsNone` and `IsSome` properties are represented as CLI static methods. The `Item` property is represented as an instance property.

8.14 Abstract Members and Interface Implementations

Abstract member definitions and interface declarations in a type definition represent promises that an object will provide an implementation for a corresponding contract.

8.14.1 Abstract Members

An *abstract member definition* in a type definition represents a promise that an object will provide an implementation for a dispatch slot. For example:

```
type IX =
    abstract M : int -> int
```

The abstract member `M` indicates that an object of type `IX` will implement a displatch slot for a member that returns an `int`.

A class definition may contain abstract member definitions, but the definition must be labeled with the `AbstractClass` attribute:

```
[<AbstractClass>]
type X() =
    abstract M : int -> int
```

An abstract member definition has the form

```
abstract access_opt member-sig
```

where a member signature has one of the following forms

```
ident typar-defns_opt : curried-sig
ident typar-defns_opt : curried-sig with get
ident typar-defns_opt : curried-sig with set
ident typar-defns_opt : curried-sig with get, set
ident typar-defns_opt : curried-sig with set, get
```

and the curried signature has the form

```
args-spec_1 -> ... -> args-spec_n -> type
```

If $n \geq 2$, then *args-spec*$_2$ … *args-spec*$_n$ must all be patterns without attribute or optional argument specifications.

If `get` or `set` is specified, the abstract member is a *property member*. If both `get` and `set` are specified, the abstract member is equivalent to two abstract members, one with `get` and one with `set`.

8.14.2 Members that Implement Abstract Members

An *implementation member* has the form:

```
override ident.ident pat_1 ... pat_n = expr
default ident.ident pat_1 ... pat_n = expr
```

Implementation members implement dispatch slots. For example:

```
[<AbstractClass>]
type BaseClass() =
    abstract AbstractMethod : int -> int

type SubClass(x: int) =
    inherit BaseClass()
    override obj.AbstractMethod n = n + x

let v1  = BaseClass()               // not allowed – BaseClass is 
abstract
let v2  = (SubClass(7) :> BaseClass)

v2.AbstractMethod 6  // evaluates to 13
```

In this example, BaseClass() declares the abstract slot AbstractMethod and the SubClass type supplies an implementation member obj.AbstractMethod, which takes an argument n and returns the sum of n and the argument that was passed in the instantiation of SubClass. The v2 object instantiates SubClass with the value 7, so v2.AbstractMethod 6 evaluates to 13.

The combination of an abstract slot declaration and a default implementation of that slot create the F# equivalent of a “virtual” method in some other languages—that is, an abstract member that is guaranteed to have an implementation. For example:

```
type BaseClass() =
    abstract AbstractMethodWithDefaultImplementation : int -> int
    default obj.AbstractMethodWithDefaultImplementation n = n

type SubClass1(x: int) =
    inherit BaseClass()
    override obj.AbstractMethodWithDefaultImplementation n = n + x

type SubClass2() =
    inherit BaseClass()

let v1  = BaseClass()    // allowed -- BaseClass contains a default 
implementation
let v2  = (SubClass1(7) :> BaseClass)
let v3  = (SubClass2() :> BaseClass)

v1.AbstractMethodWithDefaultImplementation 6  // evaluates to 6
v2.AbstractMethodWithDefaultImplementation 6  // evaluates to 13
v3.AbstractMethodWithDefaultImplementation 6  // evaluates to 6
```

Here, the BaseClass type contains a default implementation, so F# allows the instantiation of v1. The instantiation of v2 is the same as in the previous example. The instantiation of v3 is similar to that of v1, because SubClass2 inherits directly from BaseClass and does not override the default method.

> Note: The keywords `override` and `default` are synonyms. However, it is recommended that `default` be used only when the implementation is in the same class as the corresponding `abstract` definition; `override` should be used in other cases. This records the intended role of the member implementation.

Implementations may override methods from `System.Object`:

```
type BaseClass() =
    override obj.ToString() = "I'm an instance of BaseClass"

type SubClass(x: int) =
    inherit BaseClass()
    override obj.ToString() = "I'm an instance of SubClass"
```

In this example, `BaseClass` inherits from `System.Object` and overrides the `ToString` method from that class. The `SubClass`, in turn, inherits from `BaseClass` and overrides its version of the `ToString` method.

Implementations may include abstract property members:

```
[<AbstractClass>]
type BaseClass() =
    let mutable data1 = 0
    let mutable data2 = 0
    abstract AbstractProperty : int
    abstract AbstractSettableProperty : int  with get, set

    abstract AbstractPropertyWithDefaultImplementation : int
    default obj.AbstractPropertyWithDefaultImplementation = 3

    abstract AbstractSettablePropertyWithDefaultImplementation : int
with get, set
    default obj.AbstractSettablePropertyWithDefaultImplementation
        with get() = data2
        and set v = data2 <- v

type SubClass(x: int) =
    inherit BaseClass()
    let mutable data1b = 0
    let mutable data2b = 0
    override obj.AbstractProperty = 3 + x
    override obj.AbstractSettableProperty
        with get() = data1b  + x
        and set v = data1b <- v - x
    override obj.AbstractPropertyWithDefaultImplementation = 6 + x
    override obj.AbstractSettablePropertyWithDefaultImplementation
        with get() = data2b  + x
        and set v = data2b <- v - x
```

The same rules apply to both property members and method members. In the preceding example, `BaseClass` includes abstract properties named `AbstractProperty`, `AbstractSettableProperty`, `AbstractPropertyWithDefaultImplementation`, and `AbstractSettablePropertyWithDefaultImplementation` and provides default implementations for the latter two. `SubClass` provides implementations for `AbstractProperty` and `AbstractSettableProperty`, and overrides the default implementations for `AbstractPropertyWithDefaultImplementation` and `AbstractSettablePropertyWithDefaultImplementation`.

Implementation members may also implement CLI events (§8.13.10). In this case, the member should be marked with the `CLIEvent` attribute. For example:

```
type ChannelChangedHandler = delegate of obj * int -> unit

[<AbstractClass>]
type BaseClass() =
    [<CLIEvent>]
    abstract ChannelChanged : IEvent<ChannelChangedHandler, int>

type SubClass() =
    inherit BaseClass()
    let mutable channel = 7
    let channelChanged = new Event<ChannelChangedHandler, int>()

    [<CLIEvent>]
    override self.ChannelChanged = channelChanged.Publish
    member self.Channel
        with get () = channel
        and set v = channel <- v; channelChanged.Trigger(self, channel)
```

`BaseClass` implements the CLI event `IEvent`, so the abstract member `ChannelChanged` is marked with `[<CLIEvent>]` as described earlier in §8.13.10. `SubClass` provides an implementation of the abstract member, so the `[<CLIEvent>]` attribute must also precede the `override` declaration in `SubClass`.

8.14.3 Interface Implementations

An *interface implementation* specifies how objects of a given type support a particular interface. An interface in a type definition indicates that objects of the defined type support the interface. For example:

```
type IIncrement =
    abstract M : int -> int

type IDecrement =
    abstract M : int -> int

type C() =
    interface IIncrement with
```

```
        member x.M(n) = n + 1
    interface IDecrement with
        member x.M(n) = n - 1
```

The first two definitions in the example are implementations of the interfaces `IIncrement` and `IDecrement`. In the last definition,the type `C` supports these two interfaces.

No type may implement multiple different instantiations of a generic interface, either directly or through inheritance. For example, the following is not permitted:

```
// This type definition is not permitted because it implements two
instantiations
// of the same generic interface
type ClassThatTriesToImplemenTwoInstantiations() =
    interface System.IComparable<int> with
        member x.CompareTo(n : int) = 0
    interface System.IComparable<string> with
        member x.CompareTo(n : string) = 1
```

Each member of an interface implementation is checked as follows:

- The member must be an instance member definition.
- *Dispatch Slot Inference* (§14.7) is applied.
- The member is checked under the assumption that the “this” variable has the enclosing type.

In the following example, the value `x` has type `C`.

```
type C() =
    interface IIncrement with
        member x.M(n) = n + 1
    interface IDecrement with
        member x.M(n) = n - 1
```

All interface implementations are made explicit. In its first implementation, every interface must be completely implemented, even in an abstract class. However, interface implementations may be inherited from a base class. In particular, if a class `C` implements interface `I`, and a base class of `C` implements interface `I`, then `C` is not required to implement all the methods of `I`;it can implement all, some, or none of the methods instead. For example:

```
type I1 =
    abstract V1 : string
    abstract V2 : string

type I2 =
    inherit I1
    abstract V3 : string

type C1() =
    interface I1 with
        member this.V1 = "C1"
```

```
        member this.V2 = "C2"

// This is OK
type C2() =
    inherit C1()

// This is also OK; C3 implements I2 but not I1.
type C3() =
    inherit C1()
    interface I2 with
        member this.V3 = "C3"

// This is also OK; C4 implements one method in I1.
type C4() =
    inherit C1()
    interface I1 with
        member this.V2 = "C2b"
```

8.15 Equality, Hashing, and Comparison

Functional programming in F# frequently involves the use of structural equality, structural hashing, and structural comparison. For example, the following expression evaluates to `true`, because tuple types support structural equality:

```
(1, 1 + 1) = (1, 2)
```

Likewise, these two function calls return identical values:

```
hash (1, 1 +1 )
hash (1,2)
```

Similarly, an ordering on constituent parts of a tuple induces an ordering on tuples themselves, so all the following evaluate to `true`:

```
(1, 2) < (1, 3)
(1, 2) < (2, 3)
(1, 2) < (2, 1)
(1, 2) > (1, 0)
```

The same applies to lists, options, arrays, and user-defined record, union, and struct types whose constituent field types permit structural equality, hashing, and comparison. For example, given:

```
type R = R of int * int
```

then all of the following also evaluate to `true`:

```
R (1, 1 + 1) = R (1, 2)

R (1, 3) <> R (1, 2)
```

```
hash (R (1, 1 + 1)) = hash (R (1, 2))

R (1, 2) < R (1, 3)
R (1, 2) < R (2, 3)
R (1, 2) < R (2, 1)
R (1, 2) > R (1, 0)
```

To facilitate this, by default, record, union, and struct type definitions—called *structural types*—implicitly include compiler-generated declarations for structural equality, hashing, and comparison. These implicit declarations consist of the following for structural equality and hashing:

```
override x.GetHashCode() = ...
override x.Equals(y:obj) = ...
interface System.Collections.IStructuralEquatable with
    member x.Equals(yobj: obj, comparer:
System.Collections.IEqualityComparer) = ...
    member x.GetHashCode(comparer: System.IEqualityComparer) = ...
```

The following declarations enable structural comparison:

```
interface System.IComparable with
    member x.CompareTo(y:obj) = ...
interface System.Collections.IStructuralComparable with
    member x.CompareTo(yobj: obj, comparer:
System.Collections.IComparer) = ...
```

For exception types, implicit declarations for structural equality and hashings are generated, but declarations for structural comparison are not generated. Implicit declarations are never generated for interface, delegate, class, or enum types. Enum types implicitly derive support for equality, hashing, and comparison through their underlying representation as integers.

8.15.1 Equality Attributes

Several attributes affect the equality behavior of types:

```
FSharp.Core.NoEquality
FSharp.Core.ReferenceEquality
FSharp.Core.StructuralEquality
FSharp.Core.CustomEquality
```

The following table lists the effects of each attribute on a type:

Attrribute	Effect
`NoEquality`	▪ No equality or hashing is generated for the type. ▪ The type does not satisfy the `ty : equality` constraint.
`ReferenceEquality`	▪ No equality or hashing is generated for the type. ▪ The defaults for `System.Object` will implicitly be used.
`StructuralEquality`	▪ The type must be a structural type. ▪ All structural field types `ty` must satisfy `ty : equality`.
`CustomEquality`	▪ The type must have an explicit implementation of `override Equals(obj: obj)`

Attrribute	Effect
None	▪ For a non-structural type, the default is `ReferenceEquality`. ▪ For a structural type: The default is `NoEquality` if any structural field type `F` fails `F : equality`. The default is `StructuralEquality` if all structural field types `F` satisfy `F : equality`.

Equality inference also determines the *constraint dependencies* of a generic structural type. That is:

- If a structural type has a generic parameter `'T` and `T : equality` is necessary to make the type default to `StructuralEquality`, then the `EqualityConditionalOn` constraint dependency is inferred for `'T`.

8.15.2 Comparison Attributes

The comparison behavior of types can be affected by the following attributes:

```
FSharp.Core.NoComparison
FSharp.Core.StructuralComparison
FSharp.Core.CustomComparison
```

The following table lists the effects of each attribute on a type.

Attribute	Effect
`NoComparison`	▪ No comparisons are generated for the type. ▪ The type does not satisfy the *ty* `: comparison` constraint.
`StructuralComparison`	▪ The type must be a structural type other than an exception type. ▪ All structural field types must *ty* satisfy *ty* `: comparison`. ▪ An exception type may not have the `StructuralComparison` attribute.

`CustomComparison`	▪ The type must have an explicit implementation of one or both of the following:
	`interface System.IComparable`
	`interface System.Collections.IStructuralComparable`
	▪ A structural type that has an explicit implementation of one or both of these contracts must specify the `CustomComparison` attribute.

None	▪ For a non-structural or exception type, the default is `NoComparison`. ▪ For any other structural type: The default is `NoComparison` if any structural field type `F` fails `F : comparison`. The default is `StructuralComparison` if all structural field types `F` satisfy `F : comparison`.

This check also determines the *constraint dependencies* of a generic structural type. That is:

- If a structural type has a generic parameter `'T` and `T : comparison` is necessary to make the type default to `StructuralComparison`, then the `ComparisonConditionalOn` constraint dependency is inferred for `'T`.

For example:

```
[<StructuralEquality; StructuralComparison>]
type X = X of (int -> int)
```

results in the following message:

```
The struct, record or union type 'X' has the 'StructuralEquality' 
attribute
but the component type '(int -> int)' does not satisfy the 'equality' 
constraint
```

For example, given

```
type R1 =
    { myData : int }
    static member Create() = { myData = 0 }

[<ReferenceEquality>]
type R2 =
    { mutable myState : int }
    static member Fresh() = { myState = 0 }

[<StructuralEquality; NoComparison >]
type R3 =
    { someType : System.Type }
    static member Make() = { someType = typeof<int> }
```

then the following expressions all evaluate to `true`:

```
R1.Create() = R1.Create()
not (R2.Fresh() = R2.Fresh())
R3.Make() = R3.Make()
```

Combinations of equality and comparion attributes are restricted. If any of the following attributes are present, they may be used only in the following combinations:

- No attributes
- `[<NoComparison>]` on any type
- `[<NoEquality; NoComparison>]` on any type
- `[<CustomEquality; NoComparison>]` on a structural type
- `[<ReferenceEquality>]` on a non-struct structural type
- `[<ReferenceEquality; NoComparison>]` on a non-struct structural type
- `[<StructuralEquality; NoComparison>]` on a structural type
- `[<CustomEquality; CustomComparison>]` on a structural type
- `[<StructuralEquality; CustomComparison>]` on a structural type
- `[<StructuralEquality; StructuralComparison>]` on a structural type

8.15.3 Behavior of the Generated Object.Equals Implementation

For a type definition `T`, the behavior of the generated `override x.Equals(y:obj) = ...` implementation is as follows.

1. If the interface `System.IComparable` has an explicit implementation, then just call `System.IComparable.CompareTo`:

```
override x.Equals(y : obj) =
    ((x :> System.IComparable).CompareTo(y) = 0)
```

2. Otherwise:

   - Convert the `y` argument to type `T`. If the conversion fails, return `false`.
   - Return `false` if `T` is a reference type and `y` is null.
   - If `T` is a struct or record type, invoke `FSharp.Core.Operators.(=)` on each corresponding pair of fields of `x` and `y` in declaration order. This method stops at the first `false` result and returns `false`.
   - If `T` is a union type, invoke `FSharp.Core.Operators.(=)` first on the index of the union cases for the two values, then on each corresponding field pair of `x` and `y` for the data carried by the union case. This method stops at the first `false` result and returns `false`.
   - If `T` is an exception type, invoke `FSharp.Core.Operators.(=)` on the index of the tags for the two values, then on each corresponding field pair for the data carried by the exception. This method stops at the first `false` result and returns `false`.

8.15.4 Behavior of the Generated CompareTo Implementations

For a type `T`, the behavior of the generated `System.IComparable.CompareTo` implementation is as follows:

- Convert the `y` argument to type `T` . If the conversion fails, raise the `InvalidCastException`.
- If `T` is a reference type and `y` is null, return `1`.
- If `T` is a struct or record type, invoke `FSharp.Core.Operators.compare` on each corresponding pair of fields of `x` and `y` in declaration order, and return the first non-zero result.
- If `T` is a union type, invoke `FSharp.Core.Operators.compare` first on the index of the union cases for the two values, and then on each corresponding field pair of `x` and `y` for the data carried by the union case. Return the first non-zero result.

The first few lines of this code can be written:

```
interface System.IComparable with
    member x.CompareTo(y:obj) =
        let y = (obj :?> T) in
        match obj with
        | null -> 1
        | _ -> ...
```

8.15.5 Behavior of the Generated GetHashCode Implementations

For a type `T`, the generated `System.Object.GetHashCode()` override implements a combination hash of the structural elements of a structural type.

8.15.6 Behavior of Hash, =, and Compare

The generated equality, hashing, and comparison declarations that are described in sections 8.15.3, 8.15.4, and 8.15.5 use the `hash`, `=` and `compare` functions from the F# library. The behavior of these library functions is defined by the pseudocode later in this section. This code ensures:

- Ordinal comparison for strings
- Structural comparison for arrays
- Natural ordering for native integers (which do not support `System.IComparable`)

8.15.6.1 Pseudocode for FSharp.Core.Operators.compare

> Note: In practice, fast (but semantically equivalent) code is emitted for direct calls to `(=)`, `compare`, and `hash` for all base types, and faster paths are used for comparing most arrays.

```
open System

/// Pseudo code for code implementation of generic comparison.
let rec compare x y =
    let xobj = box x
    let yobj = box y
    match xobj, yobj  with
    | null, null -> 0
    | null, _ -> -1
    | _, null -> 1

    // Use Ordinal comparison for strings
    | (:? string as x),(:? string as y) ->
        String.CompareOrdinal(x, y)

    // Special types not supporting IComparable
    | (:? Array as arr1), (:? Array as arr2) ->
        ... compare the arrays by rank, lengths and elements ...
    | (:? nativeint as x),(:? nativeint as y) ->
        ... compare the native integers x and y....
    | (:? unativeint as x),(:? unativeint as y) ->
        ... compare the unsigned integers x and y....

    // Check for IComparable
    | (:? IComparable as x),_ -> x.CompareTo(yobj)
    | _,(:? IComparable as yc) -> -(sign(yc.CompareTo(xobj)))

    // Otherwise raise a runtime error
    | _ -> raise (new ArgumentException(...))
```

8.15.6.2 Pseudo code for FSharp.Core.Operators.(=)

Note: In practice, fast (but semantically equivalent) code is emitted for direct calls to `(=)`, `compare`, and `hash` for all base types, and faster paths are used for comparing most arrays

```
open System

/// Pseudo code for core implementation of generic equality.
let rec (=) x y =
      let xobj = box x
      let yobj = box y
      match xobj,yobj with
       | null,null -> true
       | null,_ -> false
       | _,null -> false

       // Special types not supporting IComparable
       | (:? Array as arr1), (:? Array as arr2) ->
           ... compare the arrays by rank, lengths and elements ...

       // Ensure NaN semantics on recursive calls
       | (:? float as f1), (:? float as f2) ->
           ... IEEE equality on f1 and f2...
       | (:? float32 as f1), (:? float32 as f2) ->
           ... IEEE equality on f1 and f2...

       // Otherwise use Object.Equals. This is reference equality
       // for reference types unless an override is provided
(implicitly
       // or explicitly).
       | _ -> xobj.Equals(yobj)
```

9. Units Of Measure

F# supports static checking of *units of measure*. Units of measure, or *measures* for short, are like types in that they can appear as parameters to other types and values (as in `float<kg>`, `vector<m/s>`, `add<m>`), can contain variables (as in `float<'U>`), and are checked for consistency by the type-checker.

However, measures differ from types in several important ways:

- Measures play no role at runtime; in fact, they are erased.
- Measures obey special rules of *equivalence*, so that `N m` can be interchanged with `m N`.
- Measures are supported by special syntax.

The syntax of constants (§4.3) is extended to support numeric constants with units of measure. The syntax of types is extended with measure type annotations.

```
measure-literal-atom :=
    long-ident      -- named measure e.g. kg
    ( measure-literal-simp )       -- parenthesized measure, such as
(N m)

measure-literal-power :=
    measure-literal-atom
    measure-literal-atom ^ int32   -- power of measure, such as m^3

measure-literal-seq :=
    measure-literal-power
    measure-literal-power measure-literal-seq

measure-literal-simp :=
    measure-literal-seq            -- implicit product, such as m s^-
2
    measure-literal-simp * measure-literal-simp   -- product, such as
m * s^3
    measure-literal-simp / measure-literal-simp   -- quotient, such
as m/s^2
    / measure-literal-simp -- reciprocal, such as /s
    1        -- dimensionless

measure-literal :=
    _        -- anonymous measure
    measure-literal-simp           -- simple measure, such as N m

const :=
    ...
    sbyte < measure-literal >      -- 8-bit integer constant
    int16 < measure-literal >      -- 16-bit integer constant
    int32 < measure-literal >      -- 32-bit integer constant
    int64 < measure-literal >      -- 64-bit integer constant
```

```
    ieee32 < measure-literal >      -- single-precision float32
constant
    ieee64 < measure-literal >      -- double-precision float constant
    decimal < measure-literal >     -- decimal constant

measure-atom :=
    typar                   -- variable measure, such as 'U
    long-ident                      -- named measure, such as kg
    ( measure-simp )                -- parenthesized measure, such as
(N m)

measure-power :=
    measure-atom
    measure-atom ^ int32            -- power of measure, such as m^3

measure-seq :=
    measure-power
    measure-power measure-seq

measure-simp :=
    measure-seq                     -- implicit product, such as 'U
'V^3
    measure-simp * measure-simp     -- product, such as 'U * 'V
    measure-simp / measure-simp     -- quotient, such as 'U / 'V
    / measure-simp          -- reciprocal, such as /'U
    1       -- dimensionless measure (no units)

measure :=
    _                       -- anonymous measure
    measure-simp    -- simple measure, such as 'U 'V
```

Measure definitions use the special `Measure` attribute on type definitions. Measure parameters use the syntax of generic parameters with the same special `Measure` attribute to parameterize types and members by units of measure. The primitive types `sbyte`, `int16`, `int32`, `int64`, `float`, `float32`, and `decimal` have non-parameterized (dimensionless) and parameterized versions.

Here is a simple example:

```
[<Measure>] type m                  // base measure: meters
[<Measure>] type s                  // base measure: seconds
[<Measure>] type sqm = m^2          // derived measure: square meters
let areaOfTriangle (baseLength:float<m>, height:float<m>) : float<sqm>
=
    baseLength*height/2.0

let distanceTravelled (speed:float<m/s>, time:float<s>) : float<m> =
speed*time
```

As with ordinary types, F# can infer that functions are generic in their units. For example, consider the following function definitions:

```
let sqr (x:float<_>) = x*x
```

```
let sumOfSquares x y = sqr x + sqr y
```

The inferred types are:

```
val sqr : float<'u> -> float<'u ^ 2>

val sumOfSquares : float<'u> -> float<'u> -> float<'u ^ 2>
```

Measures are type-like annotations such as `kg` or `m/s` or `m^2`. Their special syntax includes the use of `*` and `/` for product and quotient of measures, juxtaposition as shorthand for product, and `^` for integer powers.

9.1 Measures

Measures are built from:

- *Atomic measures* from long identifiers such as `SI.kg` or `MyUnits.feet`.
- *Product measures*, which are written *`measure measure`* (juxtaposition) or *`measure * measure`*.
- *Quotient measures*, which are written *`measure / measure`*.
- *Integer powers of measures*, which are written *`measure ^ int`*.
- *Dimensionless measures*, which are written `1`.
- Variable measures, which are written `'u` or `'U`. Variable measures can include anonymous measures `_`, which indicates that the compiler can infer the measure from the context.

Dimensionless measures indicate "without units," but are rarely needed, because non-parameterized types such as `float` are aliases for the parameterized type with `1` as parameter, that is, `float = float<1>`.

The precedence of operations involving measure is similar to that for floating-point expressions:

- Products and quotients (`*` and `/`) have the same precedence, and associate to the left, but juxtaposition has higher syntactic precedence than both `*` and `/`.
- Integer powers (`^`) have higher precedence than juxtaposition.
- The `/` symbol can also be used as a unary reciprocal operator.

9.2 Constants Annotated by Measures

A floating-point constant can be annotated with its measure by specifying a literal measure in angle brackets following the constant.

Measure annotations on constants may not include measure variables.

Here are some examples of annotated constants:

```
let earthGravity = 9.81f<m/s^2>
let atmosphere = 101325.0<N m^-2>
let zero = 0.0f<_>
```

Constants that are annotated with units of measure are assigned a corresponding numeric type with the measure parameter that is specified in the annotation. In the example above, `earthGravity` is assigned the type `float32<m/s^2>`, `atmosphere` is assigned the type `float<N/m^2>` and `zero` is assigned the type `float<'U>`.

9.3 Relations on Measures

After measers are parsed and checked, they are maintained in the following normalized form:

```
measure-int := 1 | long-ident | measure-par | measure-int measure-int |
/ measure-int
```

Powers of measures are expanded. For example, `kg^3` is equivalent to `kg kg kg`.

Two measures are indistinguishable if they can be made equivalent by repeated application of the following rules:

- *Commutativity*. *`measure-int`*$_1$ *`measure-int`*$_2$ is equivalent to *`measure-int`*$_2$ *`measure-int`*$_1$.
- *Associativity*. It does not matter what grouping is used for juxtaposition (product) of measures, so parentheses are not required. For example, `kg m s` can be split as the product of `kg m` and `s`, or as the product of `kg` and `m s`.
- *Identity*. `1` *`measure-int`* is equivalent to *`measure-int`*.
- *Inverses*. *`measure-int`* `/` *`measure-int`* is equivalent to `1`.
- *Abbreviation*. *`long-ident`* is equivalent to *`measure`* if a measure abbreviation of the form `[<Measure>] type` *`long-ident`* `=` *`measure`* is currently in scope.

Note that these are the laws of Abelian groups together with expansion of abbreviations.

For example, `kg m / s^2` is the same as `m kg / s^2`.

For presentation purposes (for example, in error messages), measures are presented in the normalized form that appears at the beginning of this section, but with the following restrictions:

- Powers are positive and greater than 1. This splits the measure into positive powers and negative powers, separated by `/`.
- Atomic measures are ordered as follows: measure parameters first, ordered alphabetically, followed by measure identifiers, ordered alphabetically.

For example, the measure expression `m^1 kg s^-1` would be normalized to `kg m / s`.

This normalized form provides a convenient way to check the equality of measures: given two measure expressions *`measure-int`*$_1$ and *`measure-int`*$_2$, reduce each to normalized form by using the rules of commutativity, associativity, identity, inverses and abbreviation, and then compare the syntax.

To check the equality of two measures, abbreviations are expanded to compare their normalized forms. However, abbreviations are not expanded for presentation. For example, consider the following definitions:

```
[<Measure>] type a
[<Measure>] type b = a * a
let x = 1<b> / 1<a>
```

The inferred type is presented as `int<b/a>`, not `int<a>`. If a measure is equivalent to `1`, however, abbreviations are expanded to cancel each other and are presented without units:

```
let y = 1<b> / 1<a a> // val y : int = 1
```

9.3.1 Constraint Solving

The mechanism described in §14.5 is extended to support equational constraints between measure expressions. Such expressions arise from equations between parameterized types—that is, when $\mathit{type}\langle\mathit{tyarg}_{11},\ldots,\ \mathit{tyarg}_{1n}\rangle = \mathit{type}\langle\mathit{tyarg}_{21},\ldots,\ \mathit{tyarg}_{2n}\rangle$ is reduced to a series of constraints $\mathit{tyarg}_{1i} = \mathit{tyarg}_{2i}$. For the arguments that are measures, rather than types, the rules listed in §9.3 are applied to obtain primitive equations of the form `'U =` *`measure-int`* where `'U` is a measure variable and *`measure-int`* is a measure expression in internal form. The variable `'U` is then replaced by *`measure-int`* wherever else it occurs. For example, the equation `float<m^2/s^2> = float<'U^2>` would be reduced to the constraint `m^2/s^2 = 'U^2`, which would be further reduced to the primitive equation `'U = m/s`.

If constraints cannot be solved, a type error occurs. For example, the following expression

```
fun (x : float<m^2>, y : float<s>) -> x + y
```

would eventually)result in the constraint `m^2 = s`, which cannot be solved, indicating a type error.

9.3.2 Generalization of Measure Variables

Analogous to the process of generalization of type variables described in §14.6.7, a generalization procedure produces measure variables over which a value, function, or member can be generalized.

9.4 Measure Definitions

Measure definitions define new named units of measure by using the same syntax as for type definitions, with the addition of the `Measure` attribute. For example:

```
[<Measure>] type kg
[<Measure>] type m
[<Measure>] type s
[<Measure>] type N = kg / m s^2
```

A primitive measure abbreviation defines a fresh, named measure that is distinct from other measures. Measure abbreviations, like type abbreviations, define new names for existing measures. Also like type abbreviations, repeatedly eliminating measure abbreviations in favor of their equivalent measures must not result in infinite measure expressions. For example, the following is not a valid measure definition because it results in the infinite squaring of `X`:

```
[<Measure>] type X = X^2
```

Measure definitions and abbreviations may not have type or measure parameters.

9.5 Measure Parameter Definitions

Measure parameter definitions can appear wherever ordinary type parameter definitions can (see §5.2.9). If an explicit parameter definition is used, the parameter name is prefixed by the special `Measure` attribute. For example:

```
val sqr<[<Measure>] 'U> : float<'U> -> float<'U^2>

type Vector<[<Measure>] 'U> =
    { X: float<'U>;
      Y: float<'U>;
      Z: float<'U>}

type Sphere<[<Measure>] 'U> =
    { Center:Vector<'U>;
      Radius:float<'U> }

type Disc<[<Measure>] 'U> =
    { Center:Vector<'U>;
      Radius:float<'U>;
      Norm:Vector<1> }

type SceneObject<[<Measure>] 'U> =
    | Sphere of Sphere<'U>
    | Disc of Disc<'U>
```

Internally, the type checker distinguishes between type parameters and measure parameters by assigning one of two *sorts* (Type or Measure) to each parameter. This technique is used to check the actual arguments to types and other parameterized definitions. The type checker rejects ill-formed types such as `float<int>` and `IEnumerable<m/s>`.

9.6 Measure Parameter Erasure

In contrast to *type* parameters on generic types, *measure* parameters are not exposed in the metadata that the runtime interprets; instead, measures are *erased*. Erasure has several consequences:

- Casting is with respect to erased types.
- Method application resolution (see §14.4) is with respect to erased types.
- Reflection is with respect to erased types.

9.7 Type Definitions with Measures in the F# Core Library

The F# core library defines the following types:

```
type float<[<Measure>] 'U>
type float32<[<Measure>] 'U>
type decimal<[<Measure>] 'U>
type int<[<Measure>] 'U>
type sbyte<[<Measure>] 'U>
type int16<[<Measure>] 'U>
type int64<[<Measure>] 'U>
```

> Note: These definitions are called *measure-annotated base types* and are marked with the `MeasureAnnotatedAbbreviation` attribute in the implementation of the library. The `MeasureAnnotatedAbbreviation` attribute is not for use in user code and in future revisions of the language may result in a warning or error.

These type definitions have the following special properties:

- They extend `System.ValueType`.
- They explicitly implement `System.IFormattable`, `System.IComparable`, `System.IConvertible`, and corresponding generic interfaces, instantiated at the given type—for example, `System.IComparable<float<'u>>` and `System.IEquatable<float<'u>>` (so that you can invoke, for example, `CompareTo` after an explicit upcast).
- As a result of erasure, their compiled form is the corresponding primitive type.
- For the purposes of constraint solving and other logical operations on types, a type equivalence holds between the unparameterized primitive type and the corresponding measured type definition that is instantiated at `<1>`:

  ```
  sbyte = sbyte<1>
  int16 = int16<1>
  int32 = int32<1>
  int64 = int64<1>
  float = float<1>
  float32 = float32<1>
  decimal = decimal<1>
  ```

- The measured type definitions `sbyte`, `int16`, `int32`, `int64`, `float32`, `float`, and `decimal` are assumed to have additional static members that have the measure types that are listed in the table. Note that `N` is any of these types, and `F` is either `float32` or `float`.

Member	Measure Type
`Sqrt`	`F<'U^2> -> F<'U>`

Member	Measure Type
`Atan2`	`F<'U> -> F<'U> -> F<1>`

`op_Addition`
`op_Subtraction`
`op_Modulus`
	`N<'U> -> N<'U> -> N<'U>`

`op_Multiply`	`N<'U> -> N<'V> -> N<'U 'V>`
`op_Division`	`N<'U> -> N<'V> -> N<'U/'V>`

`Abs`
`op_UnaryNegation`
`op_UnaryPlus`
	`N<'U> -> N<'U>`

`Sign`	`N<'U> -> int`

This mechanism is used to support units of measure in the following math functions of the F# library:
`(+)`,`(-)`, `(*)`, `(/)`,`(%)`,`(~+)`,`(~-)`,`abs`, `sign`, `atan2` and `sqrt`.

9.8 Restrictions

Measures can be used in range expressions but a properly measured step is required. For example, these are not allowed:

```
[<Measure>] type s
[1<s> .. 5<s>]       // error: The type 'int<s>' does not match the
type 'int'
[1<s> .. 1 .. 5<s>]  // error: The type 'int<s>' does not match the
type 'int'
```

However, the following range expression is valid:

```
[1<s> .. 1<s> .. 5<s>]     // int<s> list = [1; 2; 3; 4; 5]
```

10. Namespaces and Modules

F# is primarily an expression-based language. However, F# source code units are made up of *declarations*, some of which can contain further declarations. Declarations are grouped using *namespace declaration groups*, *type definitions*, and *module definitions*. These also have corresponding forms in *signatures*. For example, a file may contain multiple namespace declaration groups, each of which defines types and modules, and the types and modules may contain member, function, and value definitions, which contain expressions.

Declaration elements are processed in the context of an *environment*. The definition of the elements of an environment is found in §14.1.

```
namespace-decl-group :=
    namespace long-ident module-elems      -- elements within a namespace
    namespace global module-elems          -- elements within no namespace

module-defn :=
    attributes_opt module access_opt ident = module-defn-body

module-defn-body :=
    begin module-elems_opt end

module-elem :=
    module-function-or-value-defn          -- function or value definitions
    type-defns                             -- type definitions
    exception-defn                         -- exception definitions
    module-defn                            -- module definitions
    module-abbrev                          -- module abbreviations
    import-decl                            -- import declarations
    compiler-directive-decl                -- compiler directives

module-function-or-value-defn :=
    attributes_opt let function-defn
    attributes_opt let value-defn
    attributes_opt let rec_opt function-or-value-defns
    attributes_opt do expr

import-decl := open long-ident

module-abbrev := module ident = long-ident

compiler-directive-decl := # ident string ... string

module-elems := module-elem ... module-elem

access :=
    private
```

```
    internal
    public
```

10.1 Namespace Declaration Groups

Modules and types in an F# program are organized into *namespaces*, which encompass the identifiers that are defined in the modules and types. New components may contribute entities to existing namespaces. Each such contribution to a namespace is called a *namespace declaration group*.

In the following example, the `MyCompany.MyLibrary` namespace contains `Values` and `x`:

```
namespace MyCompany.MyLibrary

    module Values1 =
        let x = 1
```

A namespace declaration group is the basic declaration unit within an F# implementation file and is of the form

```
namespace long-ident

    module-elems
```

The `long-ident` must be fully qualified. Each such group contains a series of module and type definitions that contribute to the indicated namespace. An implementation file may contain multiple namespace declaration groups, as in this example:

```
namespace MyCompany.MyOtherLibrary

    type MyType() =
      let x = 1
        member v.P = x + 2

    module MyInnerModule =
        let myValue = 1

namespace MyCompany.MyOtherLibrary.Collections

    type MyCollection(x : int) =
      member v.P = x
```

Namespace declaration groups may not be nested.

A namespace declaration group can contain type and module definitions, but not function or value definitions. For example:

```
namespace MyCompany.MyLibrary
```

```
   // A type definition in a namespace
   type MyType() =
      let x = 1
       member v.P = x+2

   // A module definition in a namespace
   module MyInnerModule =
        let myValue = 1

   // The following is not allowed: value definitions are not allowed
in namespaces
   let addOne x = x + 1
```

When a namespace declaration group N is checked in an environment env, the individual declarations are checked in order and an overall *namespace declaration group signature* N_{sig} is inferred for the module. An entry for N is then added to the *ModulesAndNamespaces* table in the environment env (see §14.1.3).

Like module declarations, namespace declaration groups are processed sequentially rather than simultaneously, so that later namespace declaration groups are not in scope when earlier ones are processed. This prevents invalid recursive definitions.

In the following example, the declaration of `x` in `Module1` generates an error because the `Utilities.Part2` namespace is not in scope:

```
namespace Utilities.Part1

    module Module1 =
        let x = Utilities.Part2.Module2.x + 1  // error (Part2 not yet
declared)

namespace Utilities.Part2

    module Module2 =
        let x = Utilities.Part1.Module1.x + 2
```

Within a namespace declaration group, the namespace itself is implicitly opened if any preceding namespace declaration groups or referenced assemblies contribute to it. For example:

```
namespace MyCompany.MyLibrary

   module Values1 =
      let x = 1

namespace MyCompany.MyLibrary

   // Here, the implicit open of MyCompany.MyLibrary brings Values1
into scope

   module Values2 =
```

```
let x = Values1.x
```

10.2 Module Definitions

A module definition is a named collection of declarations such as values, types, and function values. Grouping code in modules helps keep related code together and helps avoid name conflicts in your program. For example:

```
module MyModule =
    let x = 1
    type Foo = A | B
    module MyNestedModule =
        let f y = y + 1
        type Bar = C | D
```

When a module definition M is checked in an environment env_0, the individual declarations are checked in order and an overall *module signature* M_{sig} is inferred for the module. An entry for M is then added to the *ModulesAndNamespaces* table to environment env_0 to form the new environment used for checking subsequent modules.

Like namespace declaration groups, module definitions are processed sequentially rather than simultaneously, so that later modules are not in scope when earlier ones are processed.

```
module Part1 =

   let x = Part2.StorageCache()  // error (Part2 not yet declared)

module Part2 =

   type StorageCache() =
      member cache.Clear() = ()
```

No two types or modules may have identical names in the same namespace. The `[<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix)>]` attribute adds the suffix `Module` to the name of a module to distinguish the module name from a type of a similar name.

For example, this is frequently used when defining a type and a set of functions and values to manipulate values of this type.

```
type Cat(kind: string) =
   member x.Meow() = printfn "meow"
   member x.Purr() = printfn "purr"
   member x.Kind = kind

[<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix
)>]
module Cat =
```

```
    let tabby = Cat "Tabby"
    let purr (c:Cat) = c.Purr()
    let purrTwice (c:Cat) = purr(); purr()

Cat.tabby |> Cat.purr |> Cat.purrTwice
```

10.2.1 Function and Value Definitions in Modules

Function and value definitionsin modules introduce named values and functions.

let rec*opt* *function-or-value-defn₁* and ... and *function-or-value-defnₙ*

The following example defines value `x` and functions `id` and `fib`:

```
module M =
    let x = 1
    let id x = x
    let rec fib x = if x <= 2 then 1 else fib (n - 1) + fib (n - 2)
```

Function and value definitions in modules may declare explicit type variables and type constraints:

```
    let pair<'T>(x : 'T) = (x, x)
    let dispose<'T when 'T :> System.IDisposable>(x : 'T) = x.Dispose()
    let convert<'T, 'U>(x) = unbox<'U>(box<'T>(x))
```

A value definition that has explicit type variables is called a type function (§10.2.3).

Function and value definitions may specify attributes:

```
    // A value definition with the System.Obsolete attribute
    [<System.Obsolete("Don't use this")>]
    let oneTwoPair = (1, 2)

    // A function definition with an attribute
    [<System.Obsolete("Don't use this either")>]
    let pear v = (v, v)
```

By the use of pattern matching, a value definition can define more than one value . In such cases, the attributes apply to each value.

```
    // A value definition that defines two values, each with an
attribute
    [<System.Obsolete("Don't use this")>]
    let (a, b) = (1, 2)
```

Values may be declared mutable:

```
    // A value definition that defines a mutable value
    let mutable count = 1
    let freshName() = (count <- count + 1; count)
```

Function and value definitions in modules are processed in the same way as function and value definitions in expressions (§14.6), with the following adjustments:

- Each defined value may have an accessibility annotation (§10.5). By default, the accessibility annotation of a function or value definition in a module is *public*.
- Each defined value is *externally accessible* if its accessibility annotation is `public` and it is not hidden by an explicit signature. Externally accessible values are guaranteed to have compiled CLI representations in compiled CLI binaries.
- Each defined value can be used to satisfy the requirements of any signature for the module (§11.2).
- Each defined value is subject to arity analysis (§14.10).
- Values may have attributes, including the `ThreadStatic` or `ContextStatic` attribute.

10.2.2 Literal Definitions in Modules

Value definitions in modules may have the `Literal` attribute. This attribute causes the value to be compiled as a constant. For example:

```
[<Literal>]
let PI = 3.141592654
```

Literal values may be used in custom attributes and pattern matching. For example:

```
[<Literal>]
let StartOfWeek = System.DayOfWeek.Monday

[<MyAttribute(StartOfWeek)>]
let feeling(day) =
    match day with
    | StartOfWeek -> "rough"
    | _ -> "great"
```

A value that has the `Literal` attribute is subject to the following restrictions:

- It may not be marked `mutable` or `inline.`
- It may not also have the `ThreadStatic` or `ContextStatic` attributes.
- The right-hand side expression must be a *literal constant expression* that is both a valid expression after checking, and is made up of either:
  - A simple constant expression, with the exception of (), native integer literals, unsigned native integer literals, byte array literals, BigInteger literals, and user-defined numeric literals.

  —OR—

  - A reference to another literal

  —OR—

- A bitwise combination of literal constant expressions

—OR—

- A “+” concatenation of two literal constant expressions which are strings

—OR—

- “enum x” or “LanguagePrimitives.EnumOfValue x” where “x” is a literal constant expression.

10.2.3 Type Function Definitions in Modules

Value definitions within modules may have explicit generic parameters. For example, `'T` is a generic parameter to the value `empty`:

```
let empty<'T> : (list<'T> * Set<'T>) = ([], Set.empty)
```

A value that has explicit generic parameters but has arity `[]` (that is, no explicit function parameters) is called a *type function*. The following are some example type functions from the F# library:

```
val typeof<'T> : System.Type
val sizeof<'T> : int
module Set =
    val empty<'T> : Set<'T>
module Map =
    val empty<'Key,'Value> : Map<'Key,'Value>
```

Type functions are rarely used in F# programming, although they are convenient in certain situations. Type functions are typically used for:

- Pure functions that compute type-specific information based on the supplied type arguments.
- Pure functions whose result is independent of inferred type arguments, such as empty sets and maps.

Type functions receive special treatment during generalization (§14.6.7) and signature conformance (§11.2). They typically have either the `RequiresExplicitTypeArguments` attribute or the `GeneralizableValue` attribute. Type functions may not be defined inside types, expressions, or computation expressions.

In general, type functions should be used only for computations that do not have observable side effects. However, type functions may still perform computations. In this example, `r` is a type function that calculates the number of times it has been called

```
let mutable count = 1
let r<'T> = (count <- count + 1); ref ([] : 'T list);;
// count = 1
let x1 = r<int>
// count = 2
let x2 = r<int>
// count = 3
let z0 =  x1
// count = 3
```

The elaborated form of a type function is that of a function definition that takes one argument of type `unit`. That is, the elaborated form of

```
let ident typar-defns  = expr
```

is the same as the compiled form for the following declaration:

```
let ident typar-defns () = expr
```

References to type functions are elaborated to invocations of such a function.

10.2.4 Active Pattern Definitions in Modules

A value definition within a module that has an *`active-pattern-op-name`* introduces pattern-matching tags into the environment when the module is accessed or opened. For example,

```
let (|A|B|C|) x = if x < 0 then A elif x = 0 then B else C
```

introduces pattern tags `A`, `B`, and `C` into the *PatItems* table in the name resolution environment.

10.2.5 “do” statements in Modules

A “do” statement within a module has the following form:

```
do expr
```

The expression *`expr`* is checked with an arbitrary initial type *`ty`*. After checking *`expr`*, *`ty`* is asserted to be equal to `unit`. If the assertion fails, a warning rather than an error is reported. This warning is suppressed for plain expressions without `do` in script files (that is, `.fsx` and `.fsscript` files).

A “do” statement may have attributes. In this example, the `STAThread` attribute specifies that `main` uses the single-threaded apartment (STA) threading model of COM:

```
let main() =
    let form = new System.Windows.Forms.Form()
    System.Windows.Forms.Application.Run(form)

[<STAThread>]
do main()
```

10.3 Import Declarations

Namespace declaration groups and module definitions can include *import declarations* in the following form:

```
open long-ident
```

Import declarations make elements of other namespace declaration groups and modules accessible by the use of unqualified names. For example:

```
open FSharp.Collections
```

```
open System
```

Import declarations can be used in:

- Module definitions and their signatures.
- Namespace declaration groups and their signatures.

An import declaration is processed by first resolving the `long-ident` to one or more namespace declaration groups and/or modules [F_1, ..., F_n] by *Name Resolution in Module and Namespace Paths* (§14.1.2). For example, `System.Collections.Generic` may resolve to one or more namespace declaration groups—one for each assembly that contributes a namespace declaration group in the current environment. Next, each F_i is added to the environment successively by using the technique specified in §14.1.3. An error occurs if any F_i is a module that has the `RequireQualifiedAccess` attribute.

10.4 Module Abbreviations

A module abbreviation defines a local name for a module long identifier, as follows:

```
module ident = long-ident
```

For example:

```
module Ops = FSharp.Core.Operators
```

Module abbreviations can be used in:

- Module definitions and their signatures.
- Namespace declaration groups and their signatures.

Module abbreviations are implicitly private to the module or namespace declaration group in which they appear.

A module abbreviation is processed by first resolving the `long-ident` to a list of modules by *Name Resolution in Module and Namespace Paths* (see §14.1). The list is then appended to the set of names that are associated with `ident` in the *ModulesAndNamespaces* table.

Module abbreviations may not be used to abbreviate namespaces.

10.5 Accessibility Annotations

Accessibilities may be specified on declaration elements in namespace declaration groups and modules, and on members in types. The table lists the accessibilities that can appear in user code:

Accessibility	Description
`public`	No restrictions on access.
`private`	Access is permitted only from the enclosing type, module, or namespace declaration group.

Accessibility	Description
`internal`	Access is permitted only from within the enclosing assembly, or from assemblies whose name is listed using the `InternalsVisibleTo` attribute in the current assembly.

The default accessibilities are public. Specifically:

- Function definitions, value definitions, type definitions, and exception definitions in modules are public.
- Modules, type definitions, and exception definitions in namespaces are public.
- Members in type definitions are public.

Some function and value definitions may not be given an accessibility and, by their nature, have restricted lexical scope. In particular:

- Function and value definitions in classes are lexically available only within the class being defined, and only from the point of their definition onward.
- Module type abbreviations are lexically available only within the module or namespace declaration group being defined, and only from their point of their definition onward.

Note that:

- `private` on a member means “private to the enclosing type or module.”
- `private` on a function or value definition in a module means “private to the module or namespace declaration group.”
- `private` on a type, module, or type representation in a module means “private to the module.”

The CLI compiled form of all non-public entities is `internal`.

> Note: The `family` and `protected` specifications are not supported in this version of the F# language.

Accessibility modifiers can appear only in the locations summarized in the following table.

Component	Location	Example

Function or value definition in module	Precedes identifier	`let private x = 1`
		`let inline private f x = 1`
		`let mutable private x = 1`

Module definition	Precedes identifier	`module private M =` `    let x = 1`
Type definition	Precedes identifier	`type private C = A | B` `type private C<'T> = A | B`
`val` definition in a class	Precedes identifier	`val private x : int`
Explicit constructor	Precedes identifier	`private new () = { inherit Base }`
Implicit constructor	Precedes identifier	`type C private() = ...`

Component	Location	Example
Member definition	Precedes identifier, but cannot appear on: ▪ `inherit` definitions ▪ `interface` definitions ▪ `abstract` definitions ▪ Individual union cases Accessibility for `inherit`, `interface`, and `abstract` definitions is always the same as that of the enclosing class.	`member private x.X = 1`

Explicit property get or set in a class	Precedes identifier	`member __.Item`
		`    with private get i = 1`
		`    and  private set i v = ()`

Type representation	Precedes identifier	`type Cases =`
		`    private`
		`        | A`
		`        | B`

11. Namespace and Module Signatures

A signature file contains one or more namespace or module signatures, and specifies the functionality that is implemented by its corresponding implementation file. It also can hide functionality that the corresponding implementation file contains.

```
namespace-decl-group-signature :=
    namespace long-ident module-signature-elements

module-signature =
    module ident = module-signature-body

module-signature-element :=
    val mutableopt curried-sig       -- value signature
    val value-defn           -- literal value signature
    type type-signatures             -- type(s) signature
    exception exception-signature  -- exception signature
    module-signature                 -- submodule signature
    module-abbrev            -- local alias for a module
    import-decl              -- locally import contents of a module

module-signature-elements := module-signature-element ... module-
signature-element

module-signature-body =
    begin module-signature-elements end

type-signature :=
    abbrev-type-signature
    record-type-signature
    union-type-signature
    anon-type-signature
    class-type-signature
    struct-type-signature
    interface-type-signature
    enum-type-signature
    delegate-type-signature
    type-extension-signature

type-signatures := type-signature ... and ... type-signature

type-signature-element :=
    attributesopt accessopt new : uncurried-sig      -- constructor
signature
    attributesopt member  accessopt member-sig       -- member signature
    attributesopt abstract accessopt member-sig      -- member signature
    attributesopt override member-sig                -- member signature
```

```
    attributes_opt default member-sig                      -- member
signature
    attributes_opt static member access_opt member-sig        -- static
member signature
    interface type                              -- interface signature

abbrev-type-signature := type-name '=' type

union-type-signature := type-name '=' union-type-cases type-
extension-elements-signature_opt

record-type-signature := type-name '=' '{' record-fields '}' type-
extension-elements-signature_opt

anon-type-signature := type-name '=' begin type-elements-signature
end

class-type-signature := type-name '=' class type-elements-signature
end

struct-type-signature := type-name '=' struct type-elements-signature
end

interface-type-signature := type-name '=' interface type-elements-
signature end

enum-type-signature := type-name '=' enum-type-cases

delegate-type-signature :=  type-name '=' delegate-sig

type-extension-signature := type-name type-extension-elements-
signature

type-extension-elements-signature := with type-elements-signature end
```

The begin and end tokens are optional when lightweight syntax is used.

Like module declarations, signature declarations are processed sequentially rather than simultaneously, so that later signature declarations are not in scope when earlier ones are processed.

```
namespace Utilities.Part1

    module Module1 =
        val x : Utilities.Part2.StorageCache  // error (Part2 not yet
declared)

namespace Utilities.Part2

   type StorageCache =
      new : unit -> unit
```

11.1 Signature Elements

A namespace or module signature declares one or more *value signatures* and one or more *type definition signatures*. A type definition signature may include one or more *member signatures*, in addition to other elements of type definitions that are specified in the signature grammar at the start of this chapter.

11.1.1 Value Signatures

A value signature indicates that a value exists in the implementation. For example, in the signature of a module, the following declares two value signatures:

```
module MyMap =
    val mapForward : index1: int * index2: int -> string
    val mapBackward : name: string -> (int * int)
```

The corresponding implementation file might contain the following implementation:

```
module MyMap =
    let mapForward (index1:int, index2:int) = string index1 + "," +
string index2
    let mapBackward (name:string) = (0, 0)
```

11.1.2 Type Definition and Member Signatures

A type definition signature indicates that a corresponding type definition appears in the implementation. For example, in an interface type, the following declares a type definition signature for `Forward` and `Backward`:

```
type IMap =
    interface
        abstract Forward : index1: int * index2: int -> string
        abstract Backward : name: string -> (int * int)
    end
```

A member signature indicates that a corresponding member appears on the corresponding type definition in the implementation. Member specifications must specify argument and return types, and can optionally specify names and attributes for parameters.

For example, the following declares a type definition signature for a type with one constructor member, one property member `Kind` and one method member `Purr`:

```
type Cat =
    new : kind:string -> Cat
    member Kind : string
    member Purr : unit -> Cat
```

The corresponding implementation file might contain the following implementation:

```
type Cat(kind: string) =
   member x.Meow() = printfn "meow"
   member x.Purr() = printfn "purr"
```

```
member x.Kind = kind
```

11.2 Signature Conformance

Values, types, and members that are present in implementations can be omitted in signatures, with the following exceptions:

- Type abbreviations may not be hidden by signatures. That is, a type abbreviation `type T = ty` in an implementation does not match `type T` (without an abbreviation) in a signature.
- Any type that is represented as a record or union must reveal either all or none of its fields or cases, in the same order as that specified in the implementation. Types that are represented as classes may reveal some, all, or none of their fields in a signature.
- Any type that is revealed to be an interface, or a type that is a class or struct with one or more constructors may not hide its `inherit` declaration, abstract dispatch slot declarations, or abstract interface declarations.

> Note: This section does not yet document all checks made by the F# 3.1 language implementation.

11.2.1 Signature Conformance for Functions and Values

If both a signature and an implementation contain a function or value definition with a given name, the signature and implementation must conform as follows:

- The declared accessibilities, `inline`, and `mutable` modifiers must be identical in both the signature and the implementation.
- If either the signature or the implementation has the `[<Literal>]` attribute, both must have this attribute. Furthermore, the declared literal values must be identical.
- The number of generic parameters—both inferred and explicit—must be identical.
- The types and type constraints must be identical up to renaming of inferred and/or explicit generic parameters. For example, assume a signature is written “`val head : seq<'T> -> 'T`” and the compiler could infer the type “`val head : seq<'a> -> 'a`” from the implementation. These are considered identical up to renaming the generic parameters.
- The arities must match, as described in the next section.

11.2.1.1 Arity Conformance for Functions and Values

Arities of functions and values must conform between implementation and signature. Arities of values are implicit in module signatures. A signature that contains the following results in the arity $[A_1 ... A_n]$ for `F`:

$$\texttt{val F :}\ ty_{1,1}\ \texttt{*}\ \ldots\ \texttt{*}\ ty_{1,A1}\ \texttt{->}\ \ldots\ \texttt{->}\ ty_{n,1}\ \texttt{*}\ \ldots\ \texttt{*}\ ty_{n,An}\ \texttt{->}\ rty$$

Arities in a signature must be equal to or shorter than the corresponding arities in an implementation, and the prefix must match. This means that F# makes a deliberate distinction between the following two signatures:

```
val F: int -> int
```

and

```
val F: (int -> int)
```

The parentheses indicate a top-level function, which might be a first-class computed expression that computes to a function value, rather than a compile-time function value.

The first signature can be satisfied only by a true function; that is, the implementation must be a lambda value as in the following:

```
let F x = x + 1
```

> Note: Because arity inference also permits right-hand-side function expressions, the implementation may currently also be:
>
> ```
> let F = fun x -> x + 1
> ```

The second signature

```
val F: (int -> int)
```

can be satisfied by any value of the appropriate type. For example:

```
let f =
    let myTable = new System.Collections.Generic.Dictionary<int,int>(4)
    fun x ->
        if myTable.ContainsKey x then
            myTable.[x]
        else
            let res = x * x
            myTable.[x] <- res
            res
```

—or—

```
let f = fun x -> x + 1
```

—or—

```
// throw an exception as soon as the module initialization is triggered
let f : int -> int = failwith "failure"
```

For both the first and second signatures, you can still use the functions as first-class function values from client code—the parentheses simply act as a constraint on the implementation of the value.

The reason for this interpretation of types in value and member signatures is that CLI interoperability requires that F# functions compile to methods, rather than to fields that are function values. Thus, signatures must contain enough information to reveal the desired arity of a method as it is revealed to other CLI programming languages.

11.2.1.2 Signature Conformance for Type Functions

If a value is a type function, then its corresponding value signature must have explicit type arguments. For example, the implementation

```
let empty<'T> : list<'T> = printfn "hello"; []
```

conforms to this signature:

```
val empty<'T> : list<'T>
```

but not to this signature:

```
val empty : list<'T>
```

The reason for this rule is that the second signature indicates that the value is, by default, generalizable (§14.6.7).

11.2.2 Signature Conformance for Members

If both a signature and an implementation contain a member with a given name, the signature and implementation must conform as follows:

- If one is an extension member, both must be extension members.
- If one is a constructor, then both must be constructors.
- If one is a property, then both must be properties.
- The types must be identical up to renaming of inferred or explicit type parameters (as for functions and values).
- The `static`, `abstract`, and `override` qualifiers must match precisely.
- Abstract members must be present in the signature if a representation is given for a type.

> Note: This section does not yet document all checks made by the F# 3.1 language implementation.

12. Program Structure and Execution

F# programs are made up of a collection of assemblies. F# assemblies are made up of static references to existing assemblies, called the *referenced assemblies*, and an interspersed sequence of signature (`.fsi`) files, implementation (`.fs`) files, script (`.fsx` or `.fsscript`) files, and interactively executed code fragments.

```
implementation-file :=
    namespace-decl-group ... namespace-decl-group
    named-module
    anonynmous-module

script-file := implementation-file
                    -- script file, additional directives allowed

signature-file:=
    namespace-decl-group-signature ... namespace-decl-group-signature
    anonynmous-module-signature
    named-module-signature

named-module :=
    module long-ident module-elems

anonymous-module :=
    module-elems

named-module-signature :=
    module long-ident module-signature-elements

anonymous-module-signature :=
    module-signature-elements

script-fragment :=
    module-elems    -- interactively entered code fragment
```

A sequence of implementation and signature files is checked as follows.

1. Form an initial environment $sig\text{-}env_0$ and $impl\text{-}env_0$ by adding all assembly references to the environment in the order in which they are supplied to the compiler. This means that the following procedure is applied for each referenced assembly:

   - Add the top level types, modules, and namespaces to the environment.
   - For each `AutoOpen` attribute in the assembly, find the types, modules, and namespaces that the attribute references and add these to the environment.

   The resulting environment becomes the active environment for the first file to be processed.

2. For each file:

- If the i^{th} file is a signature file `file.fsi`:

  a. Check it against the current signature environment $sig\text{-}env_{i-1}$, which generates the signature Sig_{file} for the current file.

  b. Add Sig_{file} to $sig\text{-}env_{i-1}$ to produce $sig\text{-}env_i$ to make it available for use in later signature files.

  The processing of the signature file has no effect on the implementation environment, so $impl\text{-}env_i$ is identical to $impl\text{-}env_{i-1}$.

- If the file is an implementation file `file.fs`, check it against the environment $impl\text{-}env_{i-1}$, which gives elaborated namespace declaration groups $Impl_{file}$.

  a. If a corresponding signature Sig_{file} exists, check $Impl_{file}$ against Sig_{file} during this process (§11.2). Then add Sig_{file} to $impl\text{-}env_{i-1}$ to produce $impl\text{-}env_i$. This step makes the signature-constrained view of the implementation file available for use in later implementation files. The processing of the implementation file has no effect on the signature environment, so $sig\text{-}env_i$ is identical to $sig\text{-}env_{i-1}$.

  b. If the implementation file has no signature file, add $Impl_{file}$ to both $sig\text{-}env_{i-1}$ and $impl\text{-}env_{i-1}$, to produce $sig\text{-}env_i$ and $impl\text{-}env_i$. This makes the contents of the implementation available for use in both later signature and implementation files.

The signature file for a particular implementation must occur before the implementation file in the compilation order. For every signature file, a corresponding implementation file must occur after the file in the compilation order. Script files may not have signatures.

12.1 Implementation Files

Implementation files consist of one or more namespace declaration groups. For example:

```
namespace MyCompany.MyOtherLibrary

    type MyType() =
      let x = 1
      member v.P = x + 2

   module MyInnerModule =
        let myValue = 1

namespace MyCompany. MyOtherLibrary.Collections

    type MyCollection(x : int) =
        member v.P = x
```

An implementation file that begins with a `module` declaration defines a single namespace declaration group with one module. For example:

```
module MyCompany.MyLibrary.MyModule

let x = 1
```

is equivalent to:

```
namespace MyCompany.MyLibrary

module MyModule =
    let x = 1
```

The final identifier in the *long-ident* that follows the `module` keyword is interpreted as the module name, and the preceding identifiers are interpreted as the namespace.

Anonymous implementation files do not have either a leading `module` or `namespace` declaration. Only the scripts and the last file within an implementation group for an executable image (.exe) may be anonymous. An anonymous implementation file contains module definitions that are implicitly placed in a module. The name of the module is generated from the name of the source file by capitalizing the first letter and removing the filename extensionIf the filename contains characters that are not valid in an F# identifier, the resulting module name is unusable and a warning occurs.

Given an initial environment env_0, an implementation file is checked as follows:

- Create a new constraint solving context.
- Check the namespace declaration groups in the file against the existing environment env_{i-1} and incrementally add them to the environment (§10.1) to create a new environment env_i.
- Apply default solutions to any remaining type inference variables that include `default` constraints. The defaults are applied in the order that the type variables appear in the type-annotated text of the checked namespace declaration groups.
- Check the inferred signature of the implementation file against any required signature by using *Signature Conformance* (§11.2). The resulting signature of an implementation file is the required signature, if it is present; otherwise it is the inferred signature.
- Report a “value restriction” error if the resulting signature of any item that is not a member, constructor, function, or type function contains any free inference type variables.
- Choose solutions for any remaining type inference variables in the elaborated form of an expression. Process any remaining type variables in the elaborated form from left-to-right to find a minimal type solution that is consistent with constraints on the type variable. If no unique minimal solution exists for a type variable, report an error.

The result of checking an implementation file is a set of elaborated namespace declaration groups.

12.2 Signature Files

Signature files specify the functionality that is implemented by a corresponding implementation file. Each signature file contains a sequence of *namespace-decl-group-signature* elements. The inclusion of a signature file in compilation implicitly applies that signature type to the contents of a corresponding implementation file.

Anonymous signature files do not have either a leading `module` or `namespace` declaration. Anonymous signature files contain *`module-elems`* that are implicitly placed in a module. The name of the module is generated from the name of the source file by capitalizing the first letter and removing the filename extension. If the filename contains characters that are not valid in an F# identifier, the resulting module name is unusable and a warning occurs.

Given an initial environment *`env`*, a signature file is checked as follows:

- Create a new constraint solving context.
- Check each *`namespace-decl-group-signature`*$_i$ in *`env`*$_{i-1}$ and add the result to that environment to create a new environment *`env`*$_i$.

The result of checking a signature file is a set of elaborated namespace declaration group types.

12.3 Script Files

Script files have the `.fsx` or `.fsscript` filename extension. They are processed in the same way as files that have the `.fs` extension, with the following exceptions:

- Side effects from all scripts are executed at program startup.
- For script files, the namespace `FSharp.Compiler.Interactive.Settings` is opened by default.
- F# Interactive references the assembly `FSharp.Compiler.Interactive.Settings.dll` by default, but the F# compiler does not. If the script uses the script helper `fsi` object, then the script should explicitly reference `FSharp.Compiler.Interactive.Settings.dll`.

Script files may add to the set of referenced assemblies by using the `#r` directive (§12.312.4).

Script files may add other signature, implementation, and script files to the list of sources by using the `#load` directive. Files are compiled in the same order that was passed to the compiler, except that each script is searched for `#load` directives and the loaded files are placed before the script, in the order they appear in the script. If a filename appears in more than one `#load` directive, the file is placed in the list only once, at the position it first appeared.

Script files may have `#nowarn` directives, which disable a warning for the entire compilation.

The F# compiler defines the `COMPILED` compilation symbol for input files that it has processed. F# Interactive defines the `INTERACTIVE` symbol.

Script files may not have corresponding signature files.

12.4 Compiler Directives

Compiler directives are declarations in non-nested modules or namespace declaration groups in the following form:

```
# id string ... string
```

The lexical preprocessor directives `#if`, `#else`, `#endif` and `#indent "off"` are similar to compiler directives. For details on `#if`, `#else`, `#endif`, see §3.3. The `#indent "off"` directive is described in §19.4.

The following directives are valid in all files:

Directive	Example	Short Description
`#nowarn`	`#nowarn "54"`	For signature (`.fsi`) files and implementation (`.fs`) files, turns off warnings within this lexical scope. For script (`.fsx` or `.fsscript`) files, turns off warnings globally.

The following directives are valid in script files:

Directive	Example	Short Description

`#r`
`#reference`
	`#r "System.Core"`
	`#r @"Nunit.Core.dll"`
	`#r @"c:\NUnit\Nunit.Core.dll"`
	`#r "nunit.core, Version=2.2.2.0, Culture=neutral, PublicKeyToken=96d09a1eb7f44a77"`
		References a DLL within this entire script.

`#I` `#Include`	`#I @"c:\Projects\Libraries\Bin"`	Adds a path to the search paths for DLLs that are referenced within this entire script.
`#load`	`#load "library.fs"` `#load "core.fsi" "core.fs"`	Loads a set of signature and implementation files into the script execution engine.

`#time`	`#time`
	`#time "on"`
	`#time "off"`
		Enables or disables the display of performance information, including elapsed real time, CPU time, and garbage collection information for each section of code that is interpreted and executed.

`#help`	`#help`	Asks the script execution environment for help.
`#q` `#quit`	`#q` `#quit`	Requests the script execution environment to halt execution and exit.

12.5 Program Execution

Execution of F# code occurs in the context of an executing CLI program into which one or more compiled F# assemblies or script fragments is loaded. During execution, the CLI program can use the functions, values, static members, and object constructors that the assemblies and script fragments define.

12.5.1 Execution of Static Initializers

Each implementation file, script file, and script fragment involves a *static initializer*. The execution of the static initializer is triggered as follows:

- For executable (.exe) files that have an explicit entry point function, the static initializer for the last file that appears on the command line is forced immediately as the first action in the execution of the entry point function.
- For executable files that have an implicit entry point, the static initializer for the last file that appears on the command line is the body of the implicit entry point function.
- For scripts, F# Interactive executes the static initializer for each program fragment immediately.
- For all other implementation files, the static initializer for the file is executed on first access of a value that has observable initialization according to the rules that follow, or first access to any member of any type in the file that has at least one “static let” or “static do” declaration.

At runtime, the static initializer evaluates, in order, the definitions in the file that have observable initialization according to the rules that follow. Definitions with observable initialization in nested modules and types are included in the static initializer for the overall file.

All definitions have observable initialization except for the following definitions in modules:

- Function definitions
- Type function definitions
- Literal definitions
- Value definitions that are generalized to have one or more type variables
- Non-mutable, non-thread-local values that are bound to an *initialization constant expression*, which is an expression whose elaborated form is one of the following:
  - A simple constant expression.
  - A null expression.
  - A use of the `typeof<_>` or `sizeof<_>` operator from `FSharp.Core.Operators`, or the `defaultof<_>` operator from `FSharp.Core.Operators.Unchecked`.
  - A let expression where the constituent expressions are initialization constant expressions.
  - A match expression where the input is an initialization constant expression, each case is a test against a constant, and each target is an initialization constant expression.
  - A use of one of the unary or binary operators `=`, `<>`, `<`, `>`, `<=`, `>=`, `+`, `-`, `*` , `<<<`, `>>>`, `|||`, `&&&`, `^^^`, `~~~`, `enum<_>`, `not`, `compare`, prefix `–`, and prefix `+` from `FSharp.Core.Operators` on one or two arguments, respectively. The arguments themselves must be initialization constant expressions, but cannot be operations on decimals or strings. Note that the operators are unchecked for arithmetic operations, and that the operators `%` and `/` are not included because their use can raise division-by-zero exceptions.
  - A use of a `[<Literal>]` value.
  - A use of a case from an enumeration type.
  - A use of a null case from a union type.
  - A use of a value that is defined in the same assembly and does not have observable initialization, or the use of a value that is defined by a “let” or “match” expression within the expression itself.

If the execution environment supports the concurrent execution of multiple threads of F# code, each static initializer runs as a mutual exclusion region. The use of a mutual exclusion region ensures that if another thread attempts to access a value that has observable initialization, that thread pauses until static initialization is complete. A static initializer runs only once, on the first thread that acquires entry to the mutual exclusion region.

Values that have observable initialization have implied CLI fields that are private to the assembly. If such a field is accessed by using CLI reflection before the execution of the corresponding initialization code, then the default value for the type of the field will be returned.

Within implementation files, generic types that have static value definitions receive a static initializer for each generic instantiation. These initializers are executed immediately before the first dereference of the static fields for the generic type, subject to any limitations present in the specific CLI implementation in used. If the static initializer for the enclosing file is first triggered during execution of the static initializer for a generic instantiation, references to static values definition in the generic class evaluate to the default value.

For example, if external code accesses `data` in this example, the static initializer runs and the program prints "hello":

```
module LibraryModule
printfn "hello"
let data = new Dictionary<int,int>()
```

That is, the side effect of printing "hello" is guaranteed to be triggered by an access to the value `data`.

If external code calls `id` or accesses `size` in the following example, the execution of the static initializer is not yet triggered. However if external code calls `f()`,the execution of the static initializer is triggered because the body refers to the value `data`, which has observable initialization.

```
module LibraryModule
printfn "hello"
let data = new Dictionary<int,int>()
let size = 3
let id x = x
let f() = data
```

All of the following represent definitions that do not have observable initialization because they are initialization constant expressions.

```
let x = System.DayOfWeek.Friday
let x = 1.0
let x = "two"
let x = enum<System.DayOfWeek>(0)
let x = 1 + 1
let x : int list = []
let x : int option = None
let x = compare 1 1
let x = match true with true -> 1 | false -> 2
let x = true && true
let x = 42 >>> 2
let x = typeof<int>
let x = Unchecked.defaultof<int>
let x = Unchecked.defaultof<string>
let x = sizeof<int>
```

12.5.2 Explicit Entry Point

The last file that is specified in the compilation order for an executable file may contain an explicit entry point. The entry point is indicated by annotating a function in a module with `EntryPoint` attribute:

- The `EntryPoint` attribute applies only to a “let”-bound function in a module. The function cannot be a member.
- This attribute can apply to only one function, and the function must be the last declaration in the last file processed on the command line. The function may be in a nested module.
- The function is asserted to have type `string[] -> int` before type checking. If the assertion fails, an error occurs.
- At runtime, the entry point is passed one argument at startup: an array that contains the same entries as `System.Environment.GetCommandLineArgs()`, minus the first entry in that array.

The function becomes the entry point to the program. At startup, F# immediately forces execution of the static initializer for the file in which the function is declared, and then evaluates the body of the function.

13. Custom Attributes and Reflection

CLI languages use metadata inspection and the `System.Reflection` libraries to make guarantees about how compiled entities appear at runtime. They also allow entities to be attributed by static data, and these attributes may be accessed and read by tools and running programs. This chapter describes these mechanisms for F#.

Attributes are given by the following grammar:

```
attribute := attribute-target:opt object-construction

attribute-set := [< attribute ; ... ; attribute >]

attributes := attribute-set ... attribute-set

attribute-target :=
    assembly
    module
    return
    field
    property
    param
    type
    constructor
    event
```

13.1 Custom Attributes

CLI languages support the notion of *custom attributes* which can be added to most declarations. These are added to the corresponding elaborated and compiled forms of the constructs to which they apply.

Custom attributes can be applied only to certain target language constructs according to the `AttributeUsage` attribute, which is found on the attribute class itself. An error occurs if an attribute is attached to a language construct that does not allow that attribute.

Custom attributes are not permitted on function or value definitions in expressions or computation expressions. Attributes on parameters are given as follows:

```
let foo([<SomeAttribute>] a) = a + 5
```

If present, the arguments to a custom attribute must be *literal constant expressions*, or arrays of the same.

Custom attributes on return values are given as follows:

```
let foo a : [<SomeAttribute>] = a + 5
```

Custom attributes on primary constructors are given before the arguments and before any accessibility annotation:

```
type Foo1 [<System.Obsolete("don't use me")>] () =
    member x.Bar() = 1

type Foo2 [<System.Obsolete("don't use me")>] private () =
    member x.Bar() = 1
```

Custom attributes are mapped to compiled CLI metadata as follows:

- Custom attributes map to the element that is specified by their target, if a target is given.
- A custom attribute on a type *type* is compiled to a custom attribute on the corresponding CLI type definition, whose `System.Type` object is returned by `typeof<type>`.
- By default, a custom attribute on a record field F for a type T is compiled to a custom attribute on the CLI property for the fieldthat is named F, unless the target of the attribute is `field`, in which case it becomes a custom attribute on the underlying backing field for the CLI property that is named _F.
- A custom attribute on a union case ABC for a type T is compiled to a custom attribute on a static method on the CLI type definition T. This method is called:
  - get_ABC if the union case takes no arguments
  - ABC otherwise
- Custom attributes on arguments are propagated only for arguments of member definitions, and not for “let”-bound function definitions.
- Custom attributes on generic parameters are not propagated.

Custom attributes that appear immediately preceding “do” statements in modules anywhere in an assembly are attached to one of the following:

- The `main` entry point of the program.
- The compiled module.
- The compiled assembly.

Custom attributes are attached to the main entry point if it is valid for them to be attached to a method according to the `AttributeUsage` attribute that is found on the attribute class itself, and likewise for the assembly. If it is valid for the attribute to be attached to either the main method or the assembly. the main method takes precedence.

For example, the STAThread attribute should be placed immediately before a top-level “do” statement.

```
let main() =
    let form = new System.Windows.Forms.Form()
    System.Windows.Forms.Application.Run(form)
```

```
[<STAThread>]
do main()
```

13.1.1 Custom Attributes and Signatures

During signature checking, custom attributes attached to items in F# signature files (`.fsi` files) are combined with custom attributes on the corresponding element from the implementation file according to the following algorithm:

- Start with lists *`AImpl`* and *`ASig`* containing the attributes in the implementation and signature, in declaration order.
- Check each attribute in *`AImpl`* against the available attributes in *`ASig`*.
- If *`ASig`* contains an attribute that is an exact match after evaluating attribute arguments, then ignore the attribute in the implementation, remove the attribute from *`ASig`*, and continue checking;
- If *`ASig`* contains an attribute that has the same attribute type but is not an exact match, then give a warning and ignore the attribute in the implementation;
- Otherwise, keep the attribute in the implementation.

The compiled element contains the compiled forms of the attributes from the signature and the retained attributes from the implementation.

This means:

- When an implementation has an attribute `X("abc")` and the signature is missing the attribute, then no warning is given and the attribute appears in the compiled assembly.
- When a signature has an attribute `X("abc")` and the implementation is missing the attribute, then no warning is given, and the attribute appears in the compiled assembly.
- When an implementation has an attribute `X("abc")` and the signature has attribute `X("def")`, then a warning is given, and only `X("def")` appears in the compiled assembly.

13.2 Reflected Forms of Declaration Elements

The `typeof` and `typedefof` F# library operators return a `System.Type` object for an F# type definition. According to typical implementations of the CLI execution environment, the `System.Type` object in turn can be used to access further information about the compiled form of F# member declarations. If this operation is supported in a particular implementation of F#, then the following rules describe which declaration elements have corresponding `System.Reflection` objects:

- All member declarations are present as corresponding methods, properties or events.
- Private and internal members and types are included.
- Type abbreviations are not given corresponding `System.Type` definitions.

In addition:

- F# modules are compiled to provide a corresponding compiled CLI type declaration and `System.Type` object, although the `System.Type` object is not accessible by using the `typeof` operator.

However:

- Internal and private function and value definitions are not guaranteed to be given corresponding compiled CLI metadata definitions. They may be removed by optimization.
- Additional internal and private compiled type and member definitions may be present in the compiled CLI assembly as necessary for the correct implementation of F# programs.
- The `System.Reflection` operations return results that are consistent with the erasure of F# type abbreviations and F# unit-of-measure annotations.
- The definition of new units of measure results in corresponding compiled CLI type declarations with an associated `System.Type`.

14. Inference Procedures

14.1 Name Resolution

The following sections describe how F# resolves names in various contexts.

14.1.1 Name Environments

Each point in the interpretation of an F# program is subject to an environment. The environment encompasses:

- All referenced external DLLs (assemblies).
- *ModulesAndNamespaces*: a table that maps *`long-ident`*s to a list of signatures. Each signature is either a namespace declaration group signature or a module signature.

  For example, `System.Collections` may map to one namespace declaration group signature for each referenced assembly that contributes to the `System.Collections` namespace, and to a module signature, if a module called `System.Collections` is declared or in a referenced assembly.

  If the program references multiple assemblies, the assemblies are added to the name resolution environment in the order in which the references appear on the command line. The order is important only if ambiguities occur in referencing the contents of assemblies—for example, if two assemblies define the type `MyNamespace.C`.

- *ExprItems*: a table that maps names to the following items:
  - A value
  - A union case for use when constructing data
  - An active pattern result tag for use when returning results from active patterns
  - A type name for each class or struct type
- *FieldLabels*: a table that maps names to sets of field references for record types
- *PatItems*: a table that maps names to the following items:
  - A union case, for use when pattern matching on data
  - An active pattern case name, for use when specifying active patterns
  - A literal definition
- *Types*: a table that maps names to type definitions. Two queries are supported on this table:
  - Find a type by name alone. This query may return multiple types. For example, in the default type-checking environment, the resolution of `System.Tuple` returns multiple tuple types.

- Find a type by name and generic arity *n*. This query returns at most one type. For example, in the default type-checking environment, the resolution of `System.Tuple` with `n = 2` returns a single type.

- *ExtensionsInScope*: a table that maps type names to one or more member definitions

The dot notation is resolved during type checking by consulting these tables.

14.1.2 Name Resolution in Module and Namespace Paths

Given an input *`long-ident`* and environment *`env`*, *Name Resolution in Module and Namespace Paths* computes the result of interpreting *`long-ident`* as a module or namespace. The procedure returns a list of modules and namespace declaration groups.

Name Resolution in Module and Namespace Paths proceeds through the following steps:

1. Consult the *ModulesAndNamespaces* table to resolve the *`long-ident`* prefix to a list of modules and namespace declaration group signatures.

2. If any identifiers remain unresolved, recursively consult the declared modules and sub-modules of these namespace declaration groups.

3. Concatenate all the results.

If the *`long-ident`* starts with the special pseudo-identifier keyword `global`, the identifier is resolved by consulting the *ModulesAndNamespaces* table and ignoring all `open` directives, including those implied by `AutoOpen` attributes.

For example, if the environment contains two referenced DLLs, and each DLL has namespace declaration groups for the namespaces `System`, `System.Collections`, and `System.Collections.Generic`, *Name Resolution in Module and Namespace Paths* for `System.Collections` returns the two namespace declaration groups named `System.Collections`, one from each assembly.

14.1.3 Opening Modules and Namespace Declaration Groups

When a module or namespace declaration group *F* is opened, the compiler adds items to the name environment as follows:

1. Add each exception label for each exception type definition (§8.11) in *F* to the *ExprItems* and *PatItems* tables in the original order of declaration in *F*.

2. Add each type definition in the original order of declaration in *F*. Adding a type definition involves the following procedure:

    a. If the type is a class or struct type (or an abbreviation of such a type), add the type name to the *ExprItems* table.

    b. If the type definition is a record, add the record field labels to the *FieldLabels* table, unless the type has the `RequireQualifiedAccess` attribute.

c. If the type is a union, add the union cases to the *ExprItems* and *PatItems* tables, unless the type has the `RequireQualifiedAccess` attribute.

d. Add the type to the *TypeNames* table. If the type has a CLI-encoded generic name such as `` List`1 ``, add an entry under both `List` and `` List`1 ``.

3. Add each value in the original order of declaration in *F*, as follows:

   a. Add the value to the *ExprItems* table.

   b. If any value is an active pattern, add the tags of that active pattern to the *PatItems* table according to the original order of declaration.

   c. If the value is a literal, add it to the *PatItems* table.

4. Add the member contents of each type extension in F_i to the *ExtensionsInScope* table according to the original order of declaration in F_i.

5. Add each sub-module or sub-namespace declaration group in F_i to the *ModulesAndNamespaces* table according to the original order of declaration in F_i.

6. Open any sub-modules that are marked with the `FSharp.Core.AutoOpen` attribute.

14.1.4 Name Resolution in Expressions

Given an input *long-ident*, environment *env*, and an optional count *n* of the number of subsequent type arguments *<_,...,_>*, *Name Resolution in Expressions* computes a result that contains the interpretation of the *long-ident<_,...,_>* prefix as a value or other expression item, and a residue path *rest*.

How Name Resolution in Expressions proceeds depends on whether *long-ident* is a single identifier or is composed of more than one identifier.

If *long-ident* is a single identifier *ident*:

1. Look up *ident* in the *ExprItems* table. Return the result and empty *rest*.

2. If *ident* does not appear in the *ExprItems* table, look it up in the *Types* table, with generic arity that matches *n* if available. Return this type and empty *rest*.

3. If *ident* does not appear in either the *ExprItems* table or the *Types* table, fail.

If *long-ident* is composed of more than one identifier *ident.rest*, *Name Resolution in Expressions* proceeds as follows:

1. If *ident* exists as a value in the *ExprItems* table, return the result, with *rest* as the residue.

2. If *ident* does not exist as a value in the *ExprItems* table, perform a backtracking search as follows:

   a. Consider each division of *long-ident* into [*namespace-or-module-path*].*ident*[.*rest*], in which the *namespace-or-module-path* becomes successively longer.

b. For each such division, consider each module signature or namespace declaration group signature `F` in the list that is produced by resolving `namespace-or-module-path` by using *Name Resolution in Module and Namespace Paths*.

c. For each such `F`, attempt to resolve `ident`[`.rest`] in the following order. If any resolution succeeds, then terminate the search:

   1) A value in `F`. Return this item and `rest`.

   2) A union case in `F`. Return this item and `rest`.

   3) An exception constructor in `F`. Return this item and `rest`.

   4) A type in `F`. If `rest` is empty, then return this type; if not, resolve using *Name Resolution for Members*.

   5) A [sub-]module in `F`. Recursively resolve `rest` against the contents of this module.

3. If steps 1 and 2 do not resolve `long-ident`, look up `ident` in the *Types* table.

   a. If the generic arity `n` is available, then look for a type that matches both `ident` and `n`.

   b. If no generic arity `n` is available, and `rest` is not empty:

      1) If the *Types* table contains a type `ident` that does not have generic arguments, resolve to this type.

      2) If the *Types* table contains a unique type `ident` that has generic arguments, resolve to this type. However, if the overall result of the *Name Resolution in Expressions* operation is a member, and the generic arguments do not appear in either the return or argument types of the item, warn that the generic arguments cannot be inferred from the type of the item.

      3) If neither of the preceding steps resolves the type, give an error.

   c. If `rest` is empty, return the type, otherwise resolve using *Name Resolution for Members*.

4. If steps 1-3 do not resolve `long-ident`, look up `ident` in the *ExprItems* table and return the result and residue `rest`.

5. Otherwise, if `ident` is a symbolic operator name, resolve to an item that indicates an implicitly resolved symbolic operator.

6. Otherwise, fail.

If the expression contains ambiguities, *Name Resolution in Expressions* returns the first result that the process generates. For example, consider the following cases:

```
module M =
    type C =
        | C of string
        | D of string
        member x.Prop1 = 3
    type Data =
        | C of string
        | E
        member x.Prop1 = 3
        member x.Prop2 = 3
    let C = 5
   open M
   let C = 4
   let D = 6

   let test1 = C             // resolves to the value C
   let test2 = C.ToString() // resolves to the value C with residue
ToString
   let test3 = M.C           // resolves to the value M.C
   let test4 = M.Data.C      // resolves to the union case M.Data.C
   let test5 = M.C.C         // error: first part resolves to the value
M.C,
                          //     and this contains no field or property
"C"
   let test6 = C.Prop1       // error: the value C does not have a
property Prop
   let test7 = M.E.Prop2     // resolves to M.E, and then a property
lookup
```

The following example shows the resolution behavior for type lookups that are ambiguous by generic arity:

```
module M =
    type C<'T>() =
        static member P = 1

    type C<'T,'U>() =
        static member P = 1

   let _ = new M.C()          // gives an error
   let _ = new M.C<int>()     // no error, resolves to C<'T>
   let _ = M.C()              // gives an error
   let _ = M.C<int>()         // no error, resolves to C<'T>
   let _ = M.C<int,int>()     // no error, resolves to C<'T,'U>
   let _ = M.C<_>()           // no error, resolves to C<'T>
   let _ = M.C<_,_>()         // no error, resolves to C<'T,'U>
   let _ = M.C.P              // gives an error
   let _ = M.C<_>.P           // no error, resolves to C<'T>
   let _ = M.C<_,_>.P         // no error, resolves to C<'T,'U>
```

The following example shows how the resolution behavior differs slightly if one of the types has no generic arguments.

```
module M =
    type C() =
        static member P = 1

    type C<'T>() =
        static member P = 1

   let _ = new M.C()           // no error, resolves to C
   let _ = new M.C<int>()      // no error, resolves to C<'T>
   let _ = M.C()               // no error, resolves to C
   let _ = M.C< >()            // no error, resolves to C
   let _ = M.C<int>()          // no error, resolves to C<'T>
   let _ = M.C< >()            // no error, resolves to C
   let _ = M.C<_>()            // no error, resolves to C<'T>
   let _ = M.C.P               // no error, resolves to C
   let _ = M.C< >.P            // no error, resolves to C
   let _ = M.C<_>.P            // no error, resolves to C<'T>
```

In the following example, the procedure issues a warning for an incomplete type. In this case, the type parameter `'T` cannot be inferred from the use `M.C.P`, because `'T` does not appear at all in the type of the resolved element `M.C<'T>.P`.

```
module M =
    type C<'T>() =
        static member P = 1

   let _ = M.C.P               // no error, resolves to C<'T>.P, warning
given
```

The effect of these rules is to prefer value names over module names for single identifiers. For example, consider this case:

```
let Foo = 1

module Foo =
   let ABC = 2
let x1 = Foo // evaluates to 1
```

The rules, however, prefer type names over value names for single identifiers, because type names appear in the *ExprItems* table. For example, consider this case:

```
let Foo = 1
type Foo() =
    static member ABC = 2
let x1 = Foo.ABC // evaluates to 2
let x2 = Foo() // evaluates to a new Foo()
```

14.1.5 Name Resolution for Members

Name Resolution for Members is a sub-procedure used to resolve `.member-ident[.rest]` to a member, in the context of a particular type *type*.

Name Resolution for Members proceeds through the following steps:

1. Search the hierarchy of the type from `System.Object` to *type*.

2. At each type, try to resolve *member-ident* to one of the following, in order:

    a. A union case of *type*.

    b. A property group of *type*.

    c. A method group of *type*.

    d. A field of *type*.

    e. An event of *type*.

    f. A property group of extension members of *type*, by consulting the *ExtensionsInScope* table.

    g. A method group of extension members of *type*, by consulting the *ExtensionsInScope* table.

    h. A nested type *type-nested* of *type*. Recursively resolve `.rest` if it is present, otherwise return *type-nested*.

3. At any type, the existence of a property, event, field, or union case named *member-ident* causes any methods or other entities of that same name from base types to be hidden.

4. Combine method groups with method groups from base types. For example:

```
type A() =
    member this.Foo(i : int) = 0

type B() =
    inherit A()
    member this.Foo(s : string) = 1

let b = new B()
b.Foo(1)     // resolves to method in A
b.Foo("abc") // resolves to method in B
```

14.1.6 Name Resolution in Patterns

Name Resolution for Patterns is used to resolve *long-ident* in the context of pattern expressions. The *long-ident* must resolve to a union case, exception label, literal value, or active pattern case name. If it does not, the *long-ident* may represent a new variable definition in the pattern.

Name Resolution for Patterns follows the same steps to resolve the *member-ident* as *Name Resolution in Expressions* (§14.1.4) except that it consults the *PatItems* table instead of the *ExprItems* table. As a result, values are not present in the namespace that is used to resolve identifiers in patterns. For example:

```
let C = 3
match 4 with
| C -> sprintf "matched, C = %d" C
| _ -> sprintf "no match, C = %d" C
```

results in `"matched, C = 4"`, because `C` is *not* present in the *PatItems* table, and hence becomes a value pattern. In contrast,

```
[<Literal>]
let C = 3

match 4 with
| C -> sprintf "matched, C = %d" C
| _ -> sprintf "no match, C = %d" C
```

results in `"no match, C = 3"`, because `C` is a literal and therefore *is* present in the *PatItems* table.

14.1.7 Name Resolution for Types

Name Resolution for Types is used to resolve *`long-ident`* in the context of a syntactic type. A generic arity that matches *n* is always available. The result is a type definition and a possible residue *`rest`*.

Name Resolution for Types proceeds through the following steps:

1. Given *`ident`*[*`.rest`*], look up *`ident`* in the *Types* table, with generic arity *n*. Return the result and residue *`rest`*.

2. If *`ident`* is not present in the *Types* table:

   a. Divide *`long-ident`* into [*`namespace-or-module-path`*]*`.ident`*[*`.rest`*], in which the *`namespace-or-module-path`* becomes successively longer.

   b. For each such division, consider each module and namespace declaration group *F* in the list that results from resolving *`namespace-or-module-path`* by using *Name Resolution in Module and Namespace Paths* (§14.1.2).

   c. For each such *F*, attempt to resolve *`ident`*[*`.rest`*] in the following order. Terminate the search when the expression is successfully resolved.

      1) A type in *F*. Return this type and residue *`rest`*.

      2) A [sub-]module in *F*. Recursively resolve *`rest`* against the contents of this module.

In the following example, the name `C` on the last line resolves to the named type `M.C<_,_>` because `C` is applied to two type arguments:

```
module M =
    type C<'T, 'U> = 'T * 'T * 'U

module N =
    type C<'T> = 'T * 'T

open M
open N

let x : C<int, string> = (1, 1, "abc")
```

14.1.8 Name Resolution for Type Variables

Whenever the F# compiler processes syntactic types and expressions, it assumes a context that maps identifiers to inference type variables. This mapping ensures that multiple uses of the same type variable name map to the same type inference variable. For example, consider the following function:

```
let f x y = (x:'T), (y:'T)
```

In this case, the compiler assigns the identifiers `x` and `y` the same static type—that is, the same type inference variable is associated with the name `'T`. The full inferred type of the function is:

```
val f<'T> : 'T -> 'T -> 'T * 'T
```

The map is used throughout the processing of expressions and types in a left-to-right order. It is initially empty for any member or any other top-level construct that contains expressions and types. Entries are eliminated from the map after they are generalized. As a result, the following code checks correctly:

```
let f () =
    let g1 (x:'T) = x
    let g2 (y:'T) = (y:string)
    g1 3, g1 "3", g2 "4"
```

The compiler generalizes `g1`, which is applied to both integer and string types. The type variable `'T` in (y:'T) on the third line refers to a different type inference variable, which is eventually constrained to be type `string`.

14.1.9 Field Label Resolution

Field Label Resolution specifies how to resolve identifiers such as *`field1`* in `{ `*`field1`*` = `*`expr`*`; ... `*`fieldN`*` = `*`expr`*` }`.

Field Label Resolution proceeds through the following steps:

1. Look up all fields in all available types in the *Types* table and the *FieldLabels* table (§8.4.2).

2. Return the set of field declarations.

14.2 Resolving Application Expressions

Application expressions that use dot notation—such as *x.Y<int>.Z(g).H.I.j*—are resolved according to a set of rules that take into account the many possible shapes and forms of these expressions and the ambiguities that may occur during their resolution. This section specifies the exact algorithmic process that is used to resolve these expressions.

Resolution of application expressions proceeds as follows:

1. Repeatedly decompose the application expression into a leading expression *expr* and a list of projections `projs`. Each projection has the following form:

   - *.long-ident-or-op* is a dot lookup projection.
   - *expr* is an application projection.
   - *<types>* is a type application projection.

   For example:

   - `x.y.Z(g).H.I.j` decomposes into `x.y.Z` and projections `(g)`, `.H.I.j`.
   - `x.M<int>(g)` decomposes into `x.M` and projections `<int>`, `(g)`.
   - `f x` decomposes into `f` and projection `x`.

   Note: In this specification we write sequences of projections by juxtaposition; for example, *(expr).long-ident<types>(expr)*. We also write *(.rest + projs)* to refer to adding a residue long identifier to the front of a list of projections, which results in *projs* if *rest* is empty and *.rest projs* otherwise.

2. After decomposition:

   - If *expr* is a long identifier expression *long-ident*, apply *Unqualified Lookup* (§14.2.1) on *long-ident* with projections *projs*.
   - If *expr* is not such an expression, check the expression against an arbitrary initial type *ty*, to generate an elaborated expression *expr*. Then process *expr*, *ty*, and *projs* by using *Expression-Qualified Lookup* (§14.2.3)

14.2.1 Unqualified Lookup

Given an input *long-ident* and projections *projs*, *Unqualified Lookup* computes the result of "looking up" *long-ident.projs* in an environment *env*. The first part of this process resolves a prefix of the information in *long-ident.projs*, and recursive resolutions typically use *Expression-Qualified Resolution* to resolve the remainder.

For example, *Unqualified Lookup* is used to resolve the vast majority of identifier references in F# code, from simple identifiers such as `sin`, to complex accesses such as `System.Environment.GetCommandLineArgs().Length`.

Unqualified Lookup proceeds through the following steps:

1. Resolve `long-ident` by using *Name Resolution in Expressions* (§14.1). This returns a *name resolution item* `item` and a *residue long identifier* `rest`.

   For example, the result of *Name Resolution in Expressions* for `v.X.Y` may be a value reference `v` along with a residue long identifier `.X.Y`. Likewise, `N.X(args).Y` may resolve to an overloaded method `N.X` and a residue long identifier `.Y`.

   Name Resolution in Expressions also takes as input the presence and count of subsequent type arguments in the first projection. If the first projection in `projs` is `<tyargs>`, *Unqualified Lookup* invokes *Name Resolution in Expressions* with a known number of type arguments. Otherwise, it is invoked with an unknown number of type arguments.

2. Apply *Item-Qualified Lookup* for `item` and (`rest` + `projs`).

14.2.2 Item-Qualified Lookup

Given an input item `item` and projections `projs`, *Item-Qualified Lookup* computes the projection `item.projs`. This computation is often a recursive process: the first resolution uses a prefix of the information in `item.projs`, and recursive resolutions resolve any remaining projections.

Item-Qualified Lookup proceeds as follows:

1. If `item` is not one of the following, return an error:

   - A named value
   - A union case
   - A group of named types
   - A group of methods
   - A group of indexer getter properties
   - A single non-indexer getter property
   - A static F# field
   - A static CLI field
   - An implicitly resolved symbolic operator name

2. If the first projection is `<types>`, then we say the resolution has a type application `<types>` with remaining projections.

3. Otherwise, checking proceeds as shown in the table.

If `item` is:	Action

If *item* is:	Action
A value reference v	1. Instantiate the type scheme of v, which results in a type *ty*. Apply these rules: ▪ If the first projection is *<types>*, process the types and use the results as the arguments to instantiate the type scheme. ▪ If the first projection is not *<types>*, the type scheme is *freshly instantiated*. ▪ If the value has the RequiresExplicitTypeArguments attribute, the first projection must be *<types>*. ▪ If the value has type byref<ty_2>, add a byref dereference to the elaborated expression. ▪ Insert implicit flexibility for the use of the value (§14.4.3). 2. Apply *Expression-Qualified Lookup* for type *ty* and any remaining projections.
A type name, where *projs* begins with *<types>.long-ident*	1. Process the types and use the results as the arguments to instantiate the named type reference, thus generating a type *ty*. 2. Apply *Name Resolution for Members* to *ty* and *long-ident*, which generates a new *item*. 3. Apply *Item-Qualified Lookup* to the new *item* and any remaining projections.
A group of type names where *projs* begins with *<types>* or *expr*	1. Process the types and use the results as the arguments to instantiate the named type reference, thus generating a type *ty*. 2. Process the object construction *ty(expr)* as an object constructor call in the same way as new *ty(expr)*. 3. Apply *Expression-Qualified Lookup* to *item* and any remaining projections.
A group of method references	1. Apply *Method Application Resolution* for the method group. *Method Application Resolution* accepts an optional set of type arguments and a syntactic expression argument. Determine the arguments based on what *projs* begins with: ▪ *<types> expr*, then use *<types>* as the type arguments and *expr* as the expression argument. ▪ *expr*, then use *expr* as the expression argument. ▪ anything else, use no expression argument or type arguments. 2. If the result of *Method Application Resolution* is labeled with the RequiresExplicitTypeArguments attribute, then explicit type arguments are required. 3. Let *fty* be the actual return type that results from *Method Application Resolution*. Apply *Expression-Qualified Lookup* to *fty* and any remaining projections.
A group of property indexer references	1. Apply *Method Application Resolution*, and use the underlying getter indexer methods for the method group. 2. Determine the arguments to *Method Application Resolution* as described for a group of methods.
A static field reference	1. Check the field for accessibility and attributes. 2. Let *fty* be the actual type of the field, taking into account the type *ty* via which the field was accessed in the case where this is a field in a generic type. 3. Apply *Expression-Qualified Lookup* to *fty* and *projs*.
A union case tag, exception tag, or active pattern result element tag	1. Check the tag for accessibility and attributes. 2. If *projs* begins with *expr*, use *expr* as the expression argument. 3. Otherwise, use no expression argument or type arguments. In this case, build a function expression for the union case. 4. Let *fty* be the actual type of the union case. 5. Apply *Expression-Qualified Lookup* to *fty* and remaining *projs*.
A CLI event reference	1. Check the event for accessibility and attributes. 2. Let *fty* be the actual type of the event. 3. Apply *Expression-Qualified Lookup* to *fty* and *projs*.

If *item* is:	Action

An implicitly resolved symbolic operator name *op*	1. If *op* is a unary, binary or the ternary operator `?<-`, resolve to the following expressions, respectively:
	`(fun (x:^a) -> (^a : static member (op) : ^a -> ^b) x)`
	`(fun (x:^a) (y:^b) ->`
	`    ((^a or ^b) : static member (op) : ^a * ^b -> ^c) (x,y))`
	`(fun (x:^a) (y:^b) (z:^c) ->`
	`    ((^a or ^b or ^c) : static member (op) : ^a * ^b * ^c -> ^d) (x,y,z))`
	2. The resulting expressions are static member constraint invocation expressions (§0), which enable the default interpretation of operators by using type-directed member resolution.
	3. Recheck the entire expression with additional subsequent projections *.projs*.

14.2.3 Expression-Qualified Lookup

Given an elaborated expression *expr* of type *ty*, and projections *projs*, *Expression-Qualified Lookup* computes the "lookups or applications" for *expr.projs*.

Expression-Qualified Lookup proceeds through the following steps:

1. Inspect *projs* and process according to the following table.

projs	Action	Comments
Empty	Assert that the type of the overall, original application expression is *ty*.	Checking is complete.
Starts with *(expr2)*	Apply *Function Application Resolution* (§14.3).	Checking is complete when *Function Application Resolution* returns.
Starts with *<types>*	Fail.	Type instantiations may not be applied to arbitrary expressions; they can apply only to generic types, generic methods, and generic values.
Starts with *.long-ident*	Resolve *long-ident* using *Name Resolution for Members* (§14.1.4). Return a name resolution item *item* and a residue long identifier *rest*. Continue processing at step 2.	For example, for *ty* = `string` and *long-ident* = `Length`, *Name Resolution for Members* returns a property reference to the CLI instance property `System.String.Length`.

2. If Step 1 returned an *item* and *rest*, report an error if *item* is not one of the following:

- A group of methods.
- A group of instance getter property indexers.
- A single instance, non-indexer getter property.
- A single instance F# field.
- A single instance CLI field.

3. Proceed based on *item* as shown in the table:

If *item* is:	Action

If `item` is:	Action
Group of methods	1. Apply *Method Application Resolution* for the method group. *Method Application Resolution* accepts an optional set of type arguments and a syntactic expression argument. If `projs` begins with: ▪ `<types>(arg)`, then use `<types>` as the type arguments and `arg` as the expression argument. ▪ `(arg)`, then use `arg` as the expression argument. ▪ otherwise, use no expression argument or type arguments. 2. Let `fty` be the actual return type resulting from *Method Application Resolution*. Apply *Expression-Qualified Lookup* to `fty` and any remaining projections.
Group of indexer properties	1. Apply *Method Application Resolution* and use the underlying getter indexer methods for the method group. 2. Determine the arguments to *Method Application Resolution* as described for a group of methods.
Non-indexer getter property	Apply *Method Application Resolution* for the method group that contains only the getter method for the property, with no type arguments and one `()` argument.
Instance intermediate language (IL) or F# field `F`	1. Check the field for accessibility and attributes. 2. Let `fty` be the actual type of the field (taking into account the type `ty` by which the field was accessed). 3. Assert that `ty` is a subtype of the actual containing type of the field. 4. Produce an elaborated form for `expr.F`. If `F` is a field in a value type then take the address of `expr` by using the *AddressOf* (`expr` , `NeverMutates`) operation §6.9.4. 5. Apply *Expression-Qualified Lookup* to `fty` and `projs`.

14.3 Function Application Resolution

Given expressions `f` and `expr` where `f` has type `ty`, and given subsequent projections `projs`, *Function Application Resolution* does the following:

1. Asserts that `f` has type `ty`$_1$ `->` `ty`$_2$ for new inference variables `ty`$_1$ and `ty`$_2$.
2. If the assertion succeeds:
   a. Check `expr` with the initial type `ty`$_1$.
   b. Process `projs` using *Expression-Qualified* against `ty`$_2$.
3. If the assertion fails, and `expr` has the form `{ computation-expr }`:
   a. Check the expression as the computation expression form `f { computation-expr }`, giving result type `ty`$_1$.
   b. Process `projs` using *Expression-Qualified Lookup* against `ty`$_1$.

14.4 Method Application Resolution

Given a method group `M`, optional type arguments `<ActualTypeArgs>`, an optional syntactic argument `obj`, an optional syntactic argument `arg`, and overall initial type `ty`, *Method Application Resolution* resolves the overloading based on the partial type information that is available. It also:

- Resolves optional and named arguments.
- Resolves “out” arguments.
- Resolves post-hoc property assignments.
- Applies method application resolution.
- Inserts *ad hoc* conversions that are only applied for method calls.

If no syntactic argument is supplied, *Method Application Resolution* tries to resolve the use of the method as a first class value, such as the method call in the following example:

```
List.map System.Environment.GetEnvironmentVariable ["PATH"; "USERNAME"]
```

Method Application Resolution proceeds through the following steps:

1. Restrict the candidate method group `M` to those methods that are *accessible* from the point of resolution.

2. If an argument `arg` is present, determine the sets of *unnamed* and *named actual arguments*, `UnnamedActualArgs` and `NamedActualArgs`:

   a. Decompose `arg` into a list of arguments:

      - If `arg` is a syntactic tuple `arg1 ,..., argN`, use these arguments.
      - If `arg` is a syntactic unit value `()`,use a zero-length list of arguments.

   b. For each argument:

      - If `arg` is a binary expression of the form `name=expr`, it is a named actual argument.
      - Otherwise, `arg` is an unnamed actual argument.

   If there are no named actual arguments, and `M` has only one candidate method, which accepts only one required argument, ignore the decomposition of `arg` to tuple form. Instead,`arg` itself is the only named actual argument.

   All named arguments must appear after all unnamed arguments.

   Examples:

   `x.M(1, 2)` has two unnamed actual arguments.

   `x.M(1, y = 2)` has one unnamed actual argument and one named actual argument.

   `x.M(1, (y = 2))` has two unnamed actual arguments.

   `x.M( printfn "hello"; ())` has one unnamed actual argument.

`x.M((a, b))` has one unnamed actual argument.

`x.M(())` has one unnamed actual argument.

3. Determine the named and unnamed *prospective actual argument types*, called *ActualArgTypes*.

   - If an argument *arg* is present, the prospective actual argument types are fresh type inference variables for each unnamed and named actual argument.

     - If the argument has the syntactic form of an address-of expression `&expr` after ignoring parentheses around the argument, equate this type with a type `byref<ty>` for a fresh type *ty*.

     - If the argument has the syntactic form of a function expression `fun` pat_1 … pat_n `->` *expr* after ignoring parentheses around the argument, equate this type with a type ty_1 `->` … ty_n `->` *rty* for fresh types ty_1 … ty_n.

   - If no argument *arg* is present:

     a. If the method group contains a single method, the prospective unnamed argument types are one fresh type inference variable for each required, non-“out” parameter that the method accepts.

     b. If the method group contains more than one method, the expected overall type of the expression is asserted to be a function type *dty* `->` *rty*.

        - If *dty* is a tuple type `(dty1 * .. * dtyN)`, the prospective argument types are `(dty1, .. ,dtyN)`.

        - If *dty* is `unit`, then the prospective argument types are an empty list.

        - If *dty* is any other type, the prospective argument types are *dty* alone.

     c. Subsequently:

        - The method application is considered to have one unnamed actual argument for each prospective unnamed actual argument type.

        - The method application is considered to have no named actual arguments.

4. For each candidate method in `M`, attempt to produce zero, one, or two *prospective method calls* $M_{possible}$ as follows:

   a. If the candidate method is generic and has been generalized, generate fresh type inference variables for its generic parameters. This results in the *FormalTypeArgs* for $M_{possible}$.

   b. Determine the *named* and *unnamed formal parameters*, called *NamedFormalArgs* and *UnnamedFormalArgs* respectively, by splitting the formal parameters for `M` into parameters that have a matching argument in *NamedActualArgs* and parameters that do not.

   c. If the number of *UnnamedFormalArgs* exceeds the number of *UnnamedActualArgs*, then modify *UnnamedFormalArgs* as follows:

- Determine the suffix of *UnnamedFormalArgs* beyond the number of *UnnamedActualArgs*.

- If all formal parameters in the suffix are "out" arguments with byref type, remove the suffix from *UnnamedFormalArgs* and call it *ImplicitlyReturnedFormalArgs*.

- If all formal parameters in the suffix are optional arguments, remove the suffix from *UnnamedFormalArgs* and call it *ImplicitlySuppliedFormalArgs*.

d. If the last element of *UnnamedFormalArgs* has the `ParamArray` attribute and type *pty*`[]` for some *pty*, then modify *UnnamedActualArgs* as follows:

- If the number of *UnnamedActualArgs* exceeds the number of *UnnamedFormalArgs-1*, produce a prospective method call named *ParamArrayActualArgs* that has the excess of *UnnamedActualArgs* removed.

- If the number of *UnnamedActualArgs* equals the number of *UnnamedFormalArgs*`-1`, produce two prospective method calls:

  - One has an empty *ParamArrayActualArgs*.

  - One has no *ParamArrayActualArgs*.

- If *ParamArrayActualArgs* has been produced, then $\mathtt{M}_{\mathtt{possible}}$ is said to *use ParamArray conversion with type* `pty`.

e. Associate each *name=arg* in *NamedActualArgs* with a *target*. A target is a *named formal parameter*, a *settable return property*, or a *settable return field* as follows:

- If one of the arguments in *NamedFormalArgs* has name *name*, that argument is the target.

- If the return type of `M`, before the application of any type arguments *ActualTypeArgs*, contains a settable property *name*,then `M` is the target.

- If the return type of `M`, before the application of any type arguments *ActualTypeArgs*, contains a settable field *name*, then `M` is the target.

f. No prospective method call is generated if any of the following are true:

- A named argument cannot be associated with a target.

- The number of *UnnamedActualArgs* is less than the number of *UnnamedFormalArgs* after steps 4 a-e.

- The number of *ActualTypeArgs*, if any actual type arguments are present, does not precisely equal the number of *FormalTypeArgs* for `M`.

- The candidate method is static and the optional syntactic argument `obj` is present, or the candidate method is an instance method and `obj` is not present.

5. Attempt to apply initial types before argument checking. If only one prospective method call $M_{possible}$ exists, *assert* $M_{possible}$ by performing the following steps:

   a. Verify that each *ActualTypeArg*$_i$ is equal to its corresponding *FormalTypeArg*$_i$.

   b. Verify that the type of *obj* is a subtype of the containing type of the method M.

   c. For each *UnnamedActualArg*$_i$ and *UnnamedFormalArg*$_i$, verify that the corresponding *ActualArgType* coerces to the type of the corresponding argument of M.

   d. If $M_{possible}$ uses ParamArray conversion with type *pty*, then for each *ParamArrayActualArg*$_i$, verify that the corresponding *ActualArgType* coerces to *pty*.

   e. For each *NamedActualArg*$_i$ that has an associated formal parameter target, verify that the corresponding *ActualArgType* coerces to the type of the corresponding argument of M.

   f. For each *NamedActualArg*$_i$ that has an associated property or field setter target, verify that the corresponding *ActualArgType* coerces to the type of the property or field.

   g. Verify that the *prospective formal return type* coerces to the *expected actual return type*. If the method M has return type *rty*, the *formal return type* is defined as follows:

      - If the prospective method call contains *ImplicitlyReturnedFormalArgs* with type ty_1, ..., ty_N, the formal return type is *rty* * ty_1 * ... * ty_N. If *rty* is unit then the formal return type is ty_1 * ... * ty_N.

      - Otherwise the formal return type is *rty*.

6. Check and elaborate argument expressions. If *arg* is present:

   - Check and elaborate each unnamed actual argument expression *arg*$_i$. Use the corresponding type in *ActualArgTypes* as the initial type.

   - Check and elaborate each named actual argument expression *arg*$_i$. Use the corresponding type in *ActualArgTypes* as the initial type.

7. Choose a unique $M_{possible}$ according to the following rules:

   - For each $M_{possible}$, determine whether the method is *applicable* by attempting to assert $M_{possible}$ as described in step 4a). If the actions in step 4a detect an inconsistent constraint set (§14.5), the method is not applicable. Regardless, the overall constraint set is left unchanged as a result of determining the applicability of each $M_{possible}$.

   - If a unique applicable $M_{possible}$ exists, choose that method. Otherwise, choose the unique *best* $M_{possible}$ by applying the following criteria, in order:

     1) Prefer candidates whose use does not constrain the use of a user-introduced generic type annotation to be equal to another type.

     2) Prefer candidates that do not use ParamArray conversion. If two candidates both use ParamArray conversion with types *pty*$_1$ and *pty*$_2$, and *pty*$_1$ feasibly subsumes *pty*$_2$, prefer the second; that is, use the candidate that has the more precise type.

     3) Prefer candidates that do not have *ImplicitlyReturnedFormalArgs*.

     4) Prefer candidates that do not have *ImplicitlySuppliedFormalArgs*.

5) If two candidates have unnamed actual argument types ty_{11} ... ty_{1n} and ty_{21} ... ty_{2n}, and each ty_{1i} either

   a. feasibly subsumes ty_{2i}, or

   b. ty_{2i} is a `System.Func` type and ty_{1i} is some other delegate type,

   then prefer the second candidate. That is, prefer any candidate that has the more specific actual argument types, and consider any `System.Func` type to be more specific than any other delegate type.

6) Prefer candidates that are not extension members over candidates that are.

7) To choose between two extension members, prefer the one that results from the most recent use of `open`.

8) Prefer candidates that are not generic over candidates that are generic—that is, prefer candidates that have empty *ActualArgTypes*.

Report an error if steps 1) through 8) do not result in the selection of a unique better method.

8. Once a unique best $M_{possible}$ is chosen, commit that method.

9. Apply attribute checks.

10. Build the resulting elaborated expression by following these steps:

    a. If the type of *obj* is a variable type or a value type, take the address of *obj* by using the *AddressOf*(*obj*, `PossiblyMutates`) operation (§6.9.4).

    b. Build the argument list by:

       - Passing each argument corresponding to an *UnamedFormalArgs* where the argument is an optional argument as a `Some` value.

       - Passing a `None` value for each argument that corresponds to an *ImplicitlySuppliedFormalArgs*.

       - Applying coercion to arguments.

    c. Bind *ImplicitlyReturnedFormalArgs* arguments by introducing mutable temporaries for each argument, passing them as byref parameters, and building a tuple from these mutable temporaries and any method return value as the overall result.

    d. For each *NamedActualArgs* whose target is a settable property or field, assign the value into the property.

    e. If *arg* is not present, return a function expression that represents a first class function value.

Two additional rules apply when checking arguments (see §8.13.7 for examples):

- If a formal parameter has delegate type *D*, an actual argument *farg* has known type ty_1 `->` ... `->` ty_n `->` *rty*, and the number of arguments of the Invoke method of delegate type *D* is precisely *n*, interpret the formal parameter in the same way as the following:

  ```
  new D(fun arg1 ... argn -> farg arg1 ... argn).
  ```

  For more information on the conversions that are automatically applied to arguments, see §8.13.6.

- If a formal parameter is an "out" parameter of type `byref<ty>`, and an actual argument type is not a byref type, interpret the actual parameter in the same way as type `ref<ty>`. That is, an F# reference cell can be passed where a `byref<ty>` is expected.

One effect of these additional rules is that a method that is used as a first class function value can resolve even if a method is overloaded and no further information is available. For example:

```
let r = new Random()
let roll = r.Next;;
```

Method Application Resolution results in the following, despite the fact that in the standard CLI library, `System.Random.Next` is overloaded:

```
val roll : int -> int
```

The reason is that if the initial type contains no information about the expected number of arguments, the F# compiler assumes that the method has one argument.

14.4.1 Additional Propagation of Known Type Information in F# 3.1

In the above descreiption of F# overload resolution, the argument expressions of a call to an overloaded set of methods

```
callerObjArgTy.Method(callerArgExpr1, … callerArgExprN)
```

calling

```
calledObjArgTy.Method(calledArgTy1, … calledArgTyN)
```

In F# 3.1 and subsequently, immediately prior to checking argument expressions, each argument position of the unnamed caller arguments for the method call is analysed to propagate type information extracted from method overloads to the expected types of lambda expressions. The new rule is applied when

- the candidates are overloaded
- the caller argument at the given unnamed argument position is a syntactic lambda, possible parenthesized
- all the corresponding formal called arguments have *calledArgTy* either of

- function type “`calledArgDomainTy1 -> … -> calledArgDomainTyN -> calledArgRangeTy`“ (after taking into account “function to delegate” adjustments), or
- some other type which would cause an overload to be discarded

- at least one overload has enough curried lambda arguments for it corresponding expected function type

In this case, for each unnamed argument position, then for each overload:

- Attempt to solve “`callerObjArgTy` = `calledObjArgTy`” for the overload, if the overload is for an instance member. When making this application, only solve type inference variables present in the `calledObjArgTy`. If any of these conversions fail, then skip the overload for the purposes of this rule
- Attempt to solve “`callerArgTy` = `(calledArgDomainTy1 -> … -> calledArgDomainTyN -> ?)`”. If this fails, then skip the overload for the purposes of this rule

14.4.2 Conditional Compilation of Member Calls

If a member definition has the `System.Diagnostics.Conditional` attribute, then any application of the member is adjusted as follows:

- The `Conditional("symbol")` attribute may apply to methods only.
- Methods that have the `Conditional` attribute must have return type `unit`. The return type may be checked either on use of the method or definition of the method.
- If *symbol* is not in the current set of conditional compilation symbols, the compiler eliminates application expressions that resolve to calls to members that have the `Conditional` attribute and ensures that arguments are not evaluated. Elimination of such expressions proceeds first with static members and then with instance members, as follows:
  - Static members: `Type.M(args)` → `()`
  - Instance members: `expr.M(args)` → `()`

14.4.3 Implicit Insertion of Flexibility for Uses of Functions and Members

At each use of a data constructor, named function, or member that forms an expression, flexibility is implicitly added to the expression. This flexibility is associated with the use of the function or member, according to the inferred type of the expression. The added flexibility allows the item to accept arguments that are statically known to be subtypes of argument types to a function without requiring explicit upcasts

The flexibility is added by adjusting each expression *expr* which represents a use of a function or member as follows:

- The type of the function or member is decomposed to the following form:

  ty_{11} * ... * ty_{1n} -> ... -> ty_{m1} * ... * ty_{mn} -> *rty*

- If the type does not decompose to this form, no flexibility is added.

- The positions ty_{ij} are called the "parameter positions" for the type. For each parameter position where ty_{ij} is not a sealed type, and is not a variable type, the type is replaced by a fresh type variable ty'_{ij} with a coercion constraint ty'_{ij} :> ty_{ij}.
- After the addition of flexibility, the expression elaborates to an expression of type

  ty'_{11} * ... * ty'_{1n} -> ... -> ty'_{m1} * ... * ty'_{mn} -> *rty*

  but otherwise is semantically equivalent to *expr* by creating an anonymous function expression and inserting appropariate coercions on arguments where necessary.

This means that F# functions whose inferred type includes an unsealed type in argument position may be passed subtypes when called, without the need for explicit upcasts. For example:

```
type Base() =
    member b.X = 1

type Derived(i : int) =
    inherit Base()
    member d.Y = i

let d = new Derived(7)

let f (b : Base) = b.X

// Call f: Base -> int with an instance of type Derived
let res = f d

// Use f as a first-class function value of type : Derived -> int
let res2 = (f : Derived -> int)
```

The F# compiler determines whether to insert flexibility after explicit instantiation, but before any arguments are checked. For example, given the following:

```
let M<'b>(c :'b, d :'b) = 1
let obj = new obj()
let str = ""
```

these expressions pass type-checking:

```
M<obj>(obj, str)
M<obj>(str, obj)
M<obj>(obj, obj)
M<obj>(str, str)
M(obj, obj)
M(str, str)
```

These expressions do not, because the target type is a variable type:

```
M(obj, str)
M(str, obj)
```

14.5 Constraint Solving

Constraint solving involves processing ("solving") non-primitive constraints to reduce them to primitive, normalized constraints on type variables. The F# compiler invokes constraint solving every time it adds a constraint to the set of current inference constraints at any point during type checking.

Given a type inference environment, the *normalized form* of constraints is a list of the following primitive constraints where *typar* is a type inference variable:

```
typar :> type
typar : null
(type or ... or type) : (member-sig)
typar : (new : unit -> 'T)
typar : struct
typar : unmanaged
typar : comparison
typar : equality
typar : not struct
typar : enum<type>
typar : delegate<type, type>
```

Each newly introduced constraint is solved as described in the following sections.

14.5.1 Solving Equational Constraints

New equational constraints in the form `typar = type` or `type = typar`, where *typar* is a type inference variable, cause *type* to replace *typar* in the constraint problem; *typar* is eliminated. Other constraints that are associated with *typar* are then no longer primitive and are solved again.

New equational constraints of the form $type\langle tyarg_{11}, ..., tyarg_{1n}\rangle = type\langle tyarg_{21}, ..., tyarg_{2n}\rangle$ are reduced to a series of constraints $tyarg_{1i} = tyarg_{2i}$ on identical named types and solved again.

14.5.2 Solving Subtype Constraints

Primitive constraints in the form `typar :> obj` are discarded.

New constraints in the form $type_1$ `:>` $type_2$, where $type_2$ is a sealed type, are reduced to the constraint $type_1 = type_2$ and solved again.

New constraints in either of these two forms are reduced to the constraints $tyarg_{11} = tyarg_{21}$ ... $tyarg_{1n} = tyarg_{2n}$ and solved again:

$$type\langle tyarg_{11}, ..., tyarg_{1n}\rangle :> type\langle tyarg_{21}, ..., tyarg_{2n}\rangle$$
$$type\langle tyarg_{11}, ..., tyarg_{1n}\rangle = type\langle tyarg_{21}, ..., tyarg_{2n}\rangle$$

> Note: F# generic types do not support covariance or contravariance. That is, although single-dimensional array types in the CLI are effectively covariant, F# treats these types as invariant during constraint solving. Likewise, F# considers CLI delegate types as invariant and ignores any CLI variance type annotations on generic interface types and generic delegate types.

New constraints of the form $type_1\langle tyarg_{11}, \ldots, tyarg_{1n}\rangle$ `:>` $type_2\langle tyarg_{21}, \ldots, tyarg_{2n}\rangle$ where $type_1$ and $type_2$ are hierarchically related, are reduced to an equational constraint on two instantiations of $type_2$ according to the subtype relation between $type_1$ and $type_2$, and solved again.

For example, if `MySubClass<'T>` is derived from `MyBaseClass<list<'T>>`, then the constraint

```
MySubClass<'T> :> MyBaseClass<int>
```

is reduced to the constraint

```
MyBaseClass<list<'T>> :> MyBaseClass<list<int>>
```

and solved again, so that the constraint `'T = int` will eventually be derived.

> **Note**: Subtype constraints on single-dimensional array types `ty[] :> ty` are reduced to residual constraints, because these types are considered to be subtypes of `System.Array`, `System.Collections.Generic.IList<'T>`, `System.Collections.Generic.ICollection<'T>`, and `System.Collections.Generic.IEnumerable<'T>`. Multidimensional array types `ty[,…,]` are also subtypes of `System.Array`.
>
> Types from other CLI languages may, in theory, support multiple instantiations of the same interface type, such as `C : I<int>, I<string>`. Consequently, it is more difficult to solve a constraint such as `C :> I<'T>`. Such constraints are rarely used in practice in F# coding. To solve this constraint, the F# compiler reduces it to a constraint `C :> I<'T>`, where `I<'T>` is the first interface type that occurs in the tree of supported interface types, when the tree is ordered from most derived to least derived, and iterated left-to-right in the order of the declarations in the CLI metadata.
>
> The F# compiler ignores CLI variance type annotations on interfaces.

New constraints of the form *type* `:> 'b` are solved again as *type* `= 'b`.

> **Note**: Such constraints typically occur only in calls to generic code from other CLI languages where a method accepts a parameter of a “naked” variable type—for example, a C# 2.0 function with a signature such as `T Choose<'T>(T x, T y)`.

14.5.3 Solving Nullness, Struct, and Other Simple Constraints

New constraints in any of the following forms, where *type* is not a variable type, are reduced to further constraints:

```
type : null
type : (new : unit -> 'T)
```

```
type : struct
type : not struct
type : enum<type>
type : delegate<type, type>
type : unmanaged
```

The compiler then resolves them according to the requirements for each kind of constraint listed in §5.2 and §5.4.8.

14.5.4 Solving Member Constraints

New constraints in the following form) are solved as *member constraints* (§5.2.3):

```
(type1 or ... or typen) : (member-sig)
```

A member constraint is satisfied if one of the types in the *support set* `type`$_1$ `...` `type`$_n$ satisfies the member constraint. A static type `type` satisfies a member constraint in the form `(static`$_{opt}$ `member ident : arg-type`$_1$ `* ... * arg-type`$_n$ `-> ret-type)` if all of the following are true:

- `type` is a named type whose type definition contains the following member, which takes *n* arguments:

  `static`$_{opt}$ `member ident : formal-arg-type`$_1$ `* ... * formal-arg-type`$_n$ `-> ret-type`

- The `type` and the constraint are both marked `static` or neither is marked `static` .
- The assertion of type inference constraints on the arguments and return types does not result in a type inference error.

As mentioned in §5.2.3, a type variable may not be involved in the support set of more than one member constraint that has the same name, staticness, argument arity, and support set. If a type variable is in the support set of more than one such constraint, the argument and return types are themselves constrained to be equal.

14.5.4.1 Simulation of Solutions for Member Constraints

Certain types are assumed to implicitly define static members even though the actual CLI metadata for types does not define these operators. This mechanism is used to implement the extensible conversion and math functions of the F# library including `sin`, `cos`, `int`, `float`, `(+)`, and `(-)`. The following table shows the static members that are implicitly defined for various types.

Type	Implicitly defined static members

Type	Implicitly defined static members
Integral types: byte, sbyte, int16, uint16, int32, uint32, int64, uint64, nativeint, unativeint	op_BitwiseAnd, op_BitwiseOr, op_ExclusiveOr, op_LeftShift, op_RightShift, op_UnaryPlus, op_UnaryNegation, op_Increment, op_Decrement, op_LogicalNot, op_OnesComplement op_Addition, op_Subtraction, op_Multiply, op_Division, op_Modulus, op_UnaryPlus op_Explicit: takes the type as an argument and returns byte, sbyte, int16, uint16, int32, uint32, int64, uint64, float32, float, decimal, nativeint, or unativeint
Signed integral CLI types: sbyte, int16, int32, int64 and nativeint	op_UnaryNegation Sign Abs
Floating-point CLI types: float32 and float	Sin, Cos, Tan, Sinh, Cosh, Tanh, Atan, Acos, Asin, Exp, Ceiling, Floor, Round, Log10, Log, Sqrt, Atan2, Pow op_Addition, op_Subtraction, op_Multiply, op_Division, op_Modulus, op_UnaryPlus op_UnaryNegation Sign Abs op_Explicit: takes the type as an argument and returns byte, sbyte, int16, uint16, int32, uint32, int64, uint64, float32, float, decimal, nativeint, or unativeint
decimal type **Note**: The decimal type is included only for the Sign static member. This is deliberate: in the CLI, System.Decimal includes the definition of static members such as op_Addition and the F# compiler does not need to simulate the existence of these methods.	Sign
String type string	op_Addition op_Explicit: takes the type as an argument and return byte, sbyte, int16, uint16, int32, uint32, int64, uint64, float32, float or decimal.

14.5.5 Over-constrained User Type Annotations

An implementation of F# must give a warning if a type inference variable that results from a user type annotation is constrained to be a type other than another type inference variable. For example, the following results in a warning because 'T has been constrained to be precisely string:

```
let f (x:'T) = (x:string)
```

During the resolution of overloaded methods, resolutions that do not give such a warning are preferred over resolutions that do give such a warning.

14.6 Checking and Elaborating Function, Value, and Member Definitions

This section describes how function, value, and member definitions are checked, generalized, and elaborated. These definitions occur in the following contexts:

- Module declarations
- Class type declarations
- Expressions
- Computation expressions

Recursive definitions can also occur in each of these locations. In addition, member definitions in a mutually recursive group of type declarations are implicitly recursive.

Each definition is one of the following:

- A function definition :

  `inline`$_{opt}$ *ident*$_1$ *pat*$_1$ ... *pat*$_n$:$_{opt}$ *return-type*$_{opt}$ = *rhs-expr*

- A value definition, which defines one or more values by matching a pattern against an expression:

  `mutable`$_{opt}$ *pat* :$_{opt}$ *type*$_{opt}$ = *rhs-expr*

- A member definition:

  `static`$_{opt}$ `member` *ident*$_{opt}$ *ident* *pat*$_1$ ... *pat*$_n$ = *expr*

For a function, value, or member definition in a class:

1. If the definition is an instance function, value or member, checking uses an environment to which both of the following have been added:
   - The instance variable for the class, if one is present.
   - All previous function and value definitions for the type, whether static or instance.
2. If the definition is static (that is, a static function, value or member defeinition), checking uses an environment to which all previous static function, value, and member definitions for the type have been added.

14.6.1 Ambiguities in Function and Value Definitions

In one case, an ambiguity exists between the syntax for function and value definitions. In particular, *ident pat = expr* can be interpreted as either a function or value definition. For example, consider the following:

```
type OneInteger = Id of int
let Id x = x
```

In this case, the ambiguity is whether `Id x` is a pattern that matches values of type `OneInteger` or is the function name and argument list of a function called `Id`. In F# this ambiguity is always resolved as a function definition. In this case, to make a value definition, use the following syntax in which the ambiguous pattern is enclosed in parentheses:

```
let v = if 3 = 4 then Id "yes" else Id "no"
let (Id answer) = v
```

14.6.2 Mutable Value Definitions

Value definitions may be marked as `mutable`. For example:

```
let mutable v = 0
while v < 10 do
    v <- v + 1
    printfn "v = %d" v
```

These variables are subject to the same restrictions as values of type `byref<_>` (§14.9), and are similarly implicitly dereferenced.

14.6.3 Processing Value Definitions

A value definition *pat* = *rhs-expr* with optional pattern type *type* is processed as follows:

1. The pattern *pat* is checked against a fresh initial type *ty* (or *type* if such a type is present). This check results in zero or more identifiers $ident_1$ ... $ident_m$, each of type ty_1 ... ty_m.

2. The expression *rhs-expr* is checked against initial type *ty*, resulting in an elaborated form *expr*.

3. Each $ident_i$ (of type ty_i) is then generalized (§14.6.7) and yields generic parameters <$typars_j$>.

4. The following rules are checked:

   - All $ident_j$ must be distinct.
   - Value definitions may not be `inline`.

5. If *pat* is a single value pattern, the resulting elaborated definition is:

   ident <$typars_1$> = *expr*
   body-expr

6. Otherwise, the resulting elaborated definitions are the following, where *tmp* is a fresh identifier and each $expr_i$ results from the compilation of the pattern *pat* (§7) against input *tmp*.

   tmp <$typars_1$... $typars_n$> = *expr*
   $ident_1$ <$typars_1$> = $expr_1$
   ...
   $ident_n$ <$typars_n$> = $expr_n$

14.6.4 Processing Function Definitions

A function definition `ident1 pat1 ... patn = rhs-expr` is processed as follows:

1. If `ident1` is an active pattern identifier then active pattern result tags are added to the environment (§10.2.4).

2. The expression `(fun pat1 ... patn : return-type -> rhs-expr)` is checked against a fresh initial type `ty1` and reduced to an elaborated form `expr1`. The return type is omitted if the definition does not specify it.

3. The `ident1` (of type `ty1`) is then generalized (§14.6.7) and yields generic parameters `<typars1>`.

4. The following rules are checked:

   - Function definitions may not be `mutable`. Mutable function values should be written as follows:
     `let mutable f = (fun args -> ...)`
   - The patterns of functions may not include optional arguments (§8.13.6).

5. The resulting elaborated definition is:

   `ident1<typars1> = expr1`

14.6.5 Processing Recursive Groups of Definitions

A group of functions and values may be declared recursive through the use of `let rec`. Groups of members in a recursive set of type definitions are also implicitly recursive. In this case, the defined values are available for use within their own definitions—that is, within all the expressions on the right-hand side of the definitions.

For example:

```
let rec twoForward count =
    printfn "at %d, taking two steps forward" count
    if count = 1000 then "got there!"
    else oneBack (count + 2)
and oneBack count =
    printfn "at %d, taking one step back " count
    twoForward (count - 1)
```

When one or more definitions specifies a value, the recursive expressions are analyzed for safety (§14.6.6). This analysis may result in warnings—including some reported at compile time—and runtime checks.

Within recursive groups, each definition in the group is checked (§14.6.7) and then the definitions are generalized incrementally. In addition, any use of an ungeneralized recursive definition results in immediate constraints on the recursively defined construct. For example, consider the following declaration:

```
let rec countDown count x =
```

```
        if count > 0 then
            let a = countDown (count - 1) 1         // constrains "x" to be
    of type int
            let b = countDown (count - 1) "Hello"  // constrains "x" to be
    of type string
            a + b
        else
            1
```

In this example, the definition is not valid because the recursive uses of `f` result in inconsistent constraints on `x`.

If a definition has a full signature, early generalization applies and recursive calls at different types are permitted (§14.6.7). For example:

```
module M =
    let rec f<'T> (x:'T) : 'T =
        let a = f 1
        let b = f "Hello"
        x
```

In this example, the definition is valid because `f` is subject to early generalization, and so the recursive uses of `f` do not result in inconsistent constraints on `x`.

14.6.6 Recursive Safety Analysis

A set of recursive definitions may include value definitions. For example:

```
type Reactor = React of (int -> React) * int

let rec zero = React((fun c  -> zero), 0)

let const n =
    let rec r = React((fun c  -> r), n)
    r
```

Recursive value definitions may result in invalid recursive cycles, such as the following:

```
let rec x = x + 1
```

The *Recursive Safety Analysis* process partially checks the safety of these definitions and convert thems to a form that uses lazy initialization, where runtime checks are inserted to check initialization.

A right-hand side expression is *safe* if it is any of the following:

- A function expression, including those whose bodies include references to variables that are defined recursively.
- An object expression that implements an interface, including interfaces whose member bodies include references to variables that are being defined recursively.
- A `lazy` delayed expression.

- A record, tuple, list, or data construction expression whose field initialization expressions are all safe.
- A value that is not being recursively bound.
- A value that is being recursively bound and appears in one of the following positions:
  - As a field initializer for a field of a record type where the field is marked `mutable`.
  - As a field initializer for an immutable field of a record type that is defined in the current assembly.

    If record fields contain recursive references to values being bound, the record fields must be initialized in the same order as their declared type, as described later in this section.
- Any expression that refers only to earlier variables defined by the sequence of recursive definitions.

Other right-hand side expressions are elaborated by adding a new definition. If the original definition is

```
u = expr
```

then a fresh value (say `v`) is generated with the definition:

```
v = lazy expr
```

and occurrences of the original variable *u* on the right-hand side are replaced by `Lazy.force v`. The following definition is then added at the end of the definition list:

```
u = v.Force()
```

Note: This specification implies that recursive value definitions are executed as an *initialization graph* of delayed computations. Some recursive references may be checked at runtime because the computations that are involved in evaluating the definitions might actually execute the delayed computations. The F# compiler gives a warning for recursive value definitions that might involve a runtime check. If runtime self-reference does occur then an exception will be raised.

Recursive value definitions that involve computation are useful when defining objects such as forms, controls, and services that respond to various inputs. For example, GUI elements that store and retrieve the state of the GUI elements as part of their specification typically involve recursive value definitions. A simple example is the following menu item, which prints out part of its state when invoked:

```
open System.Windows.Form
let rec menuItem : MenuItem =
    new MenuItem("&Say Hello",
                 new EventHandler(fun sender e ->
                      printfn "Text = %s" menuItem.Text),
                 Shortcut.CtrlH)
```

This code results in a compiler warning because, in theory, the `new MenuItem(...)` constructor might evaluate the callback as part of the construction process. However, because the `System.Windows.Forms` library is well designed, in this example this does not happen in practice, and so the warning can be suppressed or ignored by using compiler options.

The F# compiler performs a simple approximate static analysis to determine whether immediate cyclic dependencies are certain to occur during the evaluation of a set of recursive value definitions. The compiler creates a graph of definite references and reports an error if such a dependency cycle exists. All references within function expressions, object expressions, or delayed expressions are assumed to be indefinite, which makes the analysis an under-approximation. As a result, this check catches naive and direct immediate recursion dependencies, such as the following:

```
let rec A = B  + 1
and B = A + 1
```

Here, a compile-time error is reported. This check is necessarily approximate because dependencies under function expressions are assumed to be delayed, and in this case the use of a lazy initialization means that runtime checks and forces are inserted.

Note: In F# 3.1 this check does not apply to value definitions that are generic through generalization because a generic value definition is not executed immediately, but is instead represented as a generic method. For example, the following value definitions are generic because each right-hand-side is generalizable:

```
let rec a = b
and b = a
```

In compiled code they are represented as a pair of generic methods, as if the code had been written as follows:

```
let rec a<'T>() = b<'T>()
and b<'T>() = a<'T>()
```

As a result, the definitions are not executed immediately unless the functions are called. Such definitions indicate a programmer error, because executing such generic, immediately recursive definitions results in an infinite loop or an exception. In practice these definitions only occur in pathological examples, because value definitions are generalizable only when the right-hand-side is very simple, such as a single value. Where this issue is a concern, type annotations can be added to existing value definitions to ensure they are not generic. For example:

```
let rec a : int = b
and b : int = a
```

In this case, the definitions are not generic. The compiler performs immediate dependency analysis and reports an error. In addition, record fields in recursive data expressions must be initialized in the order they are declared. For example:

```
type Foo = {
    x: int
    y: int
    parent: Foo option
    children: Foo list
}

let rec parent = { x = 0; y = 0; parent = None; children =
children }
and children = [{ x = 1; y = 1; parent = Some parent; children =
[] }]

printf "%A" parent
```

Here, if the order of the fields `x` and `y` is swapped, a type-checking error occurs.

14.6.7 Generalization

Generalization is the process of inferring a generic type for a definition where possible, thereby making the construct reusable with multiple different types. Generalization is applied by default at all function, value, and member definitions, except where listed later in this section. Generalization also applies to member definitions that implement generic virtual methods in object expressions.

Generalization is applied incrementally to items in a recursive group after each item is checked.

Generalization takes a set of ungeneralized but type-checked definitions *checked-defns* that form part of a recursive group, plus a set of unchecked definitions *unchecked-defns* that have not yet been checked in the recursive group, and an environment *env*. Generalization involves the following steps:

1. Choose a subset *generalizable-defns* of *checked-defns* to generalize.

   A definition can be generalized if its inferred type is closed with respect to any inference variables that are present in the types of the *unchecked-defns* that are in the recursive group and that are not yet checked or which, in turn, cannot be generalized. A greatest-fixed-point computation repeatedly removes definitions from the set of *checked-defns* until a stable set of generalizable definitions remains.

2. Generalize all type inference variables that are not otherwise ungeneralizable and for which any of the following is true:

   - The variable is present in the inferred types of one or more of *generalizable-defns*.
   - The variable is a type parameter copied from the enclosing type definition (for members and "let" definitions in classes).
   - The variable is explicitly declared as a generic parameter on an item.

The following type inference variables cannot be generalized:

- A type inference variable *^typar* that is part of the inferred or declared type of a definition, unless the definition is marked `inline`.
- A type inference variable in an inferred type in the *ExprItems* or *PatItems* tables of *env*, or in an inferred type of a module in the *ModulesAndNamespaces* table in *env*.
- A type inference variable that is part of the inferred or declared type of a definition in which the elaborated right-hand side of the definition is not a generalizable expression, as described later in this section.
- A type inference variable that appears in a constraint that itself refers to an ungeneralizable type variable.

Generalizable type variables are computed by a greatest-fixed-point computation, as follows:

1. Start with all variables that are candidates for generalization.
2. Determine a set of variables *U* that cannot be generalized because they are free in the environment or present in ungeneralizable definitions.
3. Remove the variables in *U* from consideration.
4. Add to *U* any inference variables that have a constraint that involves a variable in *U*.
5. Repeat steps 2 through 4.

Informally, generalizable expressions represent a subset of expressions that can be freely copied and instantiated at multiple types without affecting the typical semantics of an F# program. The following expressions are generalizable:

- A function expression
- An object expression that implements an interface
- A delegate expression
- A “let” definition expression in which both the right-hand side of the definition and the body of the expression are generalizable
- A “let rec” definition expression in which the right-hand sides of all the definitions and the body of the expression are generalizable
- A tuple expression, all of whose elements are generalizable
- A record expression, all of whose elements are generalizable, where the record contains no mutable fields
- A union case expression, all of whose arguments are generalizable
- An exception expression, all of whose arguments are generalizable
- An empty array expression
- A simple constant expression
- An application of a type function that has the `GeneralizableValue` attribute.

Explicit type parameter definitions on value and member definitions can affect the process of type inference and generalization. In particular, a declaration that includes explicit generic parameters will not be generalized beyond those generic parameters. For example, consider this function:

```
let f<'T> (x : 'T) y = x
```

During type inference, this will result in a function of the following type, where `'_b` is a type inference variable that is yet to be resolved.

```
f<'T> : 'T -> '_b -> '_b
```

To permit generalization at these definitions, either remove the explicit generic parameters (if they can be inferred), or use the required number of parameters, as the following example shows:

```
let throw<'T,'U> (x:'T) (y:'U) = x
```

14.6.8 Condensation of Generalized Types

After a function or member definition is generalized, its type is condensed by removing generic type parameters that apply subtype constraints to argument positions. (The removed flexibility is implicitly reintroduced at each use of the defined function; see §14.4.3).

Condensation decomposes the type of a value or member to the following form:

$$ty_{11} * \ldots * ty_{1n} \to \ldots \to ty_{m1} * \ldots * ty_{mn} \to rty$$

The positions ty_{ij} are called the parameter positions for the type.

Condensation applies to a type parameter `'a` if all of the following are true:

- `'a` is not an explicit type parameter.
- `'a` occurs at exactly one ty_{ij} parameter position.
- `'a` has a single coercion constraint `'a :> ty` and no other constraints. However, one additional nullness constraint is permitted if *ty* satisfies the nullness constraint.
- `'a` does not occur in any other ty_{ij}, nor in *rty*.
- `'a` does not occur in the constraints of any condensed `typar`.

Condensation is a greatest-fixed-point computation that initially assumes all generalized type parameters are condensed, and then progressively removes type parameters until a minimal set remains that satisfies the above rules.

The compiler removes all condensed type parameters and replaces them with their subtype constraint *ty*. For example:

```
let F x = (x :> System.IComparable).CompareTo(x)
```

After generalization, the function is inferred to have the following type:

```
F : 'a -> int when 'a :> System.IComparable
```

In this case, the actual inferred, generalized type for `F` is condensed to:

```
F : System.IComparable -> R
```

Condensation does not apply to arguments of unconstrained variable type. For example:

```
let ignore x = ()
```

with type

```
ignore: 'a -> unit
```

In particular, this is not condensed to

```
ignore: obj -> unit
```

In rare cases, condensation affects the points at which value types are boxed. In the following example, the value `3` is now boxed at uses of the function:

```
F 3
```

If a function is not generalized, condensation is not applied. For example, consider the following:

```
let test1 =
    let ff = Seq.map id >> Seq.length
    (ff [1], ff [| 1 |])  // error here
```

In this example, `ff` is not generalized, because it is not defined by using a generalizable expression—computed functions such as `Seq.map id >> Seq.length` are not generalizable. This means that its inferred type, after processing the definition, is

```
F : '_a -> int when '_a :> seq<'_b>
```

where the type variables are not generalized and are unsolved inference variables. The application of `ff` to `[1]` equates `'a` with `int list`, making the following the type of `F`:

```
F : int list -> int
```

The application of `ff` to an array type then causes an error. This is similar to the error returned by the following:

```
let test1 =
    let ff = Seq.map id >> Seq.length
    (ff [1], ff ["one"])  // error here
```

Again, `ff` is not generalized, and its use with arguments of type `int list` and `string list` is not permitted.

14.7 Dispatch Slot Inference

The F# compiler applies *Dispatch Slot Inference* to object expressions and type definitions before it processes their members. For both object expressions and type definitions, the following are input to Dispatch Slot Inference:

- A type ty_0 that is being implemented.
- A set of members `override x.M(`$arg_1...arg_N$`)`.
- A set of additional interface types ty_1 ... ty_n.
- A further set of members `override x.M(`$arg_1...arg_N$`)` for each ty_i.

Dispatch slot inference associates each member with a unique abstract member or interface member that the collected types ty_i define or inherit.

The types ty_0 ... ty_n together imply a collection of *required types* R, each of which has a set of *required dispatch slots* Slots_R of the form `abstract M :` $aty_1...aty_N$ `->` aty_{rty}. Each dispatch slot is placed under the *most-specific* ty_i relevant to that dispatch slot. If there is no most-specific type for a dispatch slot, an error occurs.

For example, assume the following definitions:

```
type IA = interface abstract P : int end
type IB = interface inherit IA end
type ID = interface inherit IB end
```

With these definitions, the following object expression is legal. Type `IB` is the most-specific implemented type that encompasses `IA`, and therefore the implementation mapping for `P` must be listed under `IB`:

```
let x = { new ID
          interface IB with
              member x.P = 2 }
```

But given:

```
type IA = interface abstract P : int end
type IB = interface inherit IA end
type IC = interface inherit IB end
type ID = interface inherit IB inherit IC end
```

then the following object expression causes an error, because both `IB` and `IC` include the interface `IA`, and consequently the implementation mapping for `P` is ambiguous.

```
let x = { new ID
          interface IB with
              member x.P = 2
          interface IC with
              member x.P = 2 }
```

The ambiguity can be resolved by explicitly implementing interface `IA`.

After dispatch slots are assigned to types, the compiler tries to associate each member with a dispatch slot based on name and number of arguments. This is called *dispatch slot inference,* and it proceeds as follows:

- For each `member x.M(`arg_1`...`arg_N`)` in type ty_i, attempt to find a single dispatch slot in the form

  `abstract M :` aty_1`...`aty_N `->` rty

  with name `M`, argument count `N`, and most-specific implementing type ty_i.

  - To determine the argument counts, analyze the syntax of patterns and look specifically for tuple and unit patterns. Thus, the following members have argument count 1, even though the argument type is `unit`:

    ```
    member obj.ToString(() | ()) = ...
    member obj.ToString(():unit) = ...
    member obj.ToString(_:unit) = ...
    ```

  - A member may have a return type, which is ignored when determining argument counts:

    ```
    member obj.ToString() : string = ...
    ```

For example, given

```
let obj1 =
   { new System.Collections.Generic.IComparer<int> with
        member x.Compare(a,b) = compare (a % 7) (b % 7) }
```

the types of `a` and `b` are inferred by looking at the signature of the implemented dispatch slot, and are hence both inferred to be `int`.

14.8 Dispatch Slot Checking

Dispatch Slot Checking is applied to object expressions and type definitions to check consistency properties, such as ensuring that all abstract members are implemented.

After the compiler checks all bodies of all methods, it checks that a one-to-one mapping exists between dispatch slots and implementing members based on exact signature matching.

The interface methods and abstract method slots of a type are collectively known as *dispatch slots*. Each object expression and type definition results in an elaborated *dispatch map*. This map is keyed by dispatch slots, which are qualified by the declaring type of the slot. This means that a type that supports two interfaces `I` and `I2`, both of which contain the method `m`, may supply different implementations for `I.m()` and `I2.m()`.

The construction of the dispatch map for any particular type is as follows:

- If the type definition or extension has an implementation of an interface, mappings are added for each member of the interface,
- If the type definition or extension has a `default` or `override` member, a mapping is added for the associated abstract member slot.

14.9 Byref Safety Analysis

Byref arguments are pointers that can be stack-bound and are used to pass values by reference to procedures in CLI languages, often to simulate multiple return values. Byref pointers are not often used in F#; more typically, tuple values are used for multiple return values. However, a byref value can result from calling or overriding a CLI method that has a signature that involves one or more byref values.

To ensure the safety of byref arguments, the following checks are made:

- Byref types may not be used as generic arguments.
- Byref values may not be used in any of the following:
  - The argument types or body of function expressions `(fun … -> …)`.
  - The member implementations of object expressions.
  - The signature or body of let-bound functions in classes.
  - The signature or body of let-bound functions in expressions.

Note that function expressions occur in:

- The elaborated form of sequence expressions.

- The elaborated form of computation expressions.
- The elaborated form of partial applications of module-bound functions and members.

In addition:

- A generic type cannot be instantiated by a byref type.
- An object field cannot have a byref type.
- A static field or module-bound value cannot have a byref type.

As a result, a byref-typed expression can occur only in these situations:

- As an argument to a call to a module-defined function or class-defined function.
- On the right-hand-side of a value definition for a byref-typed local.

These restrictions also apply to uses of the prefix && operator for generating native pointer values.

14.10 Arity Inference

During checking, members within types and function definitions within modules are inferred to have an *arity*. An arity includes both of the following:

- The number of iterated (curried) arguments n
- A tuple length for these arguments $[A_1;...;A_n]$. A tuple length of zero indicates that the corresponding argument is of type `unit`.

Arities are inferred as follows. A function definition of the following form is given arity $[A_1;...;A_n]$, where each A_i is derived from the tuple length for the final inferred types of the patterns:

$$\texttt{let}\ ident\ pat_1\ \ldots\ pat_n = \ldots$$

For example, the following is given arity [1; 2]:

```
let f x (y,z) = x + y + z
```

Arities are also inferred from function expressions that appear on the immediate right of a value definition. For example, the following has an arity of [1]:

```
let f = fun x -> x + 1
```

Similarly, the following has an arity of [1;1]:

```
let f x = fun y -> x + y
```

Arity inference is applied partly to help define the elaborated form of a function definition. This is the form that other CLI languages see. In particular:

- A function value F in a module that has arity $[A_1;...;A_n]$ and the type
  $ty_{1,1} * \ldots * ty_{1,A1} \rightarrow \ldots \rightarrow ty_{n,1} * \ldots * ty_{n,An} \rightarrow rty$

elaborates to a CLI static method definition with signature
rty F(*ty*$_{1,1}$, ..., *ty*$_{1,A1}$, ..., *ty*$_{n,1}$, ..., *ty*$_{n,An}$).

- F# instance (respectively static) methods that have arity [A$_1$;...;A$_n$] and type
ty$_{1,1}$ * ... * *ty*$_{1,A1}$ -> ... -> *ty*$_{n,1}$ * ... * *ty*$_{n,An}$ -> *rty*
elaborate to a CLI instance (respectively static) method definition with signature
rty F(*ty*$_{1,1}$, ..., *ty*$_{1,A1}$), subject to the syntactic restrictions that result from the patterns that define the member, as described later in this section.

For example, consider a function in a module with the following definition:

```
let AddThemUp x (y, z) = x + y + z
```

This function compiles to a CLI static method with the following C# signature:

```
int AddThemUp(int x, int y, int z);
```

Arity inference applies differently to function and member definitions. Arity inference on function definitions is fully type-directed. Arity inference on members is limited if parentheses or other patterns are used to specify the member arguments. For example:

```
module Foo =
    // compiles as a static method taking 3 arguments
    let test1 (a1: int, a2: float, a3: string) = ()

    // compiles as a static method taking 3 arguments
    let test2 (aTuple : int * float * string) = ()

    // compiles as a static method taking 3 arguments
    let test3 ( (aTuple : int * float * string) ) = ()

    // compiles as a static method taking 3 arguments
    let test4 ( (a1: int, a2: float, a3: string) ) = ()

    // compiles as a static method taking 3 arguments
    let test5 (a1, a2, a3 : int *  float * string) = ()


type Bar() =
    // compiles as a static method taking 3 arguments
    static member Test1 (a1: int, a2: float, a3: string) = ()

    // compiles as a static method taking 1 tupled argument
    static member Test2 (aTuple : int * float * string) = ()

    // compiles as a static method taking 1 tupled argument
    static member Test3 ( (aTuple : int * float * string) ) = ()

    // compiles as a static method taking 1 tupled argument
    static member Test4 ( (a1: int, a2: float, a3: string) ) = ()
```

```
// compiles as a static method taking 1 tupled argument
static member Test5 (a1, a2, a3 : int * float * string) = ()
```

14.11 Additional Constraints on CLI Methods

F# treats some CLI methods and types specially, because they are common in F# programming and cause extremely difficult-to-find bugs. For each use of the following constructs, the F# compiler imposes additional *ad hoc* constraints:

`x.Equals(yobj)` requires type `ty : equality` for the static type of `x`

`x.GetHashCode()` requires type `ty : equality` for the static type of `x`

`new Dictionary<A,B>()` requires `A : equality`, for any overload that does not take an `IEqualityComparer<T>`

No constraints are added for the following operations. Consider writing wrappers around these functions to improve the type safety of the operations.

`System.Array.BinarySearch<T>(array,value)` requiring `C : comparison`, for any overload that does not take an `IComparer<T>`

`System.Array.IndexOf` requiring `C : equality`

`System.Array.LastIndexOf(array,T)` requiring `C : equality`

`System.Array.Sort<'T>(array)` requiring `C : comparison`, for any overload that does not take an `IEqualityComparer<T>`

`new SortedList<A,B>()` requiring `A : comparison`, for any overload that does not take an `IEqualityComparer<T>`

`new SortedDictionary<A,B>()` requiring `C : comparison`, for any overload that does not take an `IEqualityComparer<_>`

15. Lexical Filtering

15.1 Lightweight Syntax

F# supports lightweight syntax, in which whitespace makes indentation significant.

The lightweight syntax option is a conservative extension of the explicit language syntax, in the sense that it simply lets you leave out certain tokens such as `in` and `;;` because the parser takes indentation into account. Indentation can make a surprising difference in the readability of code. Compiling your code with the indentation-aware syntax option is useful even if you continue to use explicit tokens, because the compiler reports many indentation problems with your code and ensures a regular, clear formatting style.

In the processing of lightweight syntax, comments are considered pure whitespace. This means that the compiler ignores the indentation position of comments. Comments act as if they were replaced by whitespace characters. Tab characters cannot be used in F# files.

15.1.1 Basic Lightweight Syntax Rules by Example

The basic rules that the F# compiler applies when it processes lightweight syntax are shown below, illustrated by example.

;; delimiter

When the lightweight syntax option is enabled, top level expressions do not require the `;;` delimiter because every construct that starts in the first column is implicitly a new declaration. The `;;` delimiter is still required to terminate interactive entries to fsi.exe, but not when using F# Interactive from Visual Studio.

Lightweight Syntax

```
printf "Hello"
printf "World"
```

Normal Syntax

```
printf "Hello";;
printf "World";;
```

in keyword

When the lightweight syntax option is enabled, the `in` keyword is optional. The token after the "=" in a 'let' definition begins a new block, and the pre-parser inserts an implicit separating `in` token between each definition that begins at the same column as that token.

Lightweight Syntax

```
let SimpleSample() =
    let x = 10 + 12 - 3
    let y = x * 2 + 1
    let r1,r2 = x/3, x%3
    (x,y,r1,r2)
```

Normal Syntax

```
#indent "off"
let SimpleSample() =
    let x = 10 + 12 - 3 in
    let y = x * 2 + 1 in
    let r1,r2 = x/3, x%3 in
    (x,y,r1,r2)
```

done keyword

When the lightweight syntax option is enabled, the `done` keyword is optional. Indentation establishes the scope of structured constructs such as `match`, `for`, `while` and `if`/`then`/`else`.

Lightweight Syntax

```
let FunctionSample() =
    let tick x = printfn "tick %d" x
    let tock x = printfn "tock %d" x
    let choose f g h x =
        if f x then g x else h x
    for i = 0 to 10 do
        choose (fun n -> n%2 = 0) tick tock i
```

Normal Syntax

```
#indent "off"
let FunctionSample() =
    let tick x = printfn "tick %d" x in
    let tock x = printfn "tock %d" x in
    let choose f g h x =
        if f x then g x else h x in
    for i = 0 to 10 do
```

```
    printfn "done!"
```

```
        choose (fun n -> n%2 = 0) tick tock i
    done;
    printfn "done!"
```

if/then/else Scope

When the lightweight syntax option is enabled, the scope of `if`/`then`/`else` is implicit from indentation. Without the lightweight syntax option, `begin`/`end` or parentheses are often required to delimit such constructs.

Lightweight Syntax

```
let ArraySample() =
    let numLetters = 26
    let results = Array.create numLetters 0
    let data = "The quick brown fox"
    for i = 0 to data.Length - 1 do
        let c = data.Chars(i)
        let c = Char.ToUpper(c)
        if c >= 'A' && c <= 'Z' then
            let i = Char.code c - Char.code 'A'
            results.[i] <- results.[i] + 1
    printfn "done!"
```

Normal Syntax

```
#indent "off"
let ArraySample() =
    let numLetters = 26 in
    let results = Array.create numLetters 0 in
    let data = "The quick brown fox" in
    for i = 0 to data.Length - 1 do
        let c = data.Chars(i) in
        let c = Char.ToUpper(c)  in
        if c >= 'A' && c <= 'Z' then begin
            let i = Char.code c - Char.code 'A' in
            results.[i] <- results.[i] + 1
        end
    done;
    printfn "done!"
```

15.1.2 Inserted Tokens

Lexical filtering inserts the following hidden tokens :

```
token $in   // Note: also called ODECLEND
token $done // Note: also called ODECLEND
token $begin   // Note: also called OBLOCKBEGIN
token $end     // Note: also called OEND, OBLOCKEND and
ORIGHT_BLOCK_END
token $sep     // Note: also called OBLOCKSEP
token $app     // Note: also called HIGH_PRECEDENCE_APP
token $tyapp   // Note: also called HIGH_PRECEDENCE_TYAPP
```

Note: The following tokens are also used in the Microsoft F# implementation. They are translations of the corresponding input tokens and help provide better error messages for lightweight syntax code:

```
tokens $let $use  $let! $use! $do $do! $then $else $with
$function $fun
```

15.1.3 Grammar Rules Including Inserted Tokens

Additional grammar rules take into account the token transformations performed by lexical filtering:

```
expr +:=
     | let function-defn $in expr
     | let value-defn $in expr
     | let rec function-or-value-defns $in expr
     | while expr do expr $done
     | if expr then $begin expr $end
     | for pat in expr do expr $done
     | for expr to expr do expr $done
     | try expr $end with expr $done
     | try expr $end finally expr $done
```

```
    | expr $app expr                // equivalent to "expr(expr)"
    | expr $sep expr                // equivalent to "expr; expr"
    | expr $tyapp < types >// equivalent to "expr<types>"
    | $begin expr $end      // equivalent to "expr"

elif-branch +:=
    | elif expr then $begin expr $end

else-branch +:=
    | else $begin expr $end

class-or-struct-type-body +:=
    | $begin class-or-struct-type-body $end
                              // equivalent to class-or-struct-type-
body

module-elems +:=
  | $begin module-elem ... module-elem $end

module-abbrev +:=
  | module ident = $begin long-ident $end

module-defn +:=
  | module ident = $begin module-defn-body $end

module-signature-elements +:=
  | $begin module-signature-element ... module-signature-element $end

module-signature +:=
  | module ident = $begin module-signature-body $end
```

15.1.4 Offside Lines

Lightweight syntax is sometimes called the "offside rule". In F# code, offside lines occur at column positions. For example, an `=` token associated with `let` introduces an offside line at the column of the first non-whitespace token after the `=` token.

Other structured constructs also introduce offside lines at the following places:

- The column of the first token after `then` in an `if`/`then`/`else` construct.
- The column of the first token after `try`, `else`, `->`, `with` (in a `match`/`with` or `try`/`with`), or `with` (in a type extension).
- The column of the first token of a `(`, `{` or `begin` token.
- The start of a `let`, `if` or `module` token.

Here are some examples of how the offside rule applies to F# code. In the first example, `let` and `type` declarations are not properly aligned, which causes F# to generate a warning.

```
// "let" and "type" declarations in
// modules must be precisely aligned.
let x = 1
```

```
     let y = 2  <-- unmatched 'let'
    let z = 3   <-- warning FS0058: possible
                    incorrect indentation: this token is offside of
                    context at position (2:1)
```

In the second example, the `|` markers in the match patterns do not align properly:

```
// The "|" markers in patterns must align.
// The first "|" should always be inserted.
let f () =
    match 1+1 with
    | 2 -> printf "ok"
  | _ -> failwith "no!"   <-- syntax error
```

15.1.5 The Pre-Parse Stack

F# implements the lightweight syntax option by preparsing the token stream that results from a lexical analysis of the input text according to the lexical rules in §15.1.3. Pre-parsing for lightweight syntax uses a stack of *contexts*.

- When a column position becomes an offside line, a context is pushed.
- The closing bracketing tokens `)`, `}`, and `end` terminate offside contexts up to and including the context that the corresponding opening token introduced.

15.1.6 Full List of Offside Contexts

This section describes the full list of offside contexts that is kept on the pre-parse stack.

The *SeqBlock* context is the primary context of the analysis.It indicates a sequence of items that must be column-aligned. Where necessary for parsing, the compiler automatically inserts a delimiter that replaces the regular `in` and `;` tokens between the syntax elements. The *SeqBlock* context is pushed at the following times:

- Immediately after the start of a file, excluding lexical directives such as `#if`.
- Immediately after an `=` token is encountered in a *Let* or *Member* context.
- Immediately after a *Paren*, *Then*, *Else*, *WithAugment, Try*, *Finally*, *Do* context is pushed.
- Immediately after an infix token is encountered.
- Immediately after a `->` token is encountered in a *MatchClauses* context.
- Immediately after an `interface`, `class`, or `struct` token is encountered in a type declaration.
- Immediately after an = token is encountered in a record expression when the subsequent token either (a) occurs on the next line or (b) is one of *try*, *match*, *if*, *let*, *for*, *while* or *use*.
- Immediately after a <- token is encoutered when the subsequent token either (a) does not occur on the same line or (b) is one of *try*, *match*, *if*, *let*, *for*, *while* or *use*.

Here “immediately after” refers to the fact that the column position associated with the *SeqBlock* is the first token following the significant token.

In the last two rules, a new line is significant. For example, the following do not start a SeqBlock on the right-hand side of the "<-" token, so it does not parse correctly:

```
let mutable x = 1
// The subsequent token occurs on the same line.
X <- printfn "hello"
     2 + 2
```

To start a SeqBlock on the right, either parentheses or a new line should be used:

```
// The subsequent token does not occur on the same line, so a SeqBlock
is pushed.
X <-
     printfn "hello"
     2 + 2
```

The following contexts are associated with nested constructs that are introduced by the specified keywords:

Context	Pushed when the token stream contains…
Let	The `let` keyword
If	The `if` or `elif` keyword
Try	The `try` keyword
Lazy	The `lazy` keyword
Fun	The `fun` keyword
Function	The `function` keyword
WithLet	The `with` keyword as part of a record expression or an object expression whose members use the syntax`{ new Foo with M() = 1 and  N() = 2 }`
WithAugment	The `with` keyword as part of an extension, interface, or object expression whose members use the syntax `{ new Foo member x.M() = 1 member x. N() = 2 }`
Match	the `match` keyword
For	the `for` keyword
While	The `while` keyword
Then	The `then` keyword
Else	The `else` keyword
Do	The `do` keyword
Type	The `type` keyword
Namespace	The `namespace` keyword
Module	The `module` keyword
Member	▪ The `member`, `abstract`, `default`, or `override` keyword, if the Member context is not already active, because multiple tokens may be present. —or— ▪ `(` is the next token after the `new` keyword. This distinguishes the member declaration `new(x) = ...` from the expression `new x()`

Context	Pushed when the token stream contains...
Paren(token)	`(`, `begin`, `struct`, `sig`, `{`, `[`, `[|`, or *`quote-op-left`*
MatchClauses	The `with` keyword in a *Try* or *Match* context immediately after a `function` keyword.
Vanilla	An otherwise unprocessed keyword in a *SeqBlock* context.

15.1.7 Balancing Rules

When the compiler processes certain tokens, it pops contexts off the offside stack until the stack reaches a particular condition. When it pops a context, the compiler may insert extra tokens to indicate the end of the construct. This procedure is called balancing the stack.

The following table lists the contexts that the compiler pops and the balancing condition for each:

Token	Contexts Popped and Balancing Conditions:
`End`	Enclosing context is one of the following: ▪ `WithAugment` ▪ `Paren(interface)` ▪ `Paren(class)` ▪ `Paren(sig)` ▪ `Paren(struct)` ▪ `Paren(begin)`
`;;`	Pop all contexts from stack
`else`	`If`
`elif`	`If`
`done`	`Do`
`in`	`For or Let`
`eith`	`Match, Member, Interface, Try, Type`
`finally`	`Try`
`)`	`Paren(()`
`}`	`Paren({)`
`]`	`Paren([)`
`|]`	`Paren([|)`
`quote-op-right`	`Paren(quote-op-left)`

15.1.8 Offside Tokens, Token Insertions, and Closing Contexts

The *offside limit* for the current offside stack is the rightmost offside line for the offside contexts on the context stack. The following figure shows the offside limits:

```
let FunctionSample() =
    let tick x = printfn "tick %d" x
    let tock x = printfn "tock %d" x
    let choose f g h x =
        if f x then g x else h x
    for i = 0 to 10 do
        choose (fun n -> n%2 = 0) tick tock i
    printfn "done!"
```

Offside limit for inner `let` and `for` contexts

Offside limit for outer `let` context

When a token occurs on or before the *offside limit* for the current offside stack, and a *permitted undentation* does not apply, enclosing contexts are *closed* until the token is no longer offside. This may result in the insertion of extra delimiting tokens.

Contexts are closed as follows:

- When a *Fun* context is closed, the `$end` token is inserted.
- When a *SeqBlock*, *MatchClauses*, *Let*, or *Do* context is closed, the `$end` token is inserted, with the exception of the first *SeqBlock* pushed for the start of the file.
- When a *While* or *For* context is closed, and the offside token that forced the close is not `done`, the `$done` token is inserted.
- When a *Member* context is closed, the `$end` token is inserted.
- When a *WithAugment* context is closed, the `$end` token is inserted.

If a token is offside and a context cannot be closed, then an "undentation" warning or error is issued to indicate that the construct is badly formatted.

Tokens are also inserted in the following situations:

- When a *SeqBlock* context is pushed, the `$begin` token is inserted, with the exception of the first *SeqBlock* pushed for the start of the file.
- When a token other than `and` appears directly on the offside line of *Let* context, and the next surrounding context is a *SeqBlock*, the `$in` token is inserted.
- When a token occurs directly on the offside line of a *SeqBlock* on the second or subsequent lines of the block, the `$sep` token is inserted. This token plays the same role as `;` in the grammar rules.

  For example, consider this source text:

  ```
  let x = 1
  x
  ```

  The raw token stream contains `let`, `x`, `=`, `1`, `x` and the end-of-file marker `eof`. An initial *SeqBlock* is pushed immediately after the start of the file, at the first token in the file, with an offside line on column 0. The let token pushes a Let context. The `=` token in a Let context pushes a SeqBlock context and inserts a `$begin` token. The `1` pushes a *Vanilla* context. The final token, `x`, is offside from the *Vanilla* context, which pops that context. It is also offside from the *SeqBlock* context, which pops the context and inserts `$end`. It is also offside from the *Let* context, which inserts another `$end` token. It is directly aligned with the *SeqBlock* context, so a `$seq` token is inserted.

15.1.9 Exceptions to the Offside Rules

The compiler makes some exceptions to the offside rules when it analyzes a token to determine whether it is offside from the current context. The following table summarizes the exceptions and shows examples of each.

Context	Exception	Example

<table>
<tr><th>Context</th><th>Exception</th><th>Example</th></tr>
<tr><td>SeqBlock</td><td>An infix token may be offside by the size of the token plus one.</td><td>let x =
        expr + expr
      + expr + expr
  let x =
        expr
     |> f expr
     |> f expr</td></tr>
<tr><td>SeqBlock</td><td>An infix token may align precisely with the offside line of the SeqBlock.</td><td>let someFunction(someCollection) =
    someCollection
    |> List.map (fun x -> x + 1)</td></tr>
<tr><td>SeqBlock</td><td>The infix |> token that begins the last line is not considered as a new element in the sequence block on the right-hand side of the definition. The same also applies to end, and, with, then, and right-parenthesis operators.
In the example, the first) token does not indicate a new element in a sequence of items, even though it aligns precisely with the sequence block that starts at the beginning of the argument list.</td><td>new MenuItem("&Open...",
  new EventHandler(fun _ _ ->
                             ...
             ))</td></tr>
<tr><td>Let</td><td>The and token may align precisely with the let keyword.</td><td>let rec x = 1
and y = 2
x + y</td></tr>
<tr><td>Type</td><td>The }, end, and, and | tokens may align precisely with the type keyword.</td><td>type X =
| A
| B
with
    member x.Seven = 21 / 3
end
and Y = {
    x : int
}
and Z() = class
    member x.Eight = 4 + 4
end</td></tr>
<tr><td>For</td><td>The done token may align precisely with the for keyword.</td><td>for i = 1 to 3 do
  expr
done</td></tr>
<tr><td>SeqBlock;
Match</td><td>On the right-hand side of an arrow for a match expression, a token may align precisely with the match keyword. This exception allows the last expression to align with the match, so that a long series of matches does not increase indentation.</td><td>match x with
| Some(_) -> 1
| None ->
match y with
| Some(_) -> 2
| None ->
3</td></tr>
<tr><td>Interface</td><td>The end token may align precisely with the interface keyword.</td><td>interface IDisposable with
   member x.Dispose() = printfn
disposing!"
end</td></tr>
<tr><td>If</td><td>The then, elif, and else tokens may align precisely with the if keyword.</td><td>if big
then callSomeFunction()
elif small
then callSomeOtherFunction()
else doSomeCleanup()</td></tr>
</table>

Context	Exception	Example

Try	The finally and with tokens may align precisely with the try keyword.	Example 1:
		`try`
		`    callSomeFunction()`
		`finally`
		`    doSomeCleanup()`
		Example 2:
		`try`
		`    callSomeFunction()`
		`with Failure(s) ->`
		`    doSomeCleanup()`

Do	The done token may align precisely with the do keyword.	`for i = 1 to 3`
		`   do`
		`     expr`
		`   done`

15.1.10 Permitted Undentations

As a general rule, incremental indentation requires that nested expressions occur at increasing column positions in indentation-aware code. Warnings or syntax errors are issued when this is not the case. However, undentation is permitted for the following constructs:

- Bodies of function expressions
- Branches of if/then/else expressions
- Bodies of modules and module types

15.1.10.1 Undentation of Bodies of Function Expressions

The bodies of functions may be undented from the fun or function symbol. As a result, the compiler ignores the symbol when determining whether the body of the function satisfies the incremental indentation rule. For example, the printf expression in the following example is undented from the fun symbol that delimits the function definition:

```
let HashSample(tab: Collections.HashTable<_,_>) =
    tab.Iterate (fun c v ->
        printfn "Entry (%O,%O)" c v)
```

However, the block must not undent past other offside lines. Thefollowing is not permitted because the second line breaks the offside line established by the = in the first line:

```
let x = (function (s, n) ->
    (fun z ->
        s+n+z))
```

Constructs enclosed in brackets may be undented.

15.1.10.2 Undentation of Branches of If/Then/Else Expressions

The body of a (...) or begin ... end block in an if/then/else expression may be undented when the body of the block follows the then or else keyword but may not undent further than the if keyword. In this example, the parenthesized block follows then, so the body can be undented to the offside line established by if:

```
let IfSample(day: System.DayOfWeek) =
```

```
if day = System.DayOfWeek.Monday then (
    printf "I don't like Mondays"
)
```

15.1.10.3 Undentation of Bodies of Modules and Module Types

The bodies of modules and module types that are delimited by `begin` and `end` may be undented. For example, in the following example the two let statements that comprise the module body are undented from the `=`.

```
module MyNestedModule = begin
   let one = 1
   let two = 2
end
```

Similarly, the bodies of classes, interfaces, and structs delimited by `{` ... `}`, `class` ... `end`, `struct` ... `end`, or `interface` ... `end` may be undented to the offside line established by the `type` keyword. For example:

```
type MyNestedModule = interface
   abstract P : int
end
```

15.2 High Precedence Application

The entry `f x` in the precedence table in §4.4.2 refers to a function application in which the function and argument are separated by spaces. The entry `"f(x)"` indicates that in expressions and patterns, identifiers that are followed immediately by a left parenthesis without intervening whitespace form a "high precedence" application. Such expressions are parsed with higher precedence than prefix and dot-notation operators. Conceptually this means that

```
Example 1:  B(e)
```

is analyzed lexically as

```
Example 1: B $app (e)
```

where `$app` is an internal symbol inserted by lexical analysis. We do not show this symbol in the remainder of this specification and simply show the original source text.

This means that the following two statements

```
Example 1:  B(e).C
Example 2:  B (e).C
```

are parsed as

```
Example 1: (B(e)).C
Example 2: B ((e).C)
```

respectively.

Furthermore, arbitrary chains of method applications, property lookups, indexer lookups (`.[]`), field lookups, and function applications can be used in sequence if the arguments of method applications are parenthesized and immediately follow the method name, with no intervening spaces. For example:

```
e.Meth1(arg1,arg2).Prop1.[3].Prop2.Meth2()
```

Although the grammar and these precedence rules technically allow the use of high-precedence application expressions as direct arguments, an additional check prevents such use. Instead, such expressions must be surrounded by parentheses. For example,

```
f  e.Meth1(arg1,arg2) e.Meth2(arg1,arg2)
```

must be written

```
f  (e.Meth1(arg1,arg2)) (e.Meth2(arg1,arg2))
```

However, indexer, field, and property dot-notation lookups may be used as arguments without adding parentheses. For example:

```
f  e.Prop1 e.Prop2.[3]
```

15.3 Lexical Analysis of Type Applications

The entry `f<`*`types`*`> x` in the precedence table (§4.4.2) refers to any identifier that is followed immediately by a `<` symbol and a sequence of all of the following:

- `_`, `,`, `*`, `'`, `[`, `]`, whitespace, or identifier tokens.
- A parentheses `(` or `<` token followed by any tokens until a matching parentheses `)` or `>` is encountered.
- A final `>` token.

During this analysis, any token that is composed only of the `>` character (such as `>`, `>>`, or `>>>`) is treated as a series of individual `>` tokens. Likewise, any token composed only of `>` characters followed by a period (such as `>.`, `>>.`, or `>>>.`) is treated as a series of individual `>` tokens followed by a period.

If such a sequence of tokens follows an identifier, lexical analysis marks the construct as a *high precedence type application* and subsequent grammar rules ensure that the enclosed text is parsed as a type. Conceptually this means that

```
Example 1:  B<int>.C<int>(e).C
```

is returned as the following stream of tokens:

```
Example 1: B $app <int> .C $app <int>(e).C
```

where `$app` is an internal symbol inserted by lexical analysis. We do not show this symbol elsewhere in this specification and simply show the original source text.

The lexical analysis of type applications does not apply to the character sequence “`<>`”. A character sequence such as “`< >`” with intervening whitespace should be used to indicate an empty list of generic arguments.

```
type Foo() =
    member this.Value = 1
let b = new Foo< >()  // valid
let c = new Foo<>()  // invalid
```

16. Provided Types

Type providers are extensions provided to an F# compiler or interpreter which provide information about types available in the environment for the F# code being analysed.

The compilation context is augmented with a set of *type provider instances*. A type provider insance is interrogated for information through *type provider invocations* (TPI). Type provider invocations are all executed at compile-time. The type provider instance is not required at runtime.

Wherever an operation on a provided namespace, provided type definition or provided member is mentioned in this section, it is assumed to be a compile-time type provider invocation.

The exact protocol used to implement type provider invocations and communicate between an F# compiler/interpreter and type provider instances is implementation dependent.

As of this release of F#,

- a type provider is a .NET 4.x binary component referenced as an imported asssembly reference. The assembly should have a `TypeProviderAssemblyAttribute`, with at least one component marked with `TypeProviderAttribute`.
- a type provider instance is an object created for a component marked with `TypeProviderAttribute`.
- provided type definitions are `System.Type` objects returned by a type provider instance.
- provided methods are `System.Reflection.MethodInfo` objects returned by a type provider instance.
- provided constructors are `System.Reflection.ConstructorInfo` objects returned by a type provider instance.
- provided properties are `System.Reflection.PropertyInfo` objects returned by a type provider instance.
- provided events are `System.Reflection.EventInfo` objects returned by a type provider instance.
- provided literal fields are `System.Reflection.FieldInfo` objects returned by a type provider instance.
- provided parameters are `System.Reflection.ParameterInfo` objects returned by a type provider instance.
- provided static parameters are `System.Reflection.ParameterInfo` objects returned by a type provider instance.
- provided attributes are attribute value objects returned by a type provider instance.

16.1 Static Parameters

The syntax of types in F# is expanded to include static parameters, including named static parameters:

```
type-arg =
    …
    static-parameter

static-parameter =
    static-parameter-value
    id = static-parameter-value

static-parameter-value =
    const expr
    simple-constant-expression
```

References to provided types may include static parameters, e.g.

```
type SomeService = ODataService<"http://some.url.org/service">
```

Static parameters which are constant expressions, but not simple literal constants, may be specified using the const keyword, e.g.

```
type SomeService = CsvFile<const (__SOURCE_DIRECTORY__ + "/a.csv")>
```

Parentheses are needed around any simple contanst expressions after "const" that are not simple literal constants, e.g.

```
type K = N.T< const (+1) >
```

During type checking of a type `A<type-args>`, where A is a provided type, the TPM `GetStaticParameters` is invoked to determine the static parameters for the type A. If the static parameters match, the TPM `ApplyStaticArguments` is invoked to apply the static arguments to the provided type Static parameters must be given for each non-optional static parameter.

16.1.1 Mangling of Static Parameter Values

Static parameter values are encoded into the names used for types within F# metadata. The encoding scheme used is

encoding(A<arg1,…,argN>) =

typeName,ParamName1=**encoding**(arg1),…, ParamNameN=**encoding**(argN)

encoding(*v*) = "*s*"

where *s* is the result applying the F# 'string' operator to v (using invariant numeric formatting), and in the result " is replaced by \" and \ by \\

16.2 Provided Namespace

Each type provider instance in the assembly context reports a collection of provided namespaces though the `GetNamespaces` type provider method. Each provided namespace can in turn report further namespaces through the `GetNestedNamespaces` type provider method.

16.3 Provided Type Definitions

Each provided namespace reports provided type definitions though the `GetTypes` and `ResolveTypeName` type provider methods. The type provider is obliged to ensure that these two methods return consistent results.

Name resolution for unqualified identifiers may return provided type definitions if no other resolution is available.

16.3.1 Generated v. Erased Types

Each provided type definition may be *generated* or *erased*. In this case, the types and method calls are removed entirely during compilation and replaced with other representations. When an erased type is used, the compiler will replace it with the first concrete type in its inheritance chain as returned by the TPM `type.BaseType`. The erasure of an erased interface type is "object".

- If it has a type definition under a path **D.E.F**, and the .Assembly of that type is in a different assembly **A** to the provider's assembly, then that type definition is a "generated" type definition. Otherwise it is an erased type definition.
- Erased type definitions must return `TypeAttributes` with the `IsErased` flag set, value `0x40000000` and given by the F# literal `TypeProviderTypeAttributes.IsErased`.
- When a provided type definition is generated, its reported assembly **A** is treated as an injected assembly which is statically linked into the resulting assembly.
- Concrete type definitions (both provided and F#-authored) and object expressions may not inherit from erased types
- Concrete type definitions (both provided and F#-authored) and object expressions may not implement erased interfaces
- If an erased type definition reports an interface, its erasure must implement the erasure of that interface. The interfaces reported by an erased type definition must be unique up to erasure.
- Erased types may not be used as the target type of a runtime type test of runtime coercion.
- When determining uniqueness for F#-declared methods, uniqueness is determined after erasure of both provided types and units of measure.
- The elaborated form of F# expressions is after erasure of provided types.

- Two generated type definitions are equivalent if and only if they have the same F# path and name in the same assembly, once they are rooted according to their corresponding generative type definition.

- Two erased type definitions are only equivalent if they are provided by the same provider, using the same type name, with the same static arguments.

16.3.2 Type References

The elements of provided type definitions may reference other provided type definitions, and types from imported assemblies referenced in the compilation context. They may not reference type defined in the F# code currently being compiled.

16.3.3 Static Parameters

A provided type definition may report a set of static parameters. For such a type definition, all other provided contents are ignored.

Static parameters may be optional and/or named, indicated by the `Attributes` property of the static parameter. For a given provided type definition, multiple static parameters may not have the same name and named static arguments must come after all other arguments.

16.3.4 Kind

- Provided type definitions may be classes.

  This includes both erased and concrete types. This corresponds to the **type.IsClass** property returning true for the provided type definition.

- Provided type definitions may be interfaces.

  This includes both erased and concrete types. This corresponds to the `type.IsInterface` property returning true. Only one of `IsInterface`, `IsClass`, `IsStruct`, `IsEnum`, `IsDelegate`, `IsArray` may return true.

- Provided type definitions may be static classes.

  This includes both erased and concrete types.

- Provided type definitions may be sealed.

- Provided type definitions may not be arrays. This means the **type.IsArray** property must always return false. Provided types used in return types and argument positions may be array “symbol” types, see below.

16.3.5 Inheritance

- Provided type definitions may report base types.

- Provided type definition may report interfaces.

16.3.6 Members

- Provided type definitions may report methods.

  This corresponds to non-null results from the **type.GetMethod** and **type.GetMethods** of the provided type definition. The results returned by these methods must be consistent.

  - Provided methods may be static, instance and abstract

  - Provided methods may not be class constructors (.cctor). By .NET rules these would have to be private anyway.

  - Provided methods may be operators such as op_Addition.

- Provided type definitions may report properties.

  This corresponds to non-null results from the **type.GetProperty** and **type.GetProperties** of the provided type definition. The results returned by these methods must be consistent.

  - Provided properties may be static or instance

  - Provided properties may be indexers. This corresponds to reporting methods with name **Item**, or as identified by **DefaultMemberAttribute** non-null results from the **type.GetEvent** and **type.GetEvents** of the provided type definition. The results returned by these methods must be consistent. This include 1D, 2D, 3D and 4D indexer access notation in F# (corresponding to different numbers of parameters to the indexer property).

- Provided type definitions may report constructors.

  This corresponds to non-null results from the **type.GetConstructor** and **type.GetConstructors** of the provided type definition. The results returned by these methods must be consistent.

- Provided type definitions may report events.

  This corresponds to non-null results from the **type.GetEvent** and **type.GetEvents** of the provided type definition. The results returned by these methods must be consistent.

- Provided type definitions may report nested types.

  This corresponds to non-null results from the **type.GetNestedType** and **type.GetNestedTypes** of the provided type definition. The results returned by these methods must be consistent.

  - The nested types of an erased type may be generated types in a generated assembly. The **type.DeclaringType** property of the nested type need not report the erased type.

- Provided type definitions may report literal (constant) fields.

This corresponds to non-null results from the `type.GetField` and `type.GetFields` of the provided type definition, and is related to the fact that provided types may be enumerations. The results returned by these methods must be consistent.

- Provided type definitions may not report non-literal (i.e. non-const) fields

  This is a deliberate feature limitation, because in .NET, non-literal fields should not appear in public API surface area.

16.3.7 Attributes

- Provided type definitions, properties, constructors, events and methods may report attributes.

  This includes `ObsoleteAttribute` and `ParamArrayAttribute` attributes

16.3.8 Accessibility

- All erased provided type definitions must be public

  However, concrete provided types are each in an assembly A that gets statically linked into the resulting F# component. These assemblies may contain private types and methods. These types are not directly "provided" types, since they are not returned to the compiler by the API, but they are part of the closure of the types that are being embedded.

16.3.9 Elaborated Code

Elaborated uses of provided methods are erased to elaborated expressions returned by the TPM `GetInvokerExpression`. In the current release of F#, replacement elaborated expressions are specified via F# quotation values composed of quotations constructed with respect to the referenced assemblies in the compilation context according to the following quotation library calls:

- **Expr.NewArray**
- **Expr.NewObject**
- **Expr.WhileLoop**
- **Expr.NewDelegate**
- **Expr.ForIntegerRangeLoop**
- **Expr.Sequential**
- **Expr.TryWith**
- **Expr.TryFinally**
- **Expr.Lambda**
- **Expr.Call**
- **Expr.Constant**
- **Expr.DefaultValue**
- **Expr.NewTuple**
- **Expr.TupleGet**
- **Expr.TypeAs**

- **Expr.TypeTest**
- **Expr.Let**
- **Expr.VarSet**
- **Expr.IfThenElse**
- **Expr.Var**

The type of the quotation expression returned by `GetInvokerExpression` must be an erased type. The type provider is obliged to ensure that this type is equivalent to the erased type of the expression it is replacing.

16.3.10 Further Restrictions

- If a provided type definition reports a member with `ExtensionAttribute`, it is not treated as an extension member

- Provided type and method definitions may not be generic

  This corresponds to

  - `GetGenericArguments` returning length 0

  - For type definitions, `IsGenericType` and `IsGenericTypeDefinition` returning false

  - For method definitions, `IsGenericMethod` and `IsGenericMethodDefinition` returning false

17. Special Attributes and Types

This chapter describes attributes and types that have special significance to the F# compiler.

17.1 Custom Attributes Recognized by F#

The following custom attributes have special meanings recognized by the F# compiler. Except where indicated, the attributes may be used in F# code, in referenced assemblies authored in F#, or in assemblies that are authored in other CLI languages.

Attribute	Description
`System.ObsoleteAttribute` `[<Obsolete(...)>]`	Indicates that the construct is obsolete and gives a warning or error depending on the settings in the attribute. This attribute may be used in both F# and imported assemblies.
`System.ParamArrayAttribute` `[<ParamArray(...)>]`	When applied to an argument of a method, indicates that the method can accept a variable number of arguments. This attribute may be used in both F# and imported assemblies.
`System.ThreadStaticAttribute` `[<ThreadStatic(...)>]`	Marks a mutable static value in a class as thread static. This attribute may be used in both F# and imported assemblies.
`System.ContextStaticAttribute` `[<ContextStatic(...)>]`	Marks a mutable static value in a class as context static. This attribute may be used in both F# and imported assemblies.
`System.AttributeUsageAttribute` `[<AttributeUsage(...)>]`	Specifies the attribute usage targets for an attribute. This attribute may be used in both F# and imported assemblies.
`System.Diagnostics.ConditionalAttribute` `[<Conditional(...)>]`	Emits code to call the method only if the corresponding conditional compilation symbol is defined. This attribute may be used in both F# and imported assemblies.
`System.Reflection.AssemblyInformationalVersionAttribute` `[<AssemblyInformationalVersion(...)>]`	Attaches additional version metadata to the compiled form of the assembly. This attribute may be used in both F# and imported assemblies.
`System.Reflection.AssemblyFileVersionAttribute` `[<AssemblyFileVersion(...)>]`	Attaches file version metadata to the compiled form of the assembly. This attribute may be used in both F# and imported assemblies.

Attribute	Description
`System.Reflection.AssemblyDescriptionAttribute` `[<AssemblyDescription(...)>]`	Attaches descriptive metadata to the compiled form of the assembly, such as the “Comments” attribute in the Win32 version resource for the assembly. This attribute may be used in both F# and imported assemblies.
`System.Reflection.AssemblyTitleAttribute` `[<AssemblyTitle(...)>]`	Attaches title metadata to the compiled form of the assembly, such as the “ProductName” attribute in the Win32 version resource for the assembly. This attribute may be used in both F# and imported assemblies.
`System.Reflection.AssemblyCopyrightAttribute` `[<AssemblyCopyright(...)>]`	Attaches copyright metadata to the compiled form of the assembly, such as the “LegalCopyright” attribute in the Win32 version resource for the assembly. This attribute may be used in both F# and imported assemblies.
`System.Reflection.AssemblyTrademarkAttribute` `[<AssemblyTrademark(...)>]`	Attaches trademark metadata to the compiled form of the assembly, such as the “LegalTrademarks” attribute in the Win32 version resource for the assembly. This attribute may be used in both F# and imported assemblies.
`System.Reflection.AssemblyCompanyAttribute` `[<AssemblyCompany(...)>]`	Attaches company name metadata to the compiled form of the assembly, such as the “CompanyName” attribute in the Win32 version resource for the assembly. This attribute may be used in both F# and imported assemblies.
`System.Reflection.AssemblyProductAttribute` `[<AssemblyProduct(...)>]`	Attaches product name metadata to the compiled form of the assembly, such as the “ProductName” attribute in the Win32 version resource for the assembly. This attribute may be used in both F# and imported assemblies.
`System.Reflection.AssemblyKeyFileAttribute` `[<AssemblyKeyFile(...)>]`	Indicates to the F# compiler how to sign an assembly. This attribute may be used in both F# and imported assemblies.
`System.Reflection.DefaultMemberAttribute` `[<DefaultMember(...)>]`	When applied to a type, specifies the name of the indexer property for that type. This attribute may be used in both F# and imported assemblies.
`System.Runtime.CompilerServices.InternalsVisibleToAttribute` `[<InternalsVisibleTo(...)>]`	Directs the F# compiler to permit access to the internals of the assembly. This attribute may be used in both F# and imported assemblies.
`System.Runtime.CompilerServices.TypeForwardedToAttribute` `[<TypeForwardedTo(...)>]`	Indicates a type redirection. This attribute may be used only in imported non-F# assemblies. It is not permitted in F# code.

Attribute	Description
System.Runtime.CompilerServices.ExtensionAttribute [<Extension(...)>]	Indicates the compiled form of a C# extension member. This attribute may be used only in imported non-F# assemblies. It is not permitted in F# code.
System.Runtime.InteropServices.DllImportAttribute [<DllImport(...)>]	When applied to a function definition in a module, causes the F# compiler to ignore the implementation of the definition, and instead compile it as a CLI P/Invoke stub declaration. This attribute may be used in both F# and imported assemblies.
System.Runtime.InteropServices.MarshalAsAttribute [<MarshalAs(...)>]	When applied to a parameter or return type, specifies the marshalling attribute for a CLI P/Invoke stub declaration. This attribute may be used in both F# and imported assemblies. However, F# does not support the specification of "custom" marshallers.
System.Runtime.InteropServices.InAttribute [<In>]	When applied to a parameter, specifies the CLI In attribute. This attribute may be used in both F# and imported assemblies. However, in F# its only effect is to change the corresponding attribute in the CLI compiled form.
System.Runtime.InteropServices.OutAttribute [<Out>]	When applied to a parameter, specifies the CLI Out attribute. This attribute may be used in both F# and imported assemblies. However, in F# its only effect is to change the corresponding attribute in the CLI compiled form.
System.Runtime.InteropServices.OptionalAttribute [<Optional(...)>]	When applied to a parameter, specifies the CLI Optional attribute. This attribute may be used in both F# and imported assemblies. However, in F# its only effect is to change the corresponding attribute in the CLI compiled form.
System.Runtime.InteropServices.FieldOffsetAttribute [<FieldOffset(...)>]	When applied to a field, specifies the field offset of the underlying CLI field. This attribute may be used in both F# and imported assemblies.
System.NonSerializedAttribute [<NonSerialized>]	When applied to a field, sets the "not serialized" bit for the underlying CLI field. This attribute may be used in both F# and imported assemblies.
System.Runtime.InteropServices.StructLayoutAttribute [<StructLayout(...)>]	Specifies the layout of a CLI type. This attribute may be used in both F# and imported assemblies.
FSharp.Core.AutoSerializableAttribute [<AutoSerializable(false)>]	When added to a type with value **false**, disables default serialization, so that F# does not make the type serializable. This attribute should be used only in F# assemblies.

Attribute	Description
`FSharp.Core.CLIMutableAttribute` `[<CLIMutable>]`	When specified, a record type is compiled to a CLI representation with a default constructor with property getters and setters. This attribute should be used only in F# assemblies.
`FSharp.Core.AutoOpenAttribute` `[<AutoOpen>]`	When applied to an assembly and given a string argument, causes the namespace or module to be opened automatically when the assembly is referenced. When applied to a module without a string argument, causes the module to be opened automatically when the enclosing namespace or module is opened. This attribute should be used only in F# assemblies.

`FSharp.Core.`
`CompilationRepresentationAttribute`
`[<CompilationRepresentation(...)>]`
	Adjusts the runtime representation of a type .
	This attribute should be used only in F# assemblies.

`FSharp.Core.CompiledNameAttribute` `[<CompiledName(...)>]`	Changes the compiled name of an F# language construct. This attribute should be used only in F# assemblies.
`FSharp.Core.CustomComparisonAttribute` `[<CustomComparison>]`	When applied to an F# structural type, indicates that the type has a user-specified comparison implementation. This attribute should be used only in F# assemblies.
`FSharp.Core.CustomEqualityAttribute` `[<CustomEquality>]`	When applied to an F# structural type, indicates that the type has a user-defined equality implementation. This attribute should be used only in F# assemblies.
`FSharp.Core.DefaultAugmentationAttribute` `[<DefaultAugmentation(...)>]`	When applied to an F# discriminated union type with value false, turns off the generation of standard helper member tester, constructor and accessor members for the generated CLI class for that type. This attribute should be used only in F# assemblies.
`FSharp.Core.DefaultValueAttribute` `[<DefaultValue(...)>]`	When added to a field declaration, specifies that the field is not initialized. During type checking, a constraint is asserted that the field type supports null. If the argument to the attribute is **false**, the constraint is not asserted. This attribute should be used only in F# assemblies.
`FSharp.Core.GeneralizableValueAttribute` `[<GeneralizableValue>]`	When applied to an F# value, indicates that uses of the attribute can result in generic code through the process of type inference. For example, Set.empty. The value must typically be a type function whose implementation has no observable side effects. This attribute should be used only in F# assemblies.

Attribute	Description
FSharp.Core.LiteralAttribute [<Literal>]	When applied to a value, compiles the value as a CLI literal. This attribute should be used only in F# assemblies.
FSharp.Core.NoDynamicInvocationAttribute [<NoDynamicInvocation>]	When applied to an inline function or member definition, replaces the generated code with a stub that throws an exception at runtime. This attribute is used to replace the default generated implementation of unverifiable inline members with a verifiable stub. This attribute should be used only in F# assemblies.
FSharp.Core.CompilerMessageAttribute [<CompilerMessage(...)>]	When applied to an F# construct, indicates that the F# compiler should report a message when the construct is used. This attribute should be used only in F# assemblies.
FSharp.Core.StructAttribute [<Struct>]	Indicates that a type is a struct type. This attribute should be used only in F# assemblies.
FSharp.Core.ClassAttribute [<Class>]	Indicates that a type is a class type. This attribute should be used only in F# assemblies.
FSharp.Core.InterfaceAttribute [<Interface>]	Indicates that a type is an interface type. This attribute should be used only in F# assemblies.
FSharp.Core.MeasureAttribute [<Measure>]	Indicates that a type or generic parameter is a unit of measure definition or annotation. This attribute should be used only in F# assemblies.
FSharp.Core.ReferenceEqualityAttribute [<ReferenceEquality>]	When applied to an F# record or union type, indicates that the type should use reference equality for its default equality implementation. This attribute should be used only in F# assemblies.
FSharp.Core.ReflectedDefinitionAttribute [<ReflectedDefinition>]	Makes the quotation form of a definition available at runtime through the FSharp.Quotations.Expr.GetReflectedDefinition method. This attribute should be used only in F# assemblies.
FSharp.Core.RequireQualifiedAccessAttribute [<RequireQualifiedAccess>]	When applied to an F# module, warns if an attempt is made to open the module name. When applied to an F# union or record type, indicates that the field labels or union cases must be referenced by using a qualified path that includes the type name. This attribute should be used only in F# assemblies.

Attribute	Description
FSharp.Core. RequiresExplicitTypeArgumentsAttribute [<RequiresExplicitTypeArguments>]	When applied to an F# function or method, indicates that the function or method must be invoked with explicit type arguments, such as `typeof<int>`. This attribute should be used only in F# assemblies.
FSharp.Core.StructuralComparisonAttribute [<StructuralComparison>]	When added to a record, union, exception, or structure type, confirms the automatic generation of implementations for **IComparable** for the type. This attribute should only be used in F# assemblies.
FSharp.Core.StructuralEqualityAttribute [<StructuralEquality>]	When added to a record, union, or struct type, confirms the automatic generation of overrides for Equals and GetHashCode for the type. This attribute should be used only in F# assemblies.
FSharp.Core.VolatileFieldAttribute [<VolatileField>]	When applied to an F# field or mutable value definition, controls whether the CLI `volatile` prefix is emitted before accesses to the field. This attribute should be used only in F# assemblies.
FSharp.Core.TypeProviderXmlDocAttribute	Specifies documentation for provided type definitions and provided members
FSharp.Core.TypeProviderDefinitionLocationAttribute	Specifies location information for provided type definitions and provided members

17.2 Custom Attributes Emitted by F#

The F# compiler can emit the following custom attributes:

Attribute	**Description**
System.Diagnostics.DebuggableAttribute	Improves debuggability of F# code.
System.Diagnostics.DebuggerHiddenAttribute	Improves debuggability of F# code.
System.Diagnostics.DebuggerDisplayAttribute	Improves debuggability of F# code.
System.Diagnostics.DebuggerBrowsableAttribute	Improves debuggability of F# code.
System.Runtime.CompilerServices. CompilationRelaxationsAttribute	Enables extra JIT optimizations.
System.Runtime.CompilerServices. CompilerGeneratedAttribute	Indicates that a method, type, or property is generated by the F# compiler, and does not correspond directly to user source code.
System.Reflection.DefaultMemberAttribute	Specifies the name of the indexer property for a class.
FSharp.Core.CompilationMappingAttribute	Indicates how a CLI construct corresponds to an F# source language construct.
FSharp.Core.FSharpInterfaceDataVersionAttribute	Defines the schema number for the embedded binary resource for F#-specific interface and optimization data.

Attribute	Description
FSharp.Core.OptionalArgumentAttribute	Indicates optional arguments to F# members.

17.3 Custom Attributes Not Recognized by F#

The following custom attributes are defined in some CLI implementations and may appear to be relevant to F#. However, they either do not affect the behavior of the F# compiler, or result in an error when used in in F# code.

Attribute	Description
System.Runtime.CompilerServices.DecimalConstantAttribute	The F# compiler ignores this attribute. However, if used in F# code, it can cause some other CLI languages to interpret a decimal constant as a compile-time literal.
System.Runtime.CompilerServices.RequiredAttributeAttribute	Do not use this attribute in F# code. The F# compiler ignores it or returns an error.
System.Runtime.InteropServices.DefaultParameterValueAttribute	Do not use this attribute in F# code. The F# compiler ignores it or returns an error.
System.Runtime.InteropServices.UnmanagedFunctionPointerAttribute	Do not use this attribute in F# code. The F# compiler ignores it or returns an error.
System.Runtime.CompilerServices.FixedBufferAttribute	Do not use this attribute in F# code. The F# compiler ignores it or returns an error.
System.Runtime.CompilerServices.UnsafeValueTypeAttribute	Do not use this attribute in F# code. The F# compiler ignores it or returns an error.
System.Runtime.CompilerServices.SpecialNameAttribute	Do not use this attribute in F# code. The F# compiler ignores it or returns an error.

17.4 Exceptions Thrown by F# Language Primitives

Certain F# language and primitive library operations throw the following exceptions.

Attribute	Description
System.ArithmeticException	An arithmetic operation failed. This is the base class for exceptions such as System.DivideByZeroException and System.OverflowException.
System.ArrayTypeMismatchException	An attempt to store an element in an array failed because the runtime type of the stored element is incompatible with the runtime type of the array.
System.DivideByZeroException	An attempt to divide an integral value by zero occurred.
System.IndexOutOfRangeException	An attempt to index an array failed because the index is less than zero or outside the bounds of the array.
System.InvalidCastException	An explicit conversion from a base type or interface to a derived type failed at run time.

Attribute	Description
`System.NullReferenceException`	A null reference was used in a way that caused the referenced object to be required.
`System.OutOfMemoryException`	An attempt to use `new` to allocate memory failed.
`System.OverflowException`	An arithmetic operation in a checked context overflowed.
`System.StackOverflowException`	The execution stack was exhausted because of too many pending method calls, which typically indicates deep or unbounded recursion.
`System.TypeInitializationException`	F# initialization code for a type threw an exception that was not caught.

18. The F# Library FSharp.Core.dll

All compilations reference the following two base libraries:

- The CLI base library `mscorlib.dll`.
- The F# base library `FSharp.Core.dll`

The following namespaces are automatically opened for all F# code:

```
open FSharp
open FSharp.Core
open FSharp.Core.LanguagePrimitives
open FSharp.Core.Operators
open FSharp.Text
open FSharp.Collections
open FSharp.Core.ExtraTopLevelOperators
```

A compilation may open additional namespaces may be opened if the referenced F# DLLs contain `AutoOpenAttribute` declarations.

See also the online documentation at http://msdn.com/library/ee353567.aspx.

18.1 Basic Types (FSharp.Core)

This section provides details about the basic types that are defined in `FSharp.Core`.

18.1.1 Basic Type Abbreviations

Type Name	Description
`obj`	`System.Object`
`exn`	`System.Exception`
`nativeint`	`System.IntPtr`
`unativeint`	`System.UIntPtr`
`string`	`System.String`
`float32, single`	`System.Single`
`float, double`	`System.Double`
`sbyte, int8`	`System.SByte`
`byte, uint8`	`System.Byte`
`int16`	`System.Int16`
`uint16`	`System.UInt16`
`int32, int`	`System.Int32`
`uint32`	`System.UInt32`
`int64`	`System.Int64`
`uint64`	`System.UInt64`
`char`	`System.Char`
`bool`	`System.Boolean`
`decimal`	`System.Decimal`

18.1.2 Basic Types that Accept Unit of Measure Annotations

Type Name	Description
sbyte<_>	Underlying representation System.SByte, but accepts a unit of measure.
int16<_>	Underlying representation System.Int16, but accepts a unit of measure.
int32<_>	Underlying representation System.Int32, but accepts a unit of measure.
int64<_>	Underlying representation System.Int64, but accepts a unit of measure.
float32<_>	Underlying representation System.Single, but accepts a unit of measure.
float<_>	Underlying representation System.Double, but accepts a unit of measure.
decimal<_>	Underlying representation System.Decimal, but accepts a unit of measure.

18.1.3 The nativeptr<_> Type

When the nativeptr<*type*> is used in method argument or return position, it is represented in compiled CIL code as either:

- A CLI pointer type *type**, if *type* does not contain any generic type parameters.
- T CLI type System.IntPtr otherwise.

> **Note**: CLI pointer types are rarely used. In CLI metadata, pointer types sometimes appear in CLI metadata unsafe object constructors for the CLI type System.String.
>
> You can convert between System.UIntPtr and nativeptr<'T> by using the inlined unverifiable functions in FSharp.NativeInterop.NativePtr.
>
> nativeptr<_> compiles in different ways because CLI restricts where pointer types can appear.

18.2 Basic Operators and Functions (FSharp.Core.Operators)

18.2.1 Basic Arithmetic Operators

The following operators are defined in FSharp.Core.Operators:

Operator or Function Name	Expression Form	Description
(+)	x + y	Overloaded addition.
(-)	x - y	Overloaded subtraction.
(*)	x * y	Overloaded multiplication.
(/)	x / y	Overloaded division. For negative numbers, the behavior of this operator follows the definition of the corresponding operator in the C# specification.

Operator or Function Name	Expression Form	Description
`(%)`	`x % y`	Overloaded remainder. For integer types, the result of `x % y` is the value produced by `x – (x / y) * y`. If `y` is zero, `System.DivideByZeroException` is thrown. The remainder operator never causes an overflow. This follows the definition of the remainder operator in the C# specification. For floating-point types, the behavior of this operator also follows the definition of the remainder operator in the C# specification.
`(~-)`	`-x`	Overloaded unary negation.
`not`	`not x`	Boolean negation.

18.2.2 Generic Equality and Comparison Operators

The following operators are defined in `FSharp.Core.Operators`:

Operator or Function Name	Expression Form	Description
`(<)`	`x < y`	Generic less-than
`(<=)`	`x <= y`	Generic less-than-or-equal
`(>)`	`x > y`	Generic greater-than
`(>=)`	`x >= y`	Generic greater-than-or-equal
`(=)`	`x = y`	Generic equality
`(<>)`	`x <> y`	Generic disequality
`max`	`max x y`	Generic maximum
`min`	`min x y`	Generic minimum

18.2.3 Bitwise Operators

The following operators are defined in `FSharp.Core.Operators`:

<table>
<tr><th>Operator or Function Name</th><th>Expression Form</th><th>Description</th></tr>
<tr><td><code>(<<<)</code></td><td><code>x <<< y</code></td><td>Overloaded bitwise left-shift</td></tr>
<tr><td><code>(>>>)</code></td><td><code>x >>> y</code></td><td>Overloaded bitwise arithmetic right-shift</td></tr>
<tr><td><code>(^^^)</code></td><td><code>x ^^^ y</code></td><td>Overloaded bitwise exclusive or (XOR)</td></tr>
<tr><td><code>(&&&)</code></td><td><code>x &&& y</code></td><td>Overloaded bitwise and</td></tr>
<tr><td><code>(|||)</code></td><td><code>x ||| y</code></td><td>Overloaded bitwise or</td></tr>
<tr><td><code>(~~~)</code></td><td><code>~~~x</code></td><td>Overloaded bitwise negation</td></tr>
</table>

18.2.4 Math Operators

The following operators are defined in `FSharp.Core.Operators`:

Operator or Function Name	Expression Form	Description
`abs`	`abs x`	Overloaded absolute value
`acos`	`acos x`	Overloaded inverse cosine
`asin`	`asin x`	Overloaded inverse sine
`atan`	`atan x`	Overloaded inverse tangent
`atan2`	`atan2 x y`	Overloaded inverse tangent of `x/y`

Operator or Function Name	Expression Form	Description
ceil	ceil x	Overloaded floating-point ceiling
cos	cos x	Overloaded cosine
cosh	cosh x	Overloaded hyperbolic cosine
exp	exp x	Overloaded exponent
floor	floor x	Overloaded floating-point floor
log	log x	Overloaded natural logarithm
log10	log10 x	Overloaded base-10 logarithm
(**)	x ** y	Overloaded exponential
pown	pown x y	Overloaded integer exponential
round	round x	Overloaded rounding
sign	sign x	Overloaded sign function
sin	sin x	Overloaded sine function
sinh	sinh x	Overloaded hyperbolic sine function
sqrt	sqrt x	Overloaded square root function
tan	tan x	Overloaded tangent function
tanh	tanh x	Overloaded hyperbolic tangent function

18.2.5 Function Pipelining and Composition Operators

The following operators are defined in FSharp.Core.Operators:

<table>
<tr><th>Operator/Function Name</th><th>Expression Form</th><th>Description</th></tr>
<tr><td>(|>)</td><td>x |> f</td><td>Pipelines the value x to the function f (forward pipelining)</td></tr>
<tr><td>(>>)</td><td>f >> g</td><td>Composes two functions, so that they are applied in order from left to right</td></tr>
<tr><td>(<|)</td><td>f <| x</td><td>Pipelines the value x to the function f (backward pipelining)</td></tr>
<tr><td>(<<)</td><td>g << f</td><td>Composes two functions, so that they are applied in order from right to left (backward function composition)</td></tr>
<tr><td>ignore</td><td>ignore x</td><td>Computes and discards a value</td></tr>
</table>

18.2.6 Object Transformation Operators

The following operators are defined in FSharp.Core.Operators:

Operator/Function Name	Expression Form	Description
box	box x	Converts to object representation.
hash	hash x	Generates a hash value.
sizeof	sizeof<*type*>	Computes the size of a value of the given type.
typeof	typeof<*type*>	Computes the System.Type representation of the given type.
typedefof	typedefof<*type*>	Computes the System.Type representation of *type* and calls GetGenericTypeDefinition if it is a generic type.
unbox	unbox x	Converts from object representation.
ref	ref x	Allocates a mutable reference cell.
(!)	!x	Reads a mutable reference cell.

18.2.7 Pair Operators

The following operators are defined in `FSharp.Core.Operators`:

Operator or Function Name	Expression Form	Description
`fst`	`fst p`	Returns the first element of a pair.
`snd`	`snd p`	Returns the second element of a pair

18.2.8 Exception Operators

The following operators are defined in `FSharp.Core.Operators`:

Operator/Function Name	Expression Form	Description
`failwith`	`failwith x`	Raises a `FailureException` exception.
`invalidArg`	`invalidArg x`	Raises an `ArgumentException` exception.
`raise`	`raise x`	Raises an exception.
`reraise`	`reraise()`	Rethrows the current exception.

18.2.9 Input/Output Handles

The following operators are defined in `FSharp.Core.Operators`:

Operator or Function Name	Expression Form	Description
`stdin`	`Stdin`	Computes `System.Console.In`.
`stdout`	`Stdout`	Computes `System.Console.Out`.
`stderr`	`Stderr`	Computes `System.Console.Error`.

18.2.10 Overloaded Conversion Functions

The following operators are defined in `FSharp.Core.Operators`:

Operator or Function Name	Expression Form	Description
`byte`	`byte x`	Overloaded conversion to a byte
`sbyte`	`sbyte x`	Overloaded conversion to a signed byte
`int16`	`int16 x`	Overloaded conversion to a 16-bit integer
`uint16`	`uint16 x`	Overloaded conversion to an unsigned 16-bit integer
`int32, int`	`int32 x` `int x`	Overloaded conversion to a 32-bit integer
`uint32`	`uint32 x`	Overloaded conversion to an unsigned 32-bit integer
`int64`	`int64 x`	Overloaded conversion to a 64-bit integer
`uint64`	`uint64 x`	Overloaded conversion to an unsigned 64-bit integer
`nativeint`	`nativeint x`	Overloaded conversion to an native integer
`unativeint`	`unativeint x`	Overloaded conversion to an unsigned native integer
`float, double`	`float x` `double x`	Overloaded conversion to a 64-bit IEEE floating-point number
`float32, single`	`float32 x` `single x`	Overloaded conversion to a 32-bit IEEE floating-point number
`decimal`	`decimal x`	Overloaded conversion to a `System.Decimal` number
`char`	`char x`	Overloaded conversion to a `System.Char` value
`enum`	`enum x`	Overloaded conversion to a typed enumeration value

18.3 Checked Arithmetic Operators

The module `FSharp.Core.Operators.Checked` defines runtime-overflow-checked versions of the following operators:

Operator or Function Name	Expression Form	Description
`(+)`	`x + y`	Checked overloaded addition
`(-)`	`x – y`	Checked overloaded subtraction
`(*)`	`x * y`	Checked overloaded multiplication
`(~-)`	`-x`	Checked overloaded unary negation
`byte`	`byte x`	Checked overloaded conversion to a byte
`sbyte`	`sbyte x`	Checked overloaded conversion to a signed byte
`int16`	`int16 x`	Checked overloaded conversion to a 16-bit integer
`uint16`	`uint16 x`	Checked overloaded conversion to an unsigned 16-bit integer
`int32, int`	`int32 x` `int x`	Checked overloaded conversion to a 32-bit integer
`uint32`	`uint32 x`	Checked overloaded conversion to an unsigned 32-bit integer
`int64`	`int64 x`	Checked overloaded conversion to a 64-bit integer
`uint64`	`uint64 x`	Checked overloaded conversion to an unsigned 64-bit integer
`nativeint`	`nativeint x`	Checked overloaded conversion to an native integer
`unativeint`	`unativeint x`	Checked overloaded conversion to an unsigned native integer
`char`	`char x`	Checked overloaded conversion to a `System.Char` value

18.4 List and Option Types

18.4.1 The List Type

The following shows the elements of the F# type `FSharp.Collections.list` referred to in this specification:

```
type 'T list =
    | ([])
    | (::)  of 'T * 'T list
    static member Empty : 'T list
    member Length : int
    member IsEmpty : bool
    member Head : 'T
    member Tail : 'T list
    member Item :int -> 'T with get
    static member Cons : 'T * 'T list -> 'T list

    interface System.Collections.Generic.IEnumerable<'T>
    interface System.Collections.IEnumerable
```

18.4.2 The Option Type

The following shows the elements of the F# type `FSharp.Core.option` referred to in this specification:

```
[<DefaultAugmentation(false)>]
[<CompilationRepresentation(CompilationRepresentationFlags.UseNullAsT
rueValue)>]    type 'T option =
        | None
        | Some of 'T
        static member None : 'T option
        static member Some : 'T -> 'T option

[<CompilationRepresentation(CompilationRepresentationFlags.Instance)>
]
        member Value : 'T
        member IsSome : bool
        member IsNone : bool
```

18.5 Lazy Computations (Lazy)

See http://msdn.microsoft.com/library/ee353813.aspx

18.6 Asynchronous Computations (Async)

See http://msdn.microsoft.com/library/ee370232.aspx

18.7 Query Expressions

See http://msdn.microsoft.com/library/hh698410

18.8 Agents (MailboxProcessor)

See http://msdn.microsoft.com/library/ee370357.aspx

18.9 Event Types

See http://msdn.microsoft.com/library/ee370608.aspx

18.10 Immutable Collection Types (Map, Set)

See http://msdn.microsoft.com/library/ee353413.aspx

18.11 Text Formatting (Printf)

See http://msdn.microsoft.com/library/ee370560.aspx

18.12 Reflection

See http://msdn.microsoft.com/library/ee353491.aspx

18.13 Quotations

See http://msdn.microsoft.com/library/ee370558.aspx

18.14 Native Pointer Operations

The `FSharp.Core.NativeIntrop` namespace contains functionality for interoperating with native code.

Use of these functions is unsafe, and incorrect use may generate invalid IL code.

Operator or Function Name	Description
`NativePtr.ofNativeInt`	Returns a typed native pointer for a machine address.
`NativePtr.toNativeInt`	Returns a machine address for a typed native pointer.
`NativePtr.add`	Computes an indexed offset from the input pointer.
`NativePtr.read`	Reads the memory that the input pointer references.
`NativePtr.write`	Writes to the memory that the input pointer references.
`NativePtr.get`	Reads the memory at an indexed offset from the input pointer.
`NativePtr.set`	Writes the memory at an indexed offset from the input pointer.
`NativePtr.stackalloc`	Allocates a region of memory on the stack.

18.14.1 Stack Allocation

The `NativePtr.stackalloc` function works as follows. Given

```
stackalloc<ty> n
```

the unmanaged type `ty` specifies the type of the items that will be stored in the newly allocated location, and *n* indicates the number of these items. Taken together, these establish the required allocation size.

The `stackalloc` function allocates `n * sizeof<ty>` bytes from the call stack and returns a pointer of type `nativeptr<ty>` to the newly allocated block. The content of the newly allocated memory is undefined. If *n* is a negative value, the behavior of the function is undefined. If *n* is zero, no allocation is made, and the returned pointer is implementation-defined. If insufficient memory is available to allocate a block of the requested size, the `System.StackOverflowException` is thrown.

Use of this function is unsafe, and incorrect use might generate invalid IL code. For example, the function should not be used in `with` or `finally` blocks in `try`/`with` or `try`/`finally` expressions. These conditions are not checked by the F# compiler, because this primitive is rarely used from F# code.

There is no way to explicitly free memory that is allocated using `stackalloc`. All stack-allocated memory blocks that are created during the execution of a function or member are automatically discarded when that function or member returns. This behavior is similar to that of the `alloca` function, an extension commonly found in C and C++ implementations.

19. Features for ML Compatibility

F# has its roots in the Caml family of programming languages and its core constructs are similar to some other ML-family languages. As a result, F# supports some constructs for compatibility with other implementations of ML-family languages.

19.1 Conditional Compilation for ML Compatibility

F# supports the following constructs for conditional compilation:

```
token start-fsharp-token = "(*IF-FSHARP" | "(*F#"
token end-fsharp-token = "ENDIF-FSHARP*)" | "F#*)"
token start-ml-token = "(*IF-OCAML*)"
token end-ml-token = "(*ENDIF-OCAML*)"
```

F# ignores the `start-fsharp-token` and `end-fsharp-token` tokens. This means that sections marked

```
(*IF-FSHARP  ... ENDIF-FSHARP*)
```

—or—

```
(*F#         ... F#*)
```

are included during tokenization when compiling with the F# compiler. The intervening text is tokenized and returned in the token stream as normal.

In addition, the `start-ml-token` token is discarded and the following text is tokenized as `string`, `_` (any character), and `end-ml-token` until an `end-ml-token` is reached. Comments are not treated as special during this process and are simply processed as "other text". This means that text surrounded by the following is excluded when compiling with the F# compiler:

```
(*IF-CAML*)  ... (*ENDIF-CAML*)
 or (*IF-OCAML*)  ... (*ENDIF-OCAML*)
```

The intervening text is tokenized as "strings and other text" and the tokens are discarded until the corresponding end token is reached. Comments are not treated as special during this process and are simply processed as "other text."

The converse holds when programs are compiled using a typical ML compiler.

19.2 Extra Syntactic Forms for ML Compatibility

The following identifiers are also keywords primarily because they are keywords in OCaml. Although F# reserves several OCaml keywords for future use, the `/mlcompatibility` option enables the use of these keywords as identifiers.

```
token ocaml-ident-keyword =
      asr land lor lsl lsr lxor mod
```

Note: In F# the following alternatives are available. The precedence of these operators differs from the precedence that OCaml uses.

```
asr   >>> (on signed type)
land  &&&
lor   |||
lsl   <<<
lsr   >>> (on unsigned type)
lxor  ^^^
mod   %
sig   begin (that is, begin/end may be used instead of sig/end)
```

F# includes the following additional syntactic forms for ML compatibility:

```
expr :=
     | ...
     | expr.(expr)             // array lookup
     | expr.(expr) <- expr     // array assignment


type :=
     | ...
     | (type,...,type) long-ident // generic type instantiation

module-implementation :=
     | ...
     | module ident = struct ... end

module-signature :=
     | ...
     | module ident : sig ... end
```

An ML compatibility warning occurs when these constructs are used.

Note that the for-expression form `for var = expr1 downto expr2 do expr3` is also permitted for ML compatibility.

The following expression forms

```
expr :=
     | …
     | expr.(expr)             // array lookup
     | expr.(expr) <- expr     // array assignment
```

Are equivalent to the following uses of library-defined operators:

```
e1.(e2)               → (.()) e1 e2
e1.(e2) <- e3         → (.()<-) e1 e2 e3
```

19.3 Extra Operators

F# defines the following two additional shortcut operators:

```
e1 or e2        → (or) e1 e2
e1 & e2         → (&) e1 e2
```

19.4 File Extensions and Lexical Matters

F# supports the use of the `.ml` and `.mli` extensions on the command line. The "indentation awareness off" syntax option is implicitly enabled when using either of these filename extensions.

Lightweight syntax can be explicitly disabled in `.fs`, `.fsi`, `.fsx`, and `.fsscript` files by specifying `#indent "off"` as the first declaration in a file:

```
#indent "off"
```

When lightweight syntax is disabled, whitespace can include tab characters:

```
regexp whitespace = [ ' ' '\t' ]+
```

Appendix A: F# Grammar Summary

This appendix summarizes the grammar of the F# language. The following table describes the notation conventions used in the grammar.

Notation Conventions in Grammar Rules

Notation	Description	Example
element-name$_{opt}$	The $_{opt}$ subscript indicates that `element-name` is optional.	**let rec**$_{opt}$
`...`	An ellipsis indicates that the preceding non-terminal construct and the separator token can repeat any number of times.	*expr* `','` `...` `','` *expr*
keyword	Boldface type identifies a language keyword that must appear verbatim.	**module** *long-ident module-elems*
element-name	Italics identify an element that is defined in the grammar.	*script-fragment* : *module-elems*
`[ char1 - char2 ]`	All ASCII characters in the range from *char1* to *char2*, inclusive.	`[ a - z ]`
`[ ^ char1 - char2 ]`	All ASCI characters except those in the specified range.	`[ ^ A - Z ]`
`‘symbol’` or `“symbol”`	The literal *symbol* is used in the grammar.	`'('`, `"if"`
`(spec)`	Parentheses enclose required individual grammar elements.	`(+|-)`
`$token`	Lexical analysis inserts *$token* as a hidden symbol.	`$app`

A.1 Lexical Grammar

A.1.1 Whitespace

```
whitespace : ' '+

newline :
        '\n'
        '\r' '\n'

whitespace-or-newline :
        whitespace
        newline
```

A.1.2 Comments

block-comment-start : "(*"

block-comment-end : "*)"

end-of-line-comment : "//" [^'\n' '\r']*

A.1.3 Conditional Compilation

if-directive : "#if" *whitespace ident-text*

else-directive : "#else"

endif-directive : "#endif"

A.1.4 Identifiers and Keywords

A.1.4.1 Identifiers

digit-char : [0-9]

letter-char :
'\Lu'
'\Ll'
'\Lt'
'\Lm'
'\Lo'
'\Nl'

connecting-char : '\Pc'

combining-char :
'\Mn'
'\Mc'

formatting-char : '\Cf'

ident-start-char :
letter-char
_

ident-char :
letter-char
digit-char
connecting-char
combining-char
formatting-char
'
_

ident-text : *ident-start-char ident-char**

```
ident :
      ident-text
      `` ( [^'`' '\n' '\r' '\t'] | '`' [^ '`' '\n' '\r' '\t'] )+ ``
```

A.1.4.2 Long Identifiers

```
long-ident :  ident '.' ... '.' ident
long-ident-or-op :
      long-ident '.' ident-or-op
      ident-or-op
```

A.1.4.3 Keywords

```
ident-keyword : one of
      abstract and as assert base begin class default delegate do done
      downcast downto elif else end exception extern false finally for
      fun function global if in inherit inline interface internal lazy let
      match member module mutable namespace new null of open or
      override private public rec return sig static struct then to
      true try type upcast use val void when while with yield

reserved-ident-keyword : one of
      atomic break checked component const constraint constructor
      continue eager fixed fori functor include
      measure method mixin object parallel params process protected pure
      recursive sealed tailcall trait virtual volatile

reserved-ident-formats :
      ident-text ( '!' | '#')
```

A.1.4.4 Symbolic Keywords

```
symbolic-keyword : one of
      let! use! do! yield! return!
      | -> <- . : ( ) [ ] [< >] [| |] { }
      ' # :?> :? :> .. :: := ;; ; =
      _ ? ?? (*) <@ @> <@@ @@>

reserved-symbolic-sequence :
      ~ `
```

A.1.5 Strings and Characters

```
escape-char :  '\' ["\'ntbr]

non-escape-chars :  '\' [^"\'ntbr]

simple-char-char : any char except
      '\n' '\t' '\r' '\b' ' \ "
```

```
unicodegraph-short : '\' 'u' hexdigit hexdigit hexdigit hexdigit

unicodegraph-long :  '\' 'U' hexdigit hexdigit hexdigit hexdigit
                             hexdigit hexdigit hexdigit hexdigit

char-char :
      simple-char-char
      escape-char
      trigraph
      unicodegraph-short

string-char :
      simple-string-char
      escape-char
      non-escape-chars
      trigraph
      unicodegraph-short
      unicodegraph-long
      newline

string-elem :
      string-char
      '\' newline whitespace* string-elem

char : ' char-char '

string : " string-char* "

verbatim-string-char :
      simple-string-char
      non-escape-chars
      newline
      \
      ""

verbatim-string : @" verbatim-string-char* "

bytechar           :  ' simple-or-escape-char 'B

bytearray          :  " string-char* "B

verbatim-bytearray : @" verbatim-string-char* "B

simple-or-escape-char :
      escape-char
      simple-char

simple-char : any char except
      newline, return, tab, backspace,',\,"
```

triple-quoted-string : """ *simple-or-escape-char** """

A.1.6 Numeric Literals

```
digit : [0-9]

hexdigit     :
      digit
      [A-F]
      [a-f]

octaldigit : [0-7]

bitdigit : [0-1]

int : digit+

xint :
      0 (x|X) hexdigit+
      0 (o|O) octaldigit+
      0 (b|B) bitdigit+

sbyte : (int|xint) 'y'

byte  : (int|xint) 'uy'

int16 : (int|xint) 's'

uint16 : (int|xint) 'us'

int32 : (int|xint) 'l'

uint32 :
      (int|xint) 'ul'
      (int|xint) 'u'

nativeint : (int|xint) 'n'

unativeint : (int|xint) 'un'

int64 : (int|xint) 'L'

uint64 :
      (int|xint) 'UL'
      (int|xint) 'uL'

ieee32 :
      float [Ff]
```

```
      xint 'lf'

ieee64 :
      float
      xint 'LF'

bignum : int ('Q' | 'R' | 'Z' | 'I' | 'N' | 'G')

decimal : (float|int) [Mm]

float :
      digit+ . digit*
      digit+ (. digit* )? (e|E) (+|-)? digit+

reserved-literal-formats :
      (xint | ieee32 | ieee64) ident-char+
```

A.1.7 Line Directives

```
line-directive :
      # int
      # int string
      # int verbatim-string
      #line int
      #line int string
      #line int verbatim-string
```

A.1.8 Identifier Replacements

__SOURCE_DIRECTORY__
__SOURCE_FILE__
__LINE__

A.1.9 Operators

A.1.9.1 Operator Names

```
ident-or-op :
      ident
      ( op-name )
      (*)

op-name :
      symbolic-op
      range-op-name
      active-pattern-op-name

range-op-name :
      ..
      .. ..
```

```
active-pattern-op-name :
      | ident | ... | ident |
      | ident | ... | ident | _ |
```

A.1.9.2 Symbolic Operators

```
first-op-char : one of
       !%&*+-./<=>@^|~

op-char :
      first-op-char
      ?

quote-op-left :
      <@ <@@

quote-op-right :
      @> @@>

symbolic-op:
      ?
      ?<-
      first-op-char op-char*
      quote-op-left
      quote-op-right
```

A.1.9.3 Infix and Prefix Operators

The OP marker represents all *symbolic-op* tokens that begin with the indicated prefix, except for tokens that appear elsewhere in the table.

```
infix-or-prefix-op : one of
      +,  -, +., -., %, &, &&

prefix-op :
      infix-or-prefix-op
      ~ ~~ ~~~      (and any repetitions of ~)
      !OP           (all tokens that begin with ! except !=)

infix-op :
      infix-or-prefix-op
      -OP +OP || <OP >OP = |OP &OP ^OP *OP /OP %OP !=
                     (or any of these preceded by one or more '.')
      :=
      ::
      $
      or
      ?
```

A.1.9.4 Constants

```
const :
```

sbyte
int16
int32
int64
byte
uint16
uint32
int
uint64
ieee32
ieee64
bignum
char
string
verbatim-string
triple-quoted-string
bytestring
verbatim-bytearray
bytechar
false
true
()

A.2 Syntactic Grammar

In general, this syntax summary describes full syntax. By default, however, `.fs`, `.fsi`, `.fsx`, and `.fsscript` files support lightweight syntax, in which indentation replaces **begin**/**end** and **done** tokens. This appendix uses **begin**$_{opt}$, **end**$_{opt}$, and **done**$_{opt}$ to indicate that these tokens are omitted in lightweight syntax. Complete rules for lightweight syntax appear in §15.1.

To disable lightweight syntax:

```
#indent "off"
```

When lightweight syntax is disabled, whitespace can include tab characters:

```
whitespace : [ ' ' '\t' ]+
```

A.2.1 Program Format

```
implementation-file :
      namespace-decl-group ... namespace-decl-group
      named-module
      anonynmous-module

script-file : implementation-file

signature-file:
      namespace-decl-group-signature ... namespace-decl-group-signature
```

anonynmous-module-signature
named-module-signature

named-module : **module** *long-ident module-elems*

anonymous-module : *module-elems*

named-module-signature : **module** *long-ident module-signature-elements*

anonymous-module-signature : *module-signature-elements*

script-fragment : *module-elems*

A.2.1.1 Namespaces and Modules

namespace-decl-group :
namespace *long-ident module-elems*
namespace global *module-elems*

module-defn : *attributes*$_{opt}$ **module** *access*$_{opt}$ *ident* = **begin**$_{\mathbf{opt}}$ *module-defn-body* **end**$_{\mathbf{op}}$

module-defn-body : **begin** *module-elems*$_{opt}$ **end**

module-elem :
module-function-or-value-defn
type-defns
exception-defn
module-defn
module-abbrev
import-decl
compiler-directive-decl

module-function-or-value-defn :
attributes$_{opt}$ **let** *function-defn*
attributes$_{opt}$ **let** *value-defn*
attributes$_{opt}$ **let rec**$_{opt}$ *function-or-value-defns*
attributes$_{opt}$ **do** *expr*

import-decl : **open** *long-ident*

module-abbrev : *module ident* = *long-ident*

compiler-directive-decl : **#** *ident string* ... *string*

module-elems : *module-elem* ... *module-elem*

access :
private
internal

public

A.2.1.2 Namespace and Module Signatures

namespace-decl-group-signature : **namespace** *long-ident module-signature-elements*

module-signature : **module** *ident* = **begin**$_{opt}$ *module-signature-body* **end**$_{opt}$

module-signature-element :
  val mutable$_{opt}$ *curried-sig*
  val *value-defn*
  type *type-signatures*
  exception *exception-signature*
  module-signature
  module-abbrev
  import-decl

module-signature-elements :
  begin$_{opt}$ *module-signature-element* ... *module-signature-element* **end**$_{opt}$

module-signature-body : **begin** *module-signature-elements* **end**

type-signature :
  abbrev-type-signature
  record-type-signature
  union-type-signature
  anon-type-signature
  class-type-signature
  struct-type-signature
  interface-type-signature
  enum-type-signature
  delegate-type-signature
  type-extension-signature

type-signatures : *type-signature* ... **and** ... *type-signature*

type-signature-element :
  attributes$_{opt}$ **access**$_{opt}$ **new** : *uncurried-sig*
  attributes$_{opt}$ **member access**$_{opt}$ *member-sig*
  attributes$_{opt}$ **abstract access**$_{opt}$ *member-sig*
  attributes$_{opt}$ **override** *member-sig*
  attributes$_{opt}$ **default** *member-sig*
  attributes$_{opt}$ **static member access**$_{opt}$ *member-sig*
  interface *type*

abbrev-type-signature : *type-name* '=' *type*

union-type-signature : *type-name* '=' *union-type-cases type-extension-elements-signature*$_{opt}$

```
record-type-signature :
      type-name '=' '{' record-fields '}' type-extension-elements-signature_opt

anon-type-signature : type-name '=' begin type-elements-signature end

class-type-signature : type-name '=' class type-elements-signature end

struct-type-signature : type-name '=' struct type-elements-signature end

interface-type-signature : type-name '=' interface type-elements-signature end

enum-type-signature : type-name '=' enum-type-cases

delegate-type-signature :  type-name '=' delegate-sig

type-extension-signature : type-name type-extension-elements-signature

type-extension-elements-signature : with type-elements-signature end
```

A.2.2 Types and Type Constraints

```
type :
      ( type )
      type -> type
      type * ... * type
      typar
      long-ident
      long-ident<type-args>
      long-ident< >
      type long-ident
      type[ , ... , ]
      type typar-defns
      typar :> type
      #type

type-args :=  type-arg, ..., type-arg

type-arg :=
    type
    measure
    static-parameter

atomic-type :
      type : one of
              #type typar ( type ) long-ident long-ident<types>)
```

```
typar :
      _
      'ident
      ^ident

constraint :
      typar :> type
      typar : null
      static-typars : (member-sig )
      typar : (new : unit -> 'T)
      typar : struct
      typar : not struct
      typar : enum<type>
      typar : unmanaged
      typar : delegate<type, type>

typar-defn : attributes_opt typar

typar-defns  : < typar-defn, ..., typar-defn typar-constraints_opt >

typar-constraints : when constraint and ... and constraint

static-typars :
      ^ident
      (^ident or ... or ^ident)
```

A.2.2.1 Equality and Comparison Constraints

```
typar : equality
typar : comparison
```

A.2.2.2 Type Providers

```
static-parameter =
    static-parameter-value
    id = static-parameter-value

static-parameter-value =
    const
    const expr
```

A.2.3 Expressions

```
expr :
      const
```

```
( expr )
begin expr end
long-ident-or-op
expr '.' long-ident-or-op
expr expr
expr(expr)
expr<types>
expr infix-op expr
prefix-op expr
expr.[expr]
expr.[slice-ranges]
expr <- expr
expr , ... , expr
new type expr
{ new base-call object-members interface-impls }
{ field-initializers }
{ expr with field-initializers }
[ expr ; ... ; expr ]
[| expr ; ... ; expr |]
expr { comp-or-range-expr }
[ comp-or-range-expr]
[| comp-or-range-expr |]
lazy expr
null
expr : type
expr :> type
expr :? type
expr :?> type
upcast expr
downcast expr
```

In the following four expression forms, the `in` token is optional if *expr* appears on a subsequent line and is aligned with the *let* token.

```
let function-defn in expr
let value-defn in expr
let rec function-or-value-defns in expr
use ident = expr in expr

fun argument-pats -> expr
function rules
match expr with rules
try expr with rules
try expr finally expr
if expr then expr elif-branches_opt else-branch_opt
while expr do expr done_opt
for ident = expr to expr do expr done_opt
for pat in expr-or-range-expr do expr done_opt
assert expr
```

<@ *expr* @>
<@@ *expr* @@>
%*expr*
%%*expr*
(*static-typars* : (*member-sig*) *expr)*
expr $app *expr*   // equivalent to "*expr*(*expr*)"
expr $sep *expr*   // equivalent to "*expr*; *expr*"
expr $tyapp < *types* > // equivalent to "*expr*<*types*>"

expr< >

exprs : *expr* ',' ... ',' *expr*

expr-or-range-expr :
expr
range-expr

elif-branches : *elif-branch* ... *elif-branch*

elif-branch : **elif** *expr* **then** *expr*

else-branch : **else** *expr*

function-or-value-defn :
function-defn
value-defn

function-defn :
inline$_{opt}$ *access*$_{opt}$ *ident-or-op typar-defns*$_{opt}$ *argument-pats return-type*$_{opt}$ = *expr*
value-defn :
mutable$_{opt}$ *access*$_{opt}$ *pat typar-defns*$_{opt}$ *return-type*$_{opt}$ = *expr*

return-type :
: *type*

function-or-value-defns :
function-or-value-defn **and** ... **and** *function-or-value-defn*

argument-pats: *atomic-pat* ... *atomic-pat*

field-initializer : *long-ident* = *expr*

field- initializer s : *field-initializer* ; ... ; *field-initializer*

object-construction :
type expr
type

base-call :
    object-construction
    object-construction **as** *ident*

interface-impls : *interface-impl* ... *interface-impl*

interface-impl : **interface** *type object-members*$_{opt}$

object-members : **with** *member-defns* **end**

member-defns : *member-defn* ... *member-defn*

A.2.3.1 Computation and Range Expressions

comp-or-range-expr :
    comp-expr
    short-comp-expr
    range-expr

comp-expr :
    let! *pat* = *expr* **in** *comp-expr*
    let *pat* = *expr* **in** *comp-expr*
    do! *expr* **in** *comp-expr*
    do *expr* **in** *comp-expr*
    use! *pat* = expr **in** *comp-expr*
    use *pat* = expr **in** *comp-expr*
    yield! *expr*
    yield *expr*
    return! *expr*
    return *expr*
    if *expr* **then** *comp-expr*
    if *expr* **then** *comp-expr* **else** *comp-expr*
    match *expr* **with** *comp-rules*
    try *comp-expr* **with** *comp-rules*
    try *comp-expr* **finally** *expr*
    while *expr* **do** *expr* done$_{opt}$
    for *ident* = *expr* **to** *expr* **do** *comp-expr* done$_{opt}$
    for *pat* **in** *expr-or-range-expr* **do** *comp-expr* done$_{opt}$
    comp-expr; *comp-expr*
    expr

comp-rule : *pat pattern-guard*$_{opt}$ -> *comp-expr*

comp-rules : '|'$_{opt}$ *comp-rule* '|' ... '|' *comp-rule*

short-comp-expr : **for** *pat* **in** *expr-or-range-expr* -> *expr*

range-expr :

```
      expr .. expr
      expr .. expr .. expr

slice-ranges : slice-range , … , slice-range

slice-range :
      expr
      expr..
      ..expr
      expr..expr
      '*'
```

A.2.3.2 Computation Expressions

```
expr { for ... }
expr { let ... }
expr { let! ... }
expr { use ... }
expr { while ... }
expr { yield ... }
expr { yield! ... }
expr { try ... }
expr { return ... }
expr { return! ... }
```

A.2.3.3 Sequence Expressions

```
seq { comp-expr }
seq { short-comp-expr }
```

A.2.3.4 Range Expressions

```
seq { e1 .. e2 }
seq { e1 .. e2 .. e3 }
```

A.2.3.5 Copy and Update Record Expression

```
{ expr with field-label1 = expr1 ; … ; field-labeln = exprn }
```

A.2.3.6 Dynamic Operator Expressions

```
expr ? ident                  → (?) expr "ident"
expr1 ? (expr2)               → (?) expr1 expr2
expr1 ? ident <- expr2        → (?<-) expr1 "ident" expr2
expr1 ? (expr2) <- expr3      → (?<-) expr1 expr2 expr3
```

`"ident"` is a string literal that contains the text of `ident`.

A.2.3.7 AddressOf Operators

```
&expr
```

&&*expr*

A.2.3.8 Lookup Expressions

```
e1.[eargs]                    → e1.get_Item(eargs)
e1.[eargs] <- e3              → e1.set_Item(eargs, e3)
```

A.2.3.9 Slice Expressions

```
e1.[sliceArg1, ,,, sliceArgN]         → e1.GetSlice( args1,…,argsN)
e1.[sliceArg1, ,,, sliceArgN] <- expr → e1.SetSlice( args1,…,argsN, expr)
```

where each sliceArgN is a *slice-range* and translated to argsN (giving one or two args) as follows:

```
      *            → None, None
      e1..         → Some e1, None
      ..e2         → None, Some e2
      e1..e2       → Some e1, Some e2
      idx          → idx
```

A.2.3.10 Shortcut Operator Expressions

```
expr1 && expr2                → if expr1 then expr2 else false
expr1 || expr2                → if expr1 then true else expr2
```

A.2.3.11 Deterministic Disposal Expressions

```
use ident = expr1 in expr2
```

A.2.4 Patterns

```
rule : pat pattern-guard_opt -> expr

pattern-guard : when expr

pat :
      const
      long-ident pat-param_opt pat_opt
      _
      pat as ident
      pat '|' pat
      pat '&' pat
      pat :: pat
      pat : type
      pat,...,pat
      (pat)
      list-pat
      array-pat
      record-pat
```

```
       :? atomic-type
       :? atomic-type as ident
       null
       attributes pat

list-pat :
       [ ]
       [ pat ; ... ; pat ]

array-pat :
       [| |]
       [| pat ; ... ; pat |]

record-pat : { field-pat ; ... ; field-pat }

atomic-pat :
    pat      one of
               const  long-ident  list-pat  record-pat  array-pat  (pat)
             :? atomic-type
                       null  _ _


field-pat : long-ident = pat

pat-param :
       const
       long-ident
       [ pat-param ; ... ; pat-param ]
       ( pat-param, ..., pat-param )
       long-ident pat-param
       pat-param : type
       <@ expr @>
       <@@ expr @@>
       null

pats :  pat , ... , pat

field-pats : field-pat ; ... ; field-pat

rules : '|'opt rule '|' ... '|' rule
```

A.2.5 Type Definitions

```
type-defn :
       abbrev-type-defn
       record-type-defn
       union-type-defn
       anon-type-defn
```

class-type-defn
struct-type-defn
interface-type-defn
enum-type-defn
delegate-type-defn
type-extension

type-name : *attributes*$_{opt}$ *access*$_{opt}$ *ident* *typar-defns*$_{opt}$

abbrev-type-defn : *type-name* = *type*

union-type-defn : *type-name* '=' *union-type-cases* *type-extension-elements*$_{opt}$

union-type-cases : '|'*opt* *union-type-case* '|' ... '|' *union-type-case*

union-type-case : *attributes*$_{opt}$ *union-type-case-data*

union-type-case-data :
ident -- nullary union case
ident **of** *union-type-field* * ... * *union-type-field* -- n-ary union case
ident : *uncurried-sig* -- n-ary union case

union-type-field :
type -- unnamed union type field
ident : *type* -- named union type field

anon-type-defn :
type-name *primary-constr-args*$_{opt}$ *object-val*$_{opt}$ '=' **begin** *class-type-body* **end**

record-type-defn : *type-name* = '{' *record-fields* '}' *type-extension-elements*$_{opt}$

record-fields : *record-field* ; ... ; *record-field* ;$_{opt}$

record-field : *attributes*$_{opt}$ **mutable**$_{opt}$ *access*$_{opt}$ *ident* : *type*

class-type-defn :
type-name *primary-constr-args*$_{opt}$ *object-val*$_{opt}$ '=' **class** *class-type-body* **end**

as-defn : **as** *ident*

class-type-body :
begin$_{opt}$ *class-inherits-decl*$_{opt}$ *class-function-or-value-defns*$_{opt}$ *type-defn-elements*$_{opt}$ **end**$_{opt}$

class-inherits-decl : **inherit** *type* *expr*$_{opt}$

class-function-or-value-defn :
    attributes$_{opt}$ **static**$_{opt}$ **let rec**$_{opt}$ *function-or-value-defns*
    attributes$_{opt}$ **static**$_{opt}$ **do** *expr*

struct-type-defn :
  type-name primary-constr-args$_{opt}$ *as-defn*$_{opt}$ '=' **struct** *struct-type-body*
end

struct-type-body : *type-defn-elements*

interface-type-defn : *type-name* '=' **interface** *interface-type-body* **end**

interface-type-body : *type-defn-elements*

exception-defn :
    attributes$_{opt}$ **exception** *union-type-case-data*
    attributes$_{opt}$ **exception** *ident = long-ident*

enum-type-defn : *type-name* '=' *enum-type-cases*

enum-type-cases : '|'$_{opt}$ *enum-type-case* '|' ... '|' *enum-type-case*

enum-type-case : *ident* '=' *const*

delegate-type-defn : *type-name* '=' *delegate-sig*

delegate-sig : delegate **of** *uncurried-sig*

type-extension : *type-name type-extension-elements*

type-extension-elements : **with** *type-defn-elements* **end**

type-defn-element :
    member-defn
    interface-impl
    interface-signature

type-defn-elements : type-defn-element ... type-defn-element

primary-constr-args : *attributes*$_{opt}$ *access*$_{opt}$ (*simple-pat, ... , simplepat*)

simple-pat :
  | *ident*
  | *simple-pat* : *type*

additional-constr-defn :
    attributes$_{opt}$ *access*$_{opt}$ **new** *pat as-defn = additional-constr-expr*

additional-constr-expr :
    stmt ';' *additional-constr-expr*
    additional-constr-expr **then** *expr*
    if *expr* **then** *additional-constr-expr* **else** *additional-constr-expr*
    let val-decls **in** *additional-constr-expr*
    additional-constr-init-expr

additional-constr-init-expr :
    '{' *class-inherits-decl field-initializers* '}'
    new *type expr*

member-defn :
    attributes$_{opt}$ **static**$_{opt}$ **member** *access$_{opt}$ method-or-prop-defn*
    attributes$_{opt}$ **abstract member**$_{opt}$ *access$_{opt}$ member-sig*
    attributes$_{opt}$ **override** *access$_{opt}$ method-or-prop-defn*
    attributes$_{opt}$ **default** *access$_{opt}$ method-or-prop-defn*
    attributes$_{opt}$ **static**$_{opt}$ **val mutable**$_{opt}$ *access$_{opt}$ ident : type*
    additional-constr-defn

method-or-prop-defn :
    ident$_{opt}$ function-defn
    ident$_{opt}$ value-defn
    ident$_{opt}$ ident **with** *function-or-value-defns*
    member *ident* = *exp*
    member *ident* = *exp* **with get**
    member *ident* = *exp* **with set**
    member *ident* = *exp* **with get,set**
    member *ident* = *exp* **with set,get**

member-sig :
    ident typar-defns$_{opt}$: *curried-sig*
    ident typar-defns$_{opt}$: *curried-sig* **with get**
    ident typar-defns$_{opt}$: *curried-sig* **with set**
    ident typar-defns$_{opt}$: *curried-sig* **with get,set**
    ident typar-defns$_{opt}$: *curried-sig* **with set,get**

curried-sig : *args-spec* -> ... -> *args-spec* -> *type*

uncurried-sig : *args-spec* -> *type*

args-spec : *arg-spec* * ... * *arg-spec*

arg-spec : *attributes$_{opt}$ arg-name-spec$_{opt}$ type*

arg-name-spec : ?$_{opt}$ *ident* :

interface-spec : **interface** *type*

A.2.5.1 Property Members

```
static_opt member ident._opt ident = expr
static_opt member ident._opt ident with get pat = expr
static_opt member ident._opt ident with set pat_opt pat= expr
static_opt member ident._opt ident with get pat = expr and set pat_opt pat =
expr
static_opt member ident._opt ident with set pat_opt pat = expr and get pat =
expr
```

A.2.5.2 Method Members

```
static_opt member ident._opt ident pat1 ... patn = expr
```

A.2.5.3 Abstract Members

```
abstract access_opt member-sig

member-sig :
    ident typar-defns_opt : curried-sig
    ident typar-defns_opt : curried-sig with get
    ident typar-defns_opt : curried-sig with set
    ident typar-defns_opt : curried-sig with get, set
    ident typar-defns_opt : curried-sig with set, get

curried-sig : args-spec_1 -> ... -> args-spec_n -> type
```

A.2.5.4 Implementation Members

```
override ident.ident pat1 ... patn = expr
default ident.ident pat1 ... patn = expr
```

A.2.6 Units Of Measure

```
measure-literal-atom :
      long-ident
      ( measure-literal-simp )

measure-literal-power :
      measure-literal-atom
      measure-literal-atom ^ int32

measure-literal-seq :
      measure-literal-power
      measure-literal-power measure-literal-seq
```

```
measure-literal-simp :
      measure-literal-seq
      measure-literal-simp * measure-literal-simp
      measure-literal-simp / measure-literal-simp
      / measure-literal-simp
      1

measure-literal :
      _
      measure-literal-simp

const :
      ...
      sbyte < measure-literal >
      int16 < measure-literal >
      int32 < measure-literal >
      int64 < measure-literal >
      ieee32 < measure-literal >
      ieee64 < measure-literal >
      decimal < measure-literal >

measure-atom :
      typar
      long-ident
      ( measure-simp )

measure-power :
      measure-atom
      measure-atom ^ int32

measure-seq :
      measure-power
      measure-power measure-seq

measure-simp :
      measure-seq
      measure-simp * measure-simp
      measure-simp / measure-simp
      / measure-simp
      1

measure :
      _
      measure-simp
```

A.2.7 Custom Attributes and Reflection

attribute : *attribute-target*:$_{opt}$ *object-construction*

```
attribute-set : [< attribute ; ... ; attribute >]

attributes : attribute-set ... attribute-set

attribute-target :
      assembly
      module
      return
      field
      property
      param
      type
      constructor
      event
```

A.2.8 Compiler Directives

Compiler directives in non-nested modules or namespace declaration groups:

```
# id string ... string
```

A.3 ML Compatibility Features

A.3.1 Conditional Compilation

```
start-fsharp-token :
      "(*IF-FSHARP"
      "(*F#"
end-fsharp-token :
      "ENDIF-FSHARP*)"
      "F#*)"
start-ml-token : "(*IF-OCAML*)"
end-ml-token : "(*ENDIF-OCAML*)"
```

A.3.2 Extra Syntactic Forms

```
ocaml-ident-keyword : one of
      asr land lor lsl lsr lxor mod

expr :
      ...
      expr.(expr)           // array lookup
      expr.(expr) <- expr   // array assignment

type :
      ...
      (type,...,type) long-ident // generic type instantiation
```

module-implementation :

```
    ...
    module ident = struct ... end
```

module-signature :

```
    ...
    module ident : sig ... end
```

A.3.3 Extra Operators

e_1 `or` e_2 → `(or)` e_1 e_2

e_1 `&` e_2 → `(&)` e_1 e_2

References

Ecma International. *Standard ECMA-335,* Common Language Infrastructure (CLI)
http://www.ecma-international.org/publications/standards/Ecma-335.htm

The French National Institute for Research in Computer Science and Control (INRIA). *The Caml Language*.
http://caml.inria.fr/

Microsoft Corporation. *The C# Language Specification*
http://msdn.microsoft.com/library/ms228593.aspx

Glossary

This section contains terminology that is specific to F#. It provides a reference for terms that are used elsewhere in this document.

A

abstract member
A member in a type that represents a promise that an object will provide an implementation for a dispatch slot.

accessibility
The program text that has access to a particular declaration element or member. You can specify accessibilities on declaration elements in namespace declaration groups and modules, and on members in types. F# supports **public**, **private**, and **internal** accessibility.

and **pattern**
A pattern that consists of two patterns joined by an ampersand (&). An **and** pattern matches the input against both patterns and binds any variables that appear in either pattern.

anonymous implementation file
A file that lacks either a leading `module` or `namespace` declaration. Only the scripts and the last file within an implementation group for an executable image can be anonymous. An anonymous implementation file can contain module definitions that are implicitly placed in a module, the name of which is implicit from the name of the source file that contains the module.

anonymous variable type with a subtype constraint
A type in the form `#type`. This is equivalent to `'a when 'a :> type` where `'a` is a fresh type inference variable.

anonymous signature file
A signature file that does not have either a leading `module` or `namespace` declaration. The name of the implied module signature is derived from the file name of the signature file.

anonymous variable type
A type in the form `_`.

application expression
An expression that involves variable names, dot-notation lookups, function applications, method applications, type applications, and item lookups

assignment expression
An expression in the form $expr_1$ `<-` $expr_2$.

arity
The number of arguments to a method or function.

array expression
An expression in the form `[|`$expr_1$`;...;` $expr_n$ `|]`.

array pattern
The pattern `[|pat ; ... ; pat|]`, which matches arrays of a specified length.

array sequence expression
An expression that describes a series of elements in an array, in one of the following forms:

```
[| comp-expr |]
[| short-comp-expr |]
[| range-expr |]
```

***as* pattern**
A pattern in the form *pat* **as** *ident.* The **as** pattern binds the name *ident* to the input value and matches the input against the pattern.

automatic generalization
A technique that, during type inference, automatically makes code generic when possible, which means that the code can be used on many types of data.

B

base type declarations
A declaration that represents an additional, encapsulated type that is supported by any values that are formed by using the type definition.

block comments
Comments that are delimited by `(*` and `*)`, can span more than one line, and can be nested.

block expression
An expression in the form `begin expr end`.

C

class type definition
The definition of a type that encapsulates values that are themselves constructed by using one or more object constructors. A class type typically describes an object that can have properties, methods, and events.

coercion
The changing of data from one type to another.

comparison constraint
A constraint of the form `typar : comparison`.

compiled name
The name that appears in the compiled form of an F# program for a symbolic operator or certain symbolic keywords.

conditional expression
An expression in the following form

```
if expr1a then expr1b
elif expr3a then expr2b
…
elif exprna then exprnb
else exprlast
```

The `elif` and `else` branches are optional.

cons pattern

The pattern `pat :: pat`, which is used to decompose a list into two parts: the head, which consists of the first element, and the tail, which contains the remaining elements.

constraint

See **type constraint**.

constraint solving

The process of reducing constraints to a normalized form so that variables can be solved by deducing equations.

copy-and-update record expression

An expression in the following form:

`{ expr with field-label1 = expr1 ; … ; field-labeln = exprn }`

current inference constraints

The set of type constraints that are in effect at a particular point in the program as a result of type checking and elaboration.

curried method members

Arguments to a method that are written in an interated form.

custom attribute

A class that encapsulates information, often metadata that describes or supplements an F# declaration. Custom attributes derive from **System.Attribute** in the .NET framework and can be used in any language that targets the common language runtime.

D

default constructor constraint

A constraint of the form `typar : (new : unit -> 'T)`.

default initialization

The practice of setting the values of particular types to zero at the beginning of execution. Unlike many programming languages, F# performs default initialization in only limited circumstances.

definitely equivalent types

Static types that match exactly in definition, form, and number; or variable types that refer to the same declaration or are the same type inference variable.

delayed expression

An expression in the form `lazy expr`, which is evaluated on demand in response to a `.Value` operation on the lazy value.

delegate constraint.

A constraint of the form `typar : delegate<tupled-arg-type, return-type>`.

dispatch slot

A key representing part of the contract of an interface or class type. Each object that implements the type provides a dictionary mapping dispatch slots to member implementations.

dynamic type test pattern

The patterns `:? type` and `:? type as ident`, which match any value whose runtime type is the given type or a subtype of the given type.

E

elaborated expression

An expression that the F# compiler generates in a simpler, reduced language. An elaborated expression contains a fully resolved and annotated form of the source expression and carries more explicit information than the source expression.

enumerable extraction

The process of getting sequential values of a static type by using CLI library functions that retrieve individual, enumerable values.

enumeration constraint

A constraint in the form `typar : enum<underlying-type>`, which limits the type to an enumeration of the specified underlying type.

equality constraint.

A constraint in the form `typar : equality`, which limits the type to one that supports equality operations.

event

A configurable object that has a set of callbacks that can be triggered, often by some external action such as a mouse click or timer tick. The F# library supports the `FSharp.Control.IEvent<_,_>` type and the `FSharp.Control.Event` module to support the use of events.

F

F# Interactive

An F# dynamic compiler that runs in a command window and executes script fragments as well as complete programs.

feasible coercion

Indicates that one type either coerces to another, or could become coercible through the addition of further constraints to the current inference constraints.

feasibly equivalent types

Types that are not definitely equivalent but could become so by the addition of further constraints to the current inference constraints.

floating type variable environment

The set of types that are currently defined, for use during type inference.

fresh type

A static type that is formed from a *fresh type inference variable.*

fresh type inference variable

A variable that is created during type inference and has a unique identity.

function expression

An expression of the form `fun` pat_1 `...` pat_n `->` *expr*.

function value

The value that results at runtime from the evaluation of function expressions.

G

generic type definition

A type definition that has one or more generic type parameters. For example: `System.Collections.Generic.Dictionary<'Key,'Value>`.

guarded pattern matching rule

A rule of the form *pat* `when` *expr* that occurs as part of a pattern matching expression such as `match` $expr_0$ `with` $rule_1$ `->` $expr_1$ `|` … `|` $rule_n$ `->` $expr_n$. The guard expression *expr* is executed only if the value of $expr_0$ successfully matches the pattern *pat*.

I

identifier

A sequence of characters that is enclosed in ``` `` `` ``` double-backtick marks, excluding newlines, tabs, and double-backtick pairs themselves.

immutable value

A named value that cannot be changed.

imperative programming

One of several primary programming paradigms; others include declarative, functional, procedural, among others. An imperative program consists of a sequence of actions for the computer to perform, and the statements change the state of the program.

implementation member

An abstract member that implements a dispatch slot or CLI property.

import declaration

A declaration that makes the elements of another namespace's declarations and modules accessible by the use of unqualified names. Import definitions can appear in namespace declaration groups and module definitions.

inference type variable

A type variable that does not have a declaration site.

initialization constant expression

An expression whose elaborated form is determined to cause no observable initialization effect.

instance member

A member that is declared without `static`.

interface type definition

A declaration that represents an encapsulated type that specifies groups of related members that other classes implement.

K

keyword

A word that has a defined meaning in F# and is used as part of the language itself.

L

lambda expression

See *function expression*.

lightweight syntax

A simplified, indentation-aware syntax in which lines of code that form a sequence of declarations are aligned on the same column, and the *in* and ;; separators can be omitted. Lightweight syntax is the default for all F# code in files with extension `.fs`, `.fsx`, `.fsi` and `.fsscript`.

list

An F# data structure that consists of a sequence of items. Each item contains a pointer to the next item in the sequence.

list expression

An expression of the form `[`*expr*$_1$`;...;` *expr*$_n$`]`.

list pattern

A data recognizer pattern that describes a list. Examples are *pat* `::` *pat*, which matches the 'cons' case of F# list values; `[]`, which matches the empty list; and `[`*pat* `; ... ;` *pat*`]`, which represents a series of `::` and empty list patterns.

list sequence expression

An expression that evaluates to a sequence that is essentially a list. List sequence expressions can have the following forms:

```
[ comp-expr ]
[ short-comp-expr ]
[ range-expr ]
```

literal constant expression

An expression that consists of a simple constant expression or a simple compile-time computation.

M

member

A function that is associated with a type definition or with a value of a particular type. Member definitions can be used in type definitions. F# supports property members and method members.

member constraint

A constraint that specifies the signature that is required for a particular member function. Member constraints have the form `(`*typar* `or ... or` *typar*`) : (`*member-sig*`).`

member signature
The "footprint" of a property or method that is visible outside the defining module.

method member
An operation that is associated with a type or an object.

module
A named collection of declarations such as values, types, and function values.

module abbreviation
A statement that defines a local name for a long identifier in a module. For example:

```
module Ops = FSharp.Core.Operators
```

module signature
A description of the contents of a module that the F# compiler generates during type inference. The module signature describes the externally visible elements of the module.

N

name resolution environment
The collection of names that have been defined at the current point, which F# can use in furthy type inference and checking. The name resolution environment includes namespace declaration groups from imported namespaces in addition to names that have been defined in the current code.

named type
A type that has a name, in the form *long-ident*<ty_1,…, ty_n>, where *long-ident* resolves to a a type definition that has formal generic parameters and formal constraints.

namespace
A way of organizing the modules and types in an F# program, so that they form logical groups that are associated with a name. Identifiers must be unique within a namespace.

namespace declaration group
The basic declaration unit within an F# implementation file. It contains a series of module and type definitions that contribute to the indicated namespace. An implementation can contain multiple namespace declaration groups.

namespace declaration group signature
The "footprint" of a namespace declaration group, which describes the externally visible elements of the group.

null expression
An expression of the form `null`.

nullness constraint
A constraint in the form *typar*`: null`, which indicates that the type must support the Null literal.

null pattern
The pattern `null`, which matches the values that the CLI value `null` represents.

numeric literal
A sequence of Unicode characters or an unsigned byte array that represents a numeric value.

O

object construction expression

An expression in the form `new ty(e1 ... en)`, which constructs a new instance of a type, usually by calling a constructor method on the type.

object constructor

A member of a class that can create a value of the type and partially initialize an object. The *primary constructor* contains function and value definitions that appear at the start of the class definition, and its parameters appear in parentheses immediately after the type name. Any *additional object constructors* are specified with the ***new*** keyword, and they must call the primary constructor.

object expression

An expression that creates a new instance of a dynamically created, anonymous object type that is based on an existing base type, interface, or set of interfaces.

offside lines

Lines that occur at column positions in lightweight syntax. Offside lines are introduced by other structured constructs, such asthe `=` token associated with `let`, and the first token after **then** in an **if/then/else** construct.

offside rule

Another term for lightweight or indentation-aware syntax.

P

parenthesized expression

An expression in the form `(expr)`.

pattern matching

A switch construct that supports branched control flow and the definition of new values.

pipeline operator

The |> operator, which directs the value of one function to be input to the next function in a pipeline.

property member

A function in a type that gets or sets data about the type.

Q

quoted expression

An expression that is delimited in such a way that it is not compiled as part of your program, but instead is compiled into an object that represents an F# expression.

R

range expression

An expression that generates a sequence over a given range of values.

record construction expression

An expression that builds a record, in the form `{ field-initializer1 ; … ; field-initializern }`.

record pattern

The pattern `{ long-ident1 = pat1; ... ; long-identn = patn}`.

recursive definition

A definition in which the bound functions and values can be used within their own definitions.

reference type constraint

A constraint of the form `typar : not struct`.

reference type

A class, interface delegate, function, tuple, record, or union type. A type is a reference type if its outermost named type definition is a reference type, after expanding type definitions.

referenced assemblies

The existing assemblies to which an F# program makes static references.

rigid type variable

A type variable that refers to one or more explicit type parameter definitions.

runtime type

An object of type `System.Type` that is the runtime representation of some or all of the information carried in type definitions and static types. The runtime type associated with an objects is accessed by using the `obj.GetType()` method, which is available on all F# values.

S

script

A fragment of an F# program that can be run in F# Interactive.

sealed type definition

A type definition that is concrete and cannot be extended. Record, union, function, tuple, struct, delegate, enum, and array types are all sealed types, as are class types marked with the `SealedAttribute` attribute.

sequence expression

An expression that evaluates to a sequence of values, in one of the following forms

```
seq { comp-expr }
seq { short-comp-expr }
```

sequential execution expression

An expression that represents the sequential execution of one statement followed by another. The expression has the form `expr1; expr2`.

signature file

A file that contains information about the public signatures and accessibility of a set of F# program elements.

simple constant expressions

A numeric, string, Boolean, or unit constant.

single-line comments
A comment that begins with `//` and extends to the end of a line.

slice expression
An expression that describes a subset of an array.

static type
The type that is inferred for an expression as the result of type checking, constraint solving, and inference.

static member
A member that is prefixed by `static` and is associated with the type, rather than with any particular object.

statically resolved type variable
A type parameter in the form `^ident`. Such a parameter is replaced with an actual type at compile time instead of runtime.

string
A type that represents immutable text as a sequence of Unicode characters.

string literal
A Unicode string or an unsigned byte array that is treated as a string.

strong name
A cryptographic signature for an assembly that provides a unique name, guarantees the publisher over subsequent versions, and ensures the integrity of the contents.

subtype constraint
A constraint of the form *`typar :> type`*, which limits the type of *`typar`* to the specified *`type`*, or to a type that is derived from that *`type`*. If *`type`* is an interface, *`typar`* must implement the interface.

symbolic keyword
A symbolic or partially symbolic character sequence that is treated as a keyword.

structural type
A record, union, struct, or exception type definition.

symbolic operator
A user-defined or library-defined function that has one or more symbols as a name.

syntactic sugar
Syntax that makes code easier to read or express; often a shortcut way of expressing a more complicated relationship for which one or more other syntactic options exist.

syntactic type
The form of a type specification that appears in program source code, such as the text "`option<_>`". Syntactic types are converted to static types during the process of type checking and inference.

T

tuple
An ordered collection of values that is treated as an atomic unit. A tuple allows you to keep data organized by grouping related values together, without introducing a new type.

tuple expression
An expression in the form $expr_1$, ..., $expr_n$, which describes a tuple value.

tuple type
A type in the form ty_1 * ... * ty_n, which defines a tuple. The elaborated form of a tuple type is shorthand for a use of the family of F# library types `System.Tuple<_,...,_>`.

type abbreviation
An alias or alternative name for a type.

type annotation
An addition to an expression that specifies the type of the expression. A type annotation has the form *expr : type*.

type constraint
A restriction on a generic type parameter or type variable that limits the types that may be used to instantiate that parameter. Example type constraint include subtype constraints, null constraints, value type constraints, comparison constraints and equality constraints.

type definition kind
A class, interface, delegate, struct, record, union, enum, measure, or abstract type.

The kind of type refers to the kind of its outermost named type definition, after expanding abbreviations.

type extension
A definition that associates additional dot-notation members with an existing type.

type function
A value that has explicit generic parameters but arity `[]`—that is, it has no explicit function parameters.

type inference
A feature of F# that determines the type of a language construct when the type is not specified in the source code.

type inference environment
The set of definitions and constraints that F# uses to infer the type of a value, variable, function, or parameter, or similar language construct.

type parameter definition
In a generic function, method, or type, a placeholder for a specific type that is specified when the generic function, method, or type is instantiated.

type provider
A component that provides new types and methods that are based on the schemas of external information sources.

type variable
A variable that represents a type, rather than data.

U

undentation
The opposite of *indentation*.

underlying type

The type of the constant values of an enumeration. The underlying type of an enum must be sbyte, int16, int32, int64, byte, uint16, uint32, uint64, or char.

union pattern

The pattern `pat | pat` attempts to match the input value against the first pattern, and if that fails matches instead the second pattern. Both patterns must bind the same set of variables with the same types.

union type

A type that can hold a value that satisfies one of a number of named cases.

unit of measure

A construct similar to a type that represents a measure, such as kilogram or meters per second. Like types, measures can appear as parameters to other types and values, can contain variables, and are checked for consistency by the type-checker. Unlike types, however, measures are erased at runtime, have special equivalence rules, and are supported by special syntax.

unmanaged type

The primitive types (`sbyte`, `byte`, `char`, `nativeint`, `unativeint`, `float32`, `float`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, and `decimal`), enumeration types, and `nativeptr<_>`, or a non-generic structure whose fields are all unmanaged types.

unmanaged constraint

An addition to a type parameter that limits the type to an unmanaged type.

V

value signature

The "footprint" of a value in a module, which indicates that the value exists and is externally visible.

value type

A type that is allocated on the stack or inline in an object or array. Value types include primitive integers, floating-point numbers, and any value of a struct type.

value type constraint

A constraint of the form `typar : struct`, which limits the type of `typar` to a .NET value type.

variable type

A type of the form `'ident`, which represents the name of another type.

W

wildcard pattern

The underscore character _, which matches any input.

Index