The F# 1.9.6.16 Draft Language Specification

Note: This documentation is an informal specification of the 1.9.6.16 “Beta1” release of F# made by Microsoft Research and the Microsoft Developer Division in May 2009. It is not official Microsoft documentation. Aspects of the design of any future Microsoft and Microsoft Research releases of F# and its libraries are subject to change.

Discrepancies may exist between this specification and the 1.9.6.16 implementation. Some of these are noted as comments in this document. If you note further discrepancies please contact us and we’ll gladly address the issue in future releases.

The F# team are always very grateful for feedback on this specification and on both the design and implementation of F#. You can submit feedback by emailing fsbugs@microsoft.com.

Many thanks to the F# user community for their helpful feedback on the document so far.

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 type-safe functional programming language for the .NET framework and other implementations of the Ecma 335 Common Language Infrastructure (CLI) specification. F# was partially inspired by the OCaml language and shares some common core constructs with it.

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

System.Console.ReadKey(true)
```

Over the next few sections, we'll look at this program, line by line, describing some important aspects of F# along the way. You can explore this program either by compiling it as a project in a development environment such as Visual Studio, or by manually invoking the F# command line compiler `fsc.exe`, or by using F# Interactive, the dynamic compiler that is part of the F# distribution.

Lightweight Syntax

The F# language uses simplified, indentation-aware syntactic forms when the "lightweight syntax option" is enabled. This is the default for F# code in files with extension `.fs`, `.fsx`, `.fsi` and `.fsscript`. We recommend that you use this syntax option on since most F# code snippets you find will either declare it, or assume that it has been declared.

> F# has its roots in the Caml family of programming languages and can cross-compile some simple OCaml programs unmodified. This syntax option may be enabled by using file extensions `.ml` or `.mli`, or by using `#indent "off"` as the first tokens in a file.

Making Data Simple

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

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

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

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

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

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

F# supports a number of other highly effective techniques to simplify the process of modelling and manipulating data such as **tuples**, **options**, **records**, **discriminated unions** and **sequence expressions**. A tuple is an ordered collection of values treated like an atomic unit. In many languages, if you want to pass around a group of

related values as a single entity, you would 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, simply enclose the group of values in parentheses and separate the individual components by commas.

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

> 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. This means that, for most types, once a value of that type is created, that value can't be changed. Immutability has many benefits: it prevents many classes of bugs, and immutable data is inherently "thread safe", which makes the process of parallelizing code simpler.

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 to specify type information for a function declaration. However F# typically infers this type information for you. This is referred to as *type inference*.

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

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

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

Since `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, then the F# compiler infers that the parameter `y` must also be a string. Thus the result of `x + y` is the concatenation of both strings. Without the type annotation, F# would not have known which version of the (+) operator was desired, and would have defaulted to using 'int'.

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, here is how to define a function that returns a new tuple with its two values 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 `'a` and `'b` represent “type variables”, and the function `swap` is “generic”. Type inference greatly simplifies the process of writing reusable code fragments.

Functional Programming

Continuing the sample, we have a list of integers `numbers` and a function `square`, and we want to create a new list where each item is the result of calling 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 **lambda expression**, that takes a single parameter `x` and returns the result "`x % 2 = 0`" (a Boolean value that says whether `x` is even).

Note that 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 is a powerful switch construct that allows you to branch control flow. Pattern matching also allows you to also bind new values. For example, we can match against list elements joined together.

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

Pattern matching can also be used as a control construct, e.g. 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"
    | :? Exception -> "x is an exception"
```

Another way function values are used in F# is in concert 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 String(Array.rev (x.ToCharArray()))
```

We can use those 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** in programming, 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.

Imperative Programming

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

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

The F# library function `printf` is a simple and type-safe way to print text to the console window. Consider this example which prints an integer, 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 "holes" for integers, floats, and strings. The `%A` format may be used to print arbitrary data types (including lists).

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

.NET Interoperability and CLI Fidelity

The last line in the sample program calls the CLI function `System.Console.ReadKey` to pause the program before it closed.

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

Since F# is built on top of CLI implementations you can call any CLI library from F#. Furthermore, all F# components can be readily used from other CLI languages.

Parallel and Asynchronous Programming

The F# and CLI libraries include support for parallel and asynchronous programming. One way to write these kinds of programs is to use F#-facing libraries such as F# **asynchronous workflows**. For example, the code below is similar to our original script except it computes the Fibonacci function (using a technique that will take some time), and schedule 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.Run (Async.Parallel [ for i in 0..40 -> async { return fib(i) } ])

printfn "N^2 = %A" fibs

System.Console.ReadKey(true)
```

While the above technique shows naive CPU parallelism, the combination of asynchronous workflows and other CLI libraries can be used to implement **task parallelism**, **I/O parallelism** and **message passing agents**.

Stronger Typing for Floating Point Code

F# extends the reach of type checking and type inference to floating-point intensive domains through **units of measure inference and checking**. This allows you to typecheck programs that manipulate floating point

numbers representing physical and abstract quantities, without losing any performance in your compiled code. You can think of this feature as providing a type system for floating point code.

```
[<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 aren't 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>`.

Object Oriented Programming and Code Organization

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

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

```
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, e.g. [('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.of_seq

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

    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 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
```

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

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

Modules are a simple way to encapsulating code when you are rapid prototyping without needing to spend the time to design a strict object-oriented type hierarchy. In the example below we take our original script and place a portion of it 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 devices 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 About This Draft Specification

This draft specification describes the F# language through a mixture of informal and semiformal techniques. All examples in this specification are given assuming the use of the lightweight syntax option unless otherwise specified.

Regular expressions are given in the usual notation, e.g.

```
[A-Za-z]+
```

String of characters that are clearly not a regular expression are written verbatim (e.g. `#if`). Where appropriate quotes have been used to indicate concrete syntax, if the symbol being quoted is also used in the specification of grammar itself, e.g., `'<'` and `'|'`. For example, the regular expression

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

matches `(+)` or `(-)`. Derived, named regular expressions are defined as follows:

```
regexp letter-char = [A-Za-z]+
```

Unicode character classes are referred to by their abbreviation, e.g. `\Lu` for any uppercase letter.

Regular expressions are usually used to specify tokens.

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

In the grammar rules, the notation `[ ... ]` indicates an optional element. The notation `...` indicates repetition of the preceding non-terminal construct, with the optional repetition extending to connector e.g., *`expr`* `','` `...` `','` *`expr`* means a sequence of one or more *`expr`* elements separated by commas.

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

2 Program Structure

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

- Source code files, with extensions `.fs`, `.fsi`, `.ml`, `.mli`, `.fsx`, `.fsscript`.
    - Files with extension `.fs`, `.ml` must conform to grammar element *`implementation-file`* in §13.1.
    - Files with extension `.fsi`, `.mli` must conform to grammar element *`signature-file`* in §13.2.
    - Files with extension `.fsx` and `.fsscript` are processed just as for those with extension `.fs` except that the conditional compilation symbol `INTERACTIVE` is defined instead of `COMPILED`, the assembly `FSharp.Compiler.Interactive.Settings.dll` is referenced by default, and the namespace `Microsoft.FSharp.Compiler.Interactive.Settings` is opened by default.
- Source code fragments (for F# Interactive). These must conform to grammar element *`module-elems`*. Source code fragments can be separated by `;;` tokens.
- Assembly references (e.g. via command line arguments or interactive directives)
- Compilation parameters (e.g. via command line arguments or interactive directives)
- Interactive directives such as `#time`.

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

- **Decoding.** Each file and source code fragment is *decoded* into a stream of Unicode characters. The command line options may specify a *codepage* for this process.

Note: the C# specification gives a full description of decoding.

- **Tokenization.** Each stream of Unicode characters is *tokenized* into a token stream through the lexical analysis described in §3.
- **Lexical Filtering.** Each token stream is *filtered* by being processed through a state machine implementing the rules described in §15. When the lightweight syntax option is enabled, the rules described in that chapter insert artificial extra tokens into the token stream and replace some existing tokens with others. Some token replacements are also made regardless of the use of the use of the lightweight syntax option.
- **Parsing.** The augmented token stream is *parsed* according to the grammar specification in this document. Ambiguities in the grammar rules are resolved according to the precedence rules described in §6.1.1.
- **Importing.** The specified 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 (§13.1) and types. These namespace declaration groups are then combined to form an initial name resolution environment (§14.1).
- **Checking.** The results of parsing each token stream are *checked* one by one. Checking includes invoking the procedures such as *Name Resolution* (§14.1), *Type Inference* (§14.6), *Constraint Solving* (§14.6), *Generalization* (§14.7) and *Application Resolution* (§14.2), as well as other specific rules described in this specification.

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

- **Elaboration.** One result of checking is an *elaborated program fragment* that contains elaborated declarations, expressions and types. For most constructs (e.g. constants, control flow and data expressions) the elaborated form is simple. Elaborated forms are used for evaluation, CLI reflection and the F# expression trees returned by *expression quotation* (§6.9).
- **Execution.** Elaborated program fragments that are successfully checked are added to a collection of available program fragments. Each has a *static initializer*. Static initializers are executed as described in (§13.1.1)

3 Lexical Analysis

Lexical analysis converts an input stream of Unicode characters into a stream of tokens. This is done by repeatedly processing the input character stream using a longest-match interpretation of a collection of regular expressions specifying different possible tokens. Some tokens such as *block-comment-start* are discarded when processed as described below.

3.1 Whitespace

Whitespace is made up of spaces, tabs and newline characters:

```
regexp whitespace = [ ' ' '\t' ]+
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 `//` and extend to the end of a line.

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

When a *block-comment-start* token is matched, the subsequent text is tokenized recursively using the tokenizations defined in this section until a *block-comment-end* token is matched. The intermediate tokens are discarded. For example, comments may be nested, and strings embedded within comments are tokenized by the rules for *string* and *verbatim-string*.

> In particular strings embedded in comments are tokenized without looking for closing `*)` marks. This makes `(* Here's a code snippet: let s = "*)" *)` a valid comment.

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* beginning with `///` and are kept and associated with subsequent declaration elements.

3.3 Conditional Compilation

`#if` *ident*`/#else/#endif` are pre-processing directives that delimit conditional compilation sections. The directives delimiting such a section are recognized by the following regular expressions:

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

A pre-processing directive always occupies a separate line of source code and always begins with a # character and a pre-processing directive name. White space can occur before the # character.A source line containing a #if, #else, or #endif directive can end with whitespace and a single-line comment. Multi-line comments are not permitted on source lines containing pre-processing directives.

If an *if-directive* token is matched during tokenization, text is recursively tokenized until a corresponding *else-directive* or *endif-directive*. If the given *ident* is defined in the compilation environment (e.g. via the command line option –define), these tokens are included in the token stream. Otherwise the tokens are discarded. The converse applies to the text between any corresponding *else-directive* and the *endif-directive*.

Text that is skipped is tokenized using the following token specifications:

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

In particular

- #if *ident*/#else/#endif sections may be nested
- Strings and comments are not treated as special in skipped text.

> Note: This corresponds to the current implementation but is somewhat inconsistent: #endif markers within strings will not be skipped, unlike (* markers in strings embedded in comments.

For the purposes of this specification, the direct tokens are discarded from the returned lexical stream.

3.3.1 Conditional Compilation for OCaml Compatibility

F# allows code to be cross-compiled as both F# and OCaml code. Sections marked

```
token start-fsharp-token = "(*IF-FSHARP" | "(*F#"
token end-fsharp-token = "ENDIF-FSHARP*)" | "F#*)"
token start-ocaml-token = "(*IF-OCAML*)"
token end-ocaml-token = "(*ENDIF-OCAML*)"
```

When a *start-fsharp-token* or *end-fsharp-token* token is encountered the token is ignored. This means 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 as part of the token stream as normal. When a *start-ocaml-token* token is encountered, the token is discarded and the following text is tokenized as *string*, _ (i.e. any character) and *end-ocaml-token* until an *end-ocaml-token* is reached. Comments are not treated as special during this process and are simply processed as “other text”. This means text surrounded by

```
   (*IF-CAML*)  ... (*ENDIF-CAML*)
or (*IF-OCAML*)  ... (*ENDIF-OCAML*)
```

is excluded when compiling with the F# compiler. The intervening text is tokenized as “OCaml strings and other text” and the tokens discarded until the corresponding end token is reached. Note that comments are not treated as special during this process and are simply processed as “other text”.

Note the converse holds when compiling programs using an OCaml compiler.

3.4 Identifiers and Keywords

Identifiers follow the specification below. Any sequence of characters enclosed in `` `` double-tick marks, excluding newlines and TABs and double-tick pairs themselves, is treated as an identifier.

```
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                          e.g. myName1
  | `` [^\n\r\t`]+ | `[^\n\r\t`] ``    e.g. ``type.with unusual#name``
```

Unicode characters include those within the standard ranges. All input files are currently assumed to be UTF-8-encoded. See the C# specification for the definition of Unicode characters accepted for the above classes of characters.

Some identifiers are interpreted as keywords, and some symbolic identifiers are permitted as keywords. The identifiers treated as keywords of the F# language are shown below.

```
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 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 also keywords primarily because they are keywords in OCaml. The `--ml-compatibility` option permits OCaml keywords reserved for future use by F# to be used 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 to those used in OCaml.

```
asr      >>> (on signed type)
land     &&&
lor      |||
lsl      <<<
lsr      >>> (on unsigned type)
lxor     ^^^
mod      %
sig      begin (i.e. begin/end may be used instead of sig/end)
```

The following identifiers are reserved for future use by F#.

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

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

With the exception of the symbolic keywords such as `let!` listed later in this specification, the following token forms are reserved:

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

In the remainder of this specification we refer to the tokens generated for keywords simply by using the text of the keyword itself.

With the exception of identifiers mentioned elsewhere in this specification, identifiers beginning with two underscores are reserved for identifiers generated by the F# compiler and may not be used in user code:

```
token reserved-keyword-formats =
    | __ident-text
```

3.5 Strings and characters

String-like literals may be specified for two types: Unicode strings (type `string` = `System.String`) and unsigned byte arrays (type `byte[]` = `bytearray`). Literals may also be specified using C#-like verbatim forms that interpret `\` as a literal character rather than an escape sequence. Unicode characters in UTF-8-encoded files may be directly embedded in strings, as for identifiers (see above), as may trigraph-like specifications of Unicode characters in an identical manner to C#.

```
regexp escape-char =  '\' ["\'ntbr]
regexp non-escape-chars =  '\' [^"\'ntbr]
regexp simple-char-char =
      | (any char except newline,return,tab,backspace,',\,")


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

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,',\,"
```

Strings tokens translate to string values by concatenating all the Unicode characters for the *string-char* elements within the string. Strings may include newline characters which are embedded as \n characters. However, if a line ends with \, the newline character and any leading whitespace elements on the subsequent line are ignored. Thus

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

gives s value "abcdef", while

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

gives s value "abc\010   def" where \010 is the embedded control character for \n with ASCII value 10.

Verbatim strings may be specified using the @ symbol prior to the string. For example

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

gives `s` value `"abc\def"`.

String-like and character-like literals may also be specified for unsigned byte arrays (type `byte[]`). No Unicode characters with surrogate-pair UTF16 encodings or UTF16 encodings greater than 127 may be used in these tokens.

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 by F#.

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

Note: In earlier versions of F# the Unicode characters « and » were used as quotation operators, e.g.

```
let query = « 1+1 »
```

These are now removed, and the ASCII equivalent of these is <@ and @>, e.g.

```
let query = <@ %db.Customers @>
```

3.7 Symbolic Operators

Symbolic operators are sequences of characters as shown below, except where a combination of characters is used as a symbolic keyword.

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

token symbolic-op  =
    | first-op-char op-char*
```

For example, `&&&` and `|||` are valid symbolic operators.

The associativity and precedence of symbolic operators when used in expression forms is given later in this specification.

The following operators are used in expression quotation (§6.9).

```
token quote-op  =
  |  <@ @> <@@ @@>
```

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+                          -- e.g., 34

regexp xint =
       | int                             -- e.g., 34
       | 0 (x|X) hexdigit+               -- e.g., 0x22
       | 0 (o|O) octaldigit+             -- e.g., 0o42
       | 0 (b|B) bitdigit+               -- e.g., 0b10010

token sbyte      = xint 'y'              -- e.g., 34y
token byte       = xint 'uy'             -- e.g., 34uy
token int16      = xint 's'              -- e.g., 34s
token uint16     = xint 'us'             -- e.g., 34us
token int32      = xint 'l'              -- e.g., 34l
token uint32     = xint 'ul'             -- e.g., 34ul
                 | xint 'u'              -- e.g., 34u
token nativeint  = xint 'n'              -- e.g., 34n
token unativeint = xint 'un'             -- e.g., 34un
token int64      = xint 'L'              -- e.g., 34L
token uint64     = xint 'UL'             -- e.g., 34UL
                 | xint 'uL'             -- e.g., 34uL

token ieee32     =
       | float ['F'|'f']                 -- e.g., 3.0F or 3.0f
       | xint 'lf'                       -- e.g., 0x00000000lf
token ieee64     =
       | float                           -- e.g., 3.0
       | xint 'LF'                       -- e.g., 0x0000000000000000LF

token bignum  = int ('I' | 'N' | 'Z' | 'Q' | 'R' | 'G')
                                         -- e.g., 347426262631938326125361 71N

token decimal = (float|int) ['M'|'m']

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

3.8.1 Post-filtering of adjacent, prefix "-" tokens

Negative integers are specified using the appropriate integer negation operator, e.g., `-3`. The token steam is post-filtered by the following additional rules:

(a) If the token stream contains

```
- token
```

where these tokens are adjacent then

i. If *token* is a constant signed literal the pair of tokens is merged, e.g. -3 becomes a single token "-3"

ii. Otherwise the tokens remain, however the "-" token is marked as a `ADJACENT_PREFIX_MINUS` token.

This rule is not applied for the sequence:

```
token1 - token2
```

where all three tokens are adjacent and *token1* is a terminating token from expression forms with lower precedence than the grammar production

```
expr = MINUS expr
```

For example it is not applied for the token sequence

```
ident  -  ident
```

when all three tokens are adjacent.

(b) Otherwise the usual grammar rules apply with an addition for `ADJACENT_PREFIX_MINUS`

```
expr =  expr MINUS expr
     |  MINUS expr
     |  ADJACENT_PREFIX_MINUS 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* "..".

Note: This allows ".." to immediately follow an integer. This 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+
```

3.9 Pre-processor Declarations

3.9.1 Line Directives

Source code file names and line numbers are reported in error messages, recorded in debugging symbols and propagated to quoted expressions. F# code generated by other tools can record original filenames and line numbers using the following directives:

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

A line number directive applies to the line immediately following the directive. The first line number of an F# file is by default 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. The augmented grammar rules taking these into account are also shown there.

3.11 Identifier Replacements

The following identifiers are automatically replaced by values in F# code:

`__SOURCE_DIRECTORY__`	replaced by a literal verbatim string, e.g. `C:\source`
`__SOURCE_FILE__`	replaced by a literal verbatim string of the filename as given to the command line compiler or on a load line, e.g. `source\file.fs`, after taking into account adjustments from line directives.
`__LINE__`	replaced by a literal string giving the line number in the source file, after taking into account adjustments from line directives.

4 Basic Grammar Elements

In this section we define some 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 | _ |
```

When defining operators the symbolic operator name is placed in parentheses. For example:

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

In this example `(+++)` effectively acts as an identifier with associated text `+++`. Likewise, when defining an active pattern (§7) the element being defined is named using a special structured name, e.g.

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

Likewise these forms of "names" can be used when using values as first-class values, e.g.

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

The three character token `(*)` is used to define the `*` operator:

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

For other operators beginning with `*`, a space must be added to avoid `(*` being interpreted as the start of a comment:

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

Some operators may be used as both prefix and infix operators, e.g. `-` and `+`. When used as a prefix operator the implicit operator name has `~` prepended. For example, `-x` is parsed as an application of the operator `~-` to the expression `x`. This name is also used when giving definitions for prefix operators:

This means that these prefix operators are defined with this character added:

```
// For a complete redefinition of the operator:
let (~+) x = x

// For defining 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)
```

Symbolic operators and some symbolic keywords are given mangled names that are also visible to the programmer. The mangled names are shown below and correspond to the standard definitions for the operators in the F# library.

```
[]     op_Nil
::     op_Cons
+      op_Addition
-      op_Subtraction
*      op_Multiply
/      op_Division
@      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_LessThanOrEqual
>=     op_GreaterThanOrEqual
<      op_LessThan
>      op_GreaterThan
|>     op_PipeRight
<|     op_PipeLeft
!      op_Dereference
>>     op_ComposeRight
<<     op_ComposeLeft
<@ @>  op_Quotation
<@@ @@> op_QuotationUntyped
+=     op_AdditionAssignment
-=     op_SubtractionAssignment
*=     op_MultiplyAssignment
/=     op_DivisionAssignment
..     op_Range
.. ..  op_RangeStep
```

Other symbolic identifiers are given names by mangling to `op_`$N_1 \ldots N_n$ where N_1 to N_n are given by mangling each character shown in the table below. For example the symbolic identifier `<*` mangles to the name `op_LessMultiply`:

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

4.2 Long Identifiers

Long identifiers are simply sequences of identifiers separated by '.' and optional whitespace. Some long identifiers may terminate with operator names.

```
long-ident :=  ident '.' ... '.' ident
long-ident-or-op :=  [long-ident '.'] ident-or-op
```

4.3 Constants

Constants are used in patterns and expressions. See the lexical section above for descriptions of the valid constant formats.

```
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" (i.e. System.String)
      | verbatim-string        -- String of type "string" (i.e. 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 Precedence and Operators

The precedence of ambiguous expression constructs is as follows, from lowest (least tightly binding) to highest (most tightly binding). The marker OP indicates the class of *symbolic-op* tokens beginning with the given prefix.

```
Infix operators, expressions and precedence order for ambiguity resolution

    as                          %right
    when                        %right
    |                           %left
    ;                           %right
    let                         %nonassoc
    function, fun, match, try   %nonassoc
    if                          %nonassoc
    ->                          %right
    :=                          %right
    ,                           %nonassoc
    or ||                       %left
    & &&                        %left
    !=OP <OP >OP $OP = |OP &OP  %left
    ^OP                         %right
    ::                          %right
    :?> :?                      %nonassoc
    -OP +OP                     %left
    *OP /OP %OP                 %left
    **OP                        %right
    "f x" "lazy x" "assert x"   %left
    "| rule"                    %right  -- pattern match rules
    !OP ?OP ~OP -OP +OP         %left
    .                           %left
    f(x)                        %left  - high precedence application
    f<types>                    %left  - type application
```

Leading `.` and `$` characters are ignored when determining precedence, so, for example, `.*` and `$*` have the same precedence as `*`.

```
        $ .
```

Operators such as `.$` and `$$` have precedence given by the `$OP` class of operators.

Note: This ensures operators such as `.*` (used for pointwise-operation on matrices) and `$*` (used for scalar-by-matrix multiplication) have the expected precedence.

High precedence application and type application arise from the augmentation of the lexical token stream and are covered toward the end of this document.

The following *symbolic-op* tokens can be used to form expressions:

```
infix-op :=
    or || & && <OP >OP $OP = |OP &OP ^OP :: -OP +OP *OP /OP %OP
    **OP

infix-or=prefix-op :=
    -OP +OP % %% & &&

prefix-op :=
    !OP ?OP ~OP
```

Operators beginning with `?` and `!` are exclusively prefix, with the exception of operators beginning with `!=`.

The operator families `-OP`, `+OP`, `&`, `&&`, `%`, `%%` can be used in both infix and prefix positions.

For the most part, the precedence specification follows the same rules as the OCaml language. One significant exception is that the expression "`!x.y`" parses as "`!(x.y)`" rather than "`(!x).y`". This is because the OCaml grammar uses uppercase/lowercase distinctions to make disambiguation at parse time possible as shown in the following examples:

```
OCaml: !A.b.C.d  == (!(A.b)).(C.d)

OCaml: !a.b.c.d  == (((!a).b).c).d

F#:    !A.b.C.d  == !(A.b.C.d)

F#:    !a.b.c.d  == !(a.b.c.d)
```

Note that in the first example '!' binds two elements of a long-identifier chain, and in the second it only binds one. Thus the parsing depends on the fact that 'A' is upper case and OCaml uses this fact to know that it represents a module name. F# deliberately allows values and module names to be both upper and lower case, and so F# cannot resolve the status of identifiers (i.e. whether an identifier is a module, value, constructor etc.) at parse-time, and instead does this when parsing long identifiers chains during type checking (just as C# does). The above alteration means that parsing continues to remain independent on identifier status.

5 Types and Type Constraints

The notion of *type* is central to the static checking of F# programs, and also appears at runtime through dynamic type tests and reflection. The word is used with five distinct but related purposes:

- **Type definitions** such as the actual CLI or F# definitions of `System.String` or `Microsoft.FSharp.Core.Option<_>`.
- **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**, arise during type checking and inference, either by the translation of syntactic types appearing in the source text, or via constraints related to particular language constructs. For example, `option<int>` is the fully processed static type inferred for an expression `Some(1+1)`. Static types may contain *type variables* (see below).
- **Runtime types**, which are objects of type `System.Type` and are runtime representations of some or all of the information carried in type definitions and static types. Runtime types associated with objects are accessed via the `obj.GetType()` available on all F# values. An object's runtime type is related to the static type of the identifiers and expressions that correspond to the object. Runtime type may be a refinement of the reified representation of the static type. 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, i.e. the runtime types of all objects are *ground*. Runtime types reported by `System.Reflection` may contain type variables, represented by `System.Type` values.

The syntactic forms of types as they appear in programs are as follows:

```
type :=
  | ( type )
  | type -> type                        -- function type
  | type * ... * type                   -- tuple type
  | typar                               -- variable type
  | long-ident                          -- named type, e.g., int
  | long-ident <types>                  -- named type, e.g., list<int>
  | long-ident < >                      -- named type, e.g., IEnumerable< >
  | type long-ident                     -- named type, e.g. int list
  | type[,…,]                           -- array type
  | type lazy                           -- lazy type
  | type when constraints               -- type with constraints
  | typar :> type                       -- variable type with subtype constraint
  | #type                               -- anonymous type with subtype constraint

types :=  type, ..., type

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

constraint :=
  | typar :> type                       -- coercion constraint
  | typar : null                        -- nullness constraint
  | typar-choice : (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 : delegate<type, type>        -- delegate decomposition constraint

typar-defn := attributes_opt typar

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

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

typar-choice  :=
  | typar
  | (typar or ... or typar)

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

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 implicit generic type 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 occurs in source code and the static type 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), and a *floating type variable environment* (§14.1) and a *type inference environment* (see §14.6).

The phrase “fresh type” means a static type formed from a *fresh type inference variable* (§14.6). Type inference variables are either solved or generalized by *type inference* (see §14.6). 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* (see §14.6).

5.1.1 Named Types

A type of the form *long-ident*<ty_1,…,ty_n> is a *named type* with one or more suffixed type arguments. Named types are converted to static types as follows:

- *Name Resolution for Types* (§14.1) must resolve *long-ident* to a type definition with formal type parameters <$typar_1$,…, $typar_n$> and formal constraints *C*. The number of type arguments *n* is used as part of the name resolution process to distinguish between named types taking 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 with respect to these new variables, and constraints $ty_i = ty'_i$ added to the current inference constraints.

A type of the form *long-ident* is a named type with no type arguments.

A type of the form *type long-ident* is a named type with one type argument. It is processed as if it were written *long-ident*<*type*>.

A type of the form *ty* `lazy` is shorthand for the named type `Microsoft.FSharp.Control.Lazy`<*ty*>.

A type of the form ty_1 `->` ty_2 is a *function type*, where ty_1 is the domain of the function values associated with the type, and ty_2 is the range. In compiled code it is represented via the named type `Microsoft.FSharp.Core.FastFunc`<ty_1,ty_2>.

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 type variable name.

- If name resolution succeeds, the result is a variable type referring to an existing declared type parameter.
- If name resolution fails, the current *floating type variable environment* is consulted, though only if processing a syntactic type embedded in an expression or pattern. If the type variable name is given a type in that environment then that mapping is used. Otherwise a fresh type inference variable is created (see §14.6).

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 §14.6).

A type of the form `^ident` is a *statically resolved variable type*. A fresh type inference variable is created and added to the type inference environment (see §14.6). This type variable is tagged with an attribute indicating it may not be generalized except at `inline` definitions (see §14.7), and likewise any type variable with which it is equated via a type inference equation may similarly not be generalized.

Note: In this specification we use upper case identifiers such as 'T or 'Key for user-declared type parameters, and lower-case identifiers such as 'a or 'b for compiler-inferred type parameters.

5.1.3 Tuple Types

A type of the form ty_1 `*` ... `*` ty_n is a *tuple type*. A tuple type is shorthand for a use of the family of F# library types `Microsoft.FSharp.Core.Tuple<_,...,_>`. See §6.4.2 for the details of this encoding.

5.1.4 Array Types

A type of the form *ty*`[]` is a *single dimensional array type*.

A type of the form *ty*`[,…,]` is a *multi-dimensional array type*.

Except where specified otherwise in this document, these are treated as named types, e.g. 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 in[,][] in C#, i.e. the dimensions are swapped. This is for consistency with other postfix type names in F# such as "int list list".

Note: F# only supports multi-dimensional array types up to rank 4.

5.1.5 Constrained Types

A type of the form *type* `when` *constraints* is a *type with 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 the sub-sections that follow.

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.

5.1.5.1 Subtype Constraints

A constraint of the form *typar* `:>` *type* is an *explicit subtype constraint*. 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. The conditions governing when two types satisfy a subtype constraint are specified in §5.3.7.

Note: Subtype constraints also arise implicitly from expressions of the form *expr* `:>` *type* , patterns of the form *pattern* `:>` *type* , from uses of generic values, types and members with constraints, and from the implicit use of subsumption when calling members (§14.5).

5.1.5.2 Nullness Constraints

A constraint of the form *typar*`: null` is an *explicit nullness constraint*. During checking, *typar* is first checked as a variable type and the constraint is added to the current inference constraints. The conditions governing when a type satisfies a nullness constraint are specified in §5.3.8.

In addition:

- The *typar* must be a statically resolved type variable (REF) of the form `^`*ident*. This ensures that the constraint is resolved at compile time, and means that generic code may not use this constraint unless that code is inlined (§14.7).

Note: Nullness constraints are primarily for use during F# type checking and are used relatively rarely in F# code.

Note: Nullness constraints also arise from expressions of the form `null`.

5.1.5.3 Member Constraints

A constraint of the form `(typar or ... or typar) : (member-sig)` is an *explicit member constraint*. For example, the operator `+` is defined in the F# library with the following signature:

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

This indicates that for each use of the function the types of `^a`, `^b` and `^c` must be inferred to be named types during checking and that the named type for either `^a` or `^b` must support a static member called `+` with the given signature. The operator may be used on two values where the first supports a static member operator `+` (in C# written `static operator +(...)`).

In addition:

- Each of *typar* must be a statically resolved type variable §5.1.2 of the form `^ident`. This ensures that the constraint is resolved at compile time against a corresponding named type and means that generic code may not use this constraint unless that code is inlined (§14.7).
- The *member-sig* may not be generic, i.e. may not include explicit type parameter definitions.
- The conditions governing when a type satisfies a member constraint are specified in §5.1.5.3.

Note: Member constraints are primarily used for defining overloaded functions use in the F# library and are used relatively rarely in F# code.

Note: Member constraints also arise implicitly when using operators `+`, `-` etc., and from other values built in terms of these values or other values with signatures annotated with member constraints.

Note: Uses of overloaded operators do not give rise to generalized code unless definitions are marked as `inline`. For example, the function

```
let f x = x + x
```

gives rise to a function `f` that can only be used to add one type of value, e.g. `int` or `float` – the exact type will be determined by later constraints.

5.1.5.4 Default Constructor Constraints

A constraint of the form *typar* `: (new : unit -> 'T)` is an *explicit default constructor constraint*.

During constraint solving (§14.6), the constraint *type* `: (new : unit -> 'T)` is met if *type* has a parameterless object constructor.

Note: This constraint form is primarily to give completeness w.r.t. the full set of constraints permitted by CLI implementations. It is rarely used in F# programming.

5.1.5.5 Struct Constraints

A constraint of the form *typar* `: struct` is an *explicit struct constraint*.

During constraint solving (§14.6), the constraint *type* `: struct` is met if *type* is a value type, excluding the CLI type `System.Nullable<_>`.

Note: This constraint form is primarily to give completeness w.r.t. the full set of constraints permitted by the CLI implementations. It is rarely used in F# programming.

Note: 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<_>` tends to play the same role as `System.Nullable<_>`. For various technical reasons the two types cannot be equated, notably because `System.Nullable<System.Nullable<_>>` is not a valid type in CLI languages.

5.1.5.6 *Reference Type Constraints*

A constraint of the form *`typar : not struct`* is an *explicit reference type constraint*.

During constraint solving (§14.6), the constraint *`type : not struct`* is met if *`type`* is a reference type.

Note: This constraint form is primarily to give completeness w.r.t. the full set of constraints permitted by CLI implementations. It is rarely used in F# programming.

5.1.5.7 *Enumeration Constraints*

A constraint of the form *`typar : enum<underlying-type>`* is an *explicit enumeration constraint*.

During constraint solving (§14.6), the constraint *`type : enum<underlying-type>`* is met if `type` is a CLI-compatible enumeration type with constant literal values of type *`underlying-type`*.

Note: This constraint form primarily exists to allow the definition of library functions such as enum. It is rarely used directly in F# programming.

Note: The `enum` constraint doesn't imply anything about subtyping, e.g. an 'enum' constraint doesn't imply that the type is a subtype of System.Enum

5.1.5.8 *Delegate Constraints*

A constraint of the form *`typar : delegate<tupled-args-type, return-type>`* is an *explicit delegate constraint*.

During constraint solving (§14.6), the constraint *`type : delegate<tupled-arg-type, return-types>`* is met if *`type`* is some delegate type D with declaration *`type D = delegate of object * arg1 * ... * argN`* where *`tupled-arg-type = arg1 * ... * argN`*. That is, the delegate must match the standard CLI design pattern where the '`sender`' is given as the first argument to the event.

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

Note: The `delegate` constraint doesn't imply anything about subtyping, e.g. a 'delegate' constraint doesn't imply that the type is a subtype of System.Delegate.

Note: The `delegate` constraint only applies to delegate types that follow the "standard" form for CLI event handlers, where the first argument is a "sender" object. This is because the purpose of the constraint is to simplify the presentation of CLI event handlers to the F# programmer.

5.2 Type Parameter Definitions

Type parameter definitions (or *explicit type parameter binding sites*) can occur at value definitions in modules, member definitions, type definitions, record field definitions and corresponding specifications in signatures.

For example:

```
let id<'T> (x:'T) = x
val id<'T> : 'T -> 'T
type Funcs<'T1,'T2> =
    { Forward: 'T1 -> 'T2;
      Backward : 'T2 -> 'T2 }
```

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()
```

However note that the same declaration can be written using the following more convenient syntactic form

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

Explicit type parameter definitions can declare custom attributes on type parameter definitions (§11.2).

5.3 Logical Properties of Types

As described above, during type checking and elaboration, syntactic types and constraints are processed into a reduced form made up of:

- named types `ident<types>`, where each `ident` consists a function type, a specific shape of array type, a specific *n*-tuple type or a specific type definition; or
- type variables `'ident`

5.3.1 Characteristics of Type Definitions

Named types refer to type definitions, e.g. CLI type definitions such as `System.String` and types defined in F# code (§8). The following terms are used to categorize type definitions:

- Type definitions may be *generic*, e.g. `System.Collections.Generic.Dictionary<_,_>`.
- The generic parameters of type definitions may have associated *formal type constraints*.
- Type definitions may have *custom attributes* (§11.2), some of which are relevant to checking and inference.
- Type definitions in F# code may be *type abbreviations* (§8.2). These are eliminated for the purposes of checking and inference (see §5.3.2).
- Type definitions have a *kind* which is one of *class*, *interface*, *delegate*, *struct*, *record*, *union, enum, measure* or *abstract*. For F# types the kind is determined at the point of declaration through a simple procedure called *Type Kind Inference* (see §8.1) if it is not given explicitly as part of the type definition.
- Type definitions may be *sealed.* Record, union, function, tuple, struct, delegate, enum and array types are all sealed, as are class types marked with the `SealedAttribute` attribute.
- Type definitions may have zero or one *base type declarations*. These represent an additional, encapsulated type supported by any values 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 supported by values formed using the type.

The *kind* of 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. Likewise we refer to *interface*, *delegate*, *struct*, *record*, *union*, *abstract, tuple* and *variable* types.

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

5.3.2 Expanding Abbreviations and Inference Equations

At each point they are discussed in this specification, static types are assumed to be treated as equivalent modulo any type equations in the current inference constraints and through the elimination of type abbreviations (see §8.2). For example, static types may refer to type abbreviations (see §8.2) such as `int`, an abbreviation for `System.Int32`, given by the declaration

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

in the F# library. Likewise consider the process of checking the function

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

As we shall see in §6, during checking fresh type inference variables are created for values `x` and `y`, say ty_1 and ty_2. The process of checking imposes constraints ty_1 `= string` and ty_1 `=` ty_2. The second constraint arises from the use of the generic operator `(=)`. The process of constraint solving means that the ty_2 `= string` is "known", and thus the type of `y` is "known". This in turn allows name resolution of `y` to proceed.

All relations on static types are considered modulo these equational inference constraints and 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 back to users. This typically means that type abbreviations should be preserved in the logical structure of types throughout the checking process.

5.3.3 Type Variables and Binding

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 *binding site*. For example, given

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

the binding site of `'T` is the type definition of `C`.

Type variables without a binding site are *inference variables*. If an expression is made up of multiple sub-expressions, then the resulting constraint set is normally the union of the constraints arising. However, for some constructs (notably let and member bindings), *generalization* is applied (see §14.7). This means that some intermediate inference variables and constraints are factored out of the intermediate constraint sets and new implicit binding site(s) are assigned for these variables.

For example, given

```
let id x = x
```

the type inference variable associated with the value `x` is generalized and has an implicit binding site at the definition of function `id`. Occasionally it is useful to give a more fully annotated representations of the inferred and generalized type information, e.g.,

```
let id<'a> x:'a = x:'a
```

5.3.4 Base Type of a Type

The *base type* of a static type is as follows:

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

The specification of these types can be found in the CLI specifications and corresponding implementation documentation.

5.3.5 Interfaces Types of a Type

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

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

5.3.6 Type Equivalence

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

- 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 binding site or are the same type inference variable.

This means 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 through the addition of further constraints to the current inference constraints. Thus `list<int>` and `list<'a>` are feasibly equivalent for the empty constraint set.

5.3.7 Subtyping and Coercion

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

Variable types `'T` coerce to all types ty where a constraint of the form `'T :>` ty_2 exists in the current inference constraints, 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*

5.3.8 Nullness

One of the aims of the design of F# is to greatly reduce the use of `null` literals in common programming tasks, since they are generally result in highly error-prone code. However,

- the use of some `null` literals is needed in order to interoperate with CLI libraries
- the appearance of `null` values during execution can't be completely precluded for technical reasons related to the CLI and CLI libraries.

As a result, different F# types vary in their treatment of the `null` literal and `null` values. For these purposes, all ground types and type definitions can be categorized into the following four kinds:

- **Types with the `null` literal.** These have `null` as an "extra" value. Types in this category are:
    - All CLI reference types defined in other CLI languages

  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 are types that *don't admit the `null` literal* but do have *`null` as an abnormal value*. Types in this category are:
    - All F# list, record, tuple, function, class and interface types
    - All F# union types apart from those with `null` as a normal value (see below)

  For these types, the use of the `null` literal is not directly permitted. However it is, strictly speaking, possible to generate a `null` value for these types using "backdoor" techniques such as `Unchecked.defaultof<`*`type`*`>`. For these types `null` is considered an abnormal value. The behaviour of all operations on with respect to `null` values is defined in §6.10.

- **Types with `null` as a representation value.** These are types that *don't admit 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. Types in this category are:
    - The `unit` type. The `null` value is used to represent all values of this type.
    - Any discriminated union type with the `Microsoft.FSharp.Core.CompilationRepresentation(CompilationRepresentationFlags.UseNullAsTrueValue)` attribute flag and a single nullary discriminated union case. The `null` value is used to represent this case. In particular, `null` is used as a representation for `None` in the F# `option<_>` type.

- **Types without `null`.** These are types that *don't admit the `null` literal* and *don't have the `null` value*. This covers all value types, e.g. primitive integers, floating point numbers, and any value of a CLI or F# struct type.

Given this, a static type ty *satisfies a nullness constraint* ty `: null` if it:

- Has an outermost named type with the `null` literal
- A variable type with a *typar* `: null` constraint

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

- "Limited" default initialization may be used when types are known to have a valid and "safe" default zero value. For example, the types of fields labelled with `DefaultValueAttribute(true)` are checked to ensure they admit default initialization.
- "Arbitrary" default initialization may be encountered through the use of CLI libraries and through a very specific, limited set of F# library primitives, in particular `Unchecked.defaultof<_>` and `Array.zeroCreate`.

In particular, a type *admits default initialization* if it is either

- A type satisfying the nullness constraint
- A primitive value type
- A struct type whose field types all admit default initialization

5.3.9 Dynamic Conversion Between Types

One ground type *vty* *dynamically converts to* another type *ty* if:

- *ty* coerces to *vty*.
- *vty* is `int32[]`and *ty* is `uint32[]`(or vice-versa). Likewise for `sbyte[]`/`byte[]`, `int16[]`/`uint16[]`, `int64[]`/`uint64[]`, `nativeint[]`/`unativeint[]`.
- *vty* is *enum*`[]` where *enum* has underlying type *underlying*, and *ty* is *underlying*`[]` (or vice-versa), or the (un)signed equivalent of *underlying*`[]` by the rule above.
- *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, this specification covers the additional rules of CLI dynamic conversions, all of which hold for F# types, e.g.

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

Above, 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
```

Above, 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
```

Above, 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
```

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

5.4 Static Type Schemes

During type checking and generalization (§14.7), generic values and methods are given *type schemes*, a technique used in this specification to note the inferred binding sites for generic type variables. Type schemes are written `<typars when constraints> : type`, e.g.

```
<'T> : int -> 'T
<'T, 'U when 'U :> System.IDisposable> : 'T -> ('T -> 'U) -> 'U
```

Two type schemes are *definitely equivalent* if their type variables, type constraints and types are equivalent after renaming variables, e.g. `<'T> : int -> 'T` is equivalent to `<'U> : int -> 'U`. The order and number of type variable declarations is significant, so `<'U,'T> : 'T -> 'U` is not equivalent to `<'T,'U> : 'T -> 'U`.

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-range]                    -- slice expression (1D)
  | expr.[slice-range, slice-range]       -- slice expression (2D)
  | expr <- expr                          -- assignment expression

  | expr , ... , expr                     -- tuple expression
  | new type expr                         -- simple object expression
  | { new base-construction               -- object expression
      with val-or-member-defns end
    interface-impls }
  | { field-binds }                       -- record expression
  | { expr with field-binds }             -- 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 bindings in expr                  -- binding expression
  | let rec bindings in expr              -- recursive binding expression
  | use bindings in expr                  -- resource binding expression

  | fun pat ... pat -> 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 expr then expr]* [else expr] -- 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 @>                            -- expression quotation
  | <@@ expr @@>                          -- expression quotation
  | %expr                                 -- expression splice
```

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

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

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

binding :=
  | inline_opt accessibilty_opt ident typar-defns_opt pat ... pat return-type_opt = expr
                                                          -- function binding
  | mutable_opt access_opt pat typar-defns_opt return-type_opt = expr
                                                          -- value binding

return-type :=
  | : type

bindings := binding and ... and binding

field-bind :=
  | long-ident = expr                   -- field binding

field-binds := field-bind ; ... ; field-bind

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

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

interface-impl :=
  | interface type with val-or-member-defns end          -- interface implementation

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

val-or-member-defns :=
  | bindings                            -- members using binding syntax
  | member-defns                        -- members using member syntax

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
  | do!  expr in comp-expr          -- sequential computation
  | use! pat = expr in comp-expr    -- auto cleanup computation
  | yield! expr                     -- yield computation
  | yield expr                      -- yield result
  | return! expr                    -- return computation
  | return expr                     -- return result
  | expr                            -- control flow or imperative action

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-range :=
  | 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 Ambiguities

6.1.1 Ambiguities

Ambiguities in the grammar of expressions are resolved via reference to the precedence list in §4.4 and as indicated elsewhere in this specification.

6.2 Some checking and inference terminology

The rules applied when checking individual expression forms are described in the following subsections, with reference to procedures such as *Name Resolution* (§14.1) and *Constraint Solving* (§14.6).

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 dictates 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 simple "ty_1 = ty_2 is asserted" indicates that the constraint "ty_1 = ty_2" is added to the current inference constraints.
- The phrase "the 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:

- Adding constraints to the type inference constraint set may fail if an inconsistent constraint set is detected (§14.6), in which case either an error is reported or, if we are only attempting to *assert* the condition, then the state of the inference procedure is left unchanged and the test fails.

6.3 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 carries more explicit information than the source form, e.g. the elaborated form of `System.Console.WriteLine("Hello")` carries the information about exactly which overloaded method definition the call has resolved to. Elaborated forms are underlined in this specification, e.g. `let x = 1 in x+x`.

> Review: Elaborated forms are described informally at the moment. We need a formal grammar and notation for these.

With the exception of the extra resolution information noted above, elaborated forms are syntactically a subset of syntactic expressions, and in some cases (e.g. constants) the elaborated form is the same as the source form. The elaborated forms used in this specification are:

- Constants
- Resolved value references *`path`*
- Lambda expressions `(fun ident -> expr)`
- Primitive object expressions `{ new ty(args) with bindings interface-impls }`
- Data expressions (tuples, tagged data, array creation, record creation)
- Default initialization expressions (giving default values for types)
- Primitive `let ident = expr in expr` bindings of variables
- Primitive `let rec ident = expr and ... and ident = expr in expr` bindings of functions and tagged data.
- Applications of methods and functions (with static overloading resolved), including applications of dynamic dispatch through particular method dispatch slot.
- 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/catch `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 discriminated union cases
    - switches on the runtime types of objects

The following constructs are used in the elaborated forms of expressions that make direct local and array assignments 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 are used as the basis of evaluation (see §6.10) and also form the basis for the expression trees returned by *expression quotation* (see §6.9).

Convention: when describing the process of elaborating compound expressions we omit the process of recursively elaborating sub-expressions.

6.4 Data Expressions

6.4.1 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       // bignum          (Microsoft.FSharp.Math.BigInt 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            (Microsoft.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 N, Q, R, Z, I, G are used for user and library-defined types through 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 permits the use of the literal form `32Z` to generate a sequence of 32 'Z' characters. No literal syntax will be available for numbers outside the range of 32-bit integers.

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

6.4.2 Tuple Expressions

An expression of the form $expr_1, \ldots, expr_n$ 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 $(ty_1 * \ldots * ty_n)$ for fresh types $ty_1 \ldots ty_n$ and each individual expression e_i is checked using initial type ty_i.

Tuple types and expressions are encoded into uses of a family of F# library types named `System.Tuple`. For tuple types:

- For `n` <= 7 the reified form is `new Tuple<`$ty_1,\ldots,ty_N$`>`.
- For large `n` , tuple types are shorthand for types involving the additional F# library type `System.Tuple<_>` as follows:
- For `n` = 8 the reified form is `new Tuple<`$ty_1,\ldots,ty_7$`,Tuple<`ty_8`>>`.
- For 9 <= `n` the reified form is `new Tuple<`$ty_1,\ldots,ty_7,ty_8$`>` where ty_8 is the converted form of the type $(ty_8 *\ldots* ty_N)$.

Runtime types that are instantiations of the eight-tuple type `Tuple<_,_,_,_,_,_,_,_>` always have either `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 referenced by F# code an error may be reported by the F# compiler.

Tuple expressions elaborate to uses of `Microsoft.FSharp.Core.Tuple` as follows:

- For n <= 7 they elaborate to `new Tuple<`$ty_1,\ldots,ty_n$`>(`$expr_1,\ldots,expr_n$`)`.
- For n = 8 they elaborate to `new Tuple<`$ty_1,\ldots,ty_7$`,Tuple<`ty_8`>>(`$expr_1,\ldots,expr_7$`, new Tuple<`ty_8`>(`$expr_8$`)`.
- For 9 <= n they elaborate to `new Tuple<`$ty_1,\ldots ty_7,ty_{8n}$`>(`$expr_1,\ldots,$ `new` $ty_{8n}(e_{8n})$ where ty_{8n} is the type $(ty_8*\ldots* ty_n)$ and $expr_{8n}$ is the elaborated form of the expression $expr_8,\ldots, expr_n$.

Note that the encoded form of a tuple types is visible in the F# type system. For example, `(1,2)` statically has a type compatible with `Microsoft.FSharp.Core.Tuple<int,int>`. Likewise `(1,2,3,4,5,6,7,8,9)` has 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 given 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.

6.4.3 List Expressions

An expression of the form `[`$expr_1$`;...;` $expr_n$`]` is a *list expression* and has type `Microsoft.FSharp.Collections.List<`ty`>` for a fresh type ty. Each expression $expr_i$ is checked using ty as its initial type.

List expressions elaborate to uses of `Microsoft.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 discriminated union cases with symbolic names `::` and `[]` respectively.

6.4.4 Array Expressions

An expression of the form `[|`$expr_1$`;...;` $expr_n$ `|]` is an *array expression* and is of type ty`[]` for a fresh type ty. Each expression $expr_i$ is checked using ty as its initial type.

Array expressions are a primitive elaborated form.

Note: The Microsoft 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`.

6.4.5 Record Expressions

An expression of the form `{` $field\text{-}bind_1$ `;` … `;` $field\text{-}bind_n$ `}` 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 }
```

Below `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 via their containing type name:

```
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  }
```

Each *field-bind*$_i$ has the form *field-label*$_i$ `=` *expr*$_i$. Each *field-label*$_i$ is a *long-ident* which must resolve to a field F_i in a unique record type R as follows:

- If *field-label*$_i$ is a single identifier *fld* and the initial type is known to be a record type R`<_,...,_>` with field F_i with name *fld* then the field label resolves to F_i.
- If *field-label*$_i$ is not a single identifier or the initial type is a variable type, then the field label is resolved by performing *Field Label Resolution* (see §14.1) on *field-label*$_i$, giving a set of fields $FSet_i$. Each element of this set has a corresponding record type, giving $RSet_i$. The intersection of all $RSet_i$ must give a single record type R, and each field then resolves to the corresponding field in R.

The set of fields must be complete, i.e. precisely one field binding for each field in record type R. Each referenced field must be accessible (see §11.1), as must the type R.

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 given by

- Assume the type of the corresponding field F_i in R`<`ty_1`,...,`ty_N`>` is fty_i
- If the type of F_i prior to taking into account the instantiation `<`ty_1`,...,`ty_N`>` is a nominal, unsealed type then the initial type is a fresh type inference variable ty_i with a constraint ty_i `:>` in fty_i.
- Otherwise the initial type is fty_i.

Primitive record constructions are an elaborated form where the fields appear in the same order as given in the record type definition. Record expressions themselves elaborate to a form that may introduce `let` bindings to ensure that expressions are evaluated in the order the field bindings 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 (§8.5.3). For example:

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

6.4.6 Copy-and-update Record Expressions

An expression of the form

{ expr `with` *field-label*$_1$ `=` *expr*$_1$ `;` … `;` *field-label*$_n$ `=` *expr*$_n$ *}*

is a *copy-and-update record expression*. Each *field-label*$_i$ is a *long-ident*. For example, below `data2` is defined 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. Subsequently, the field bindings are resolved using the same technique as for record expressions. The field labels must each 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 given by

- Assume the type of the corresponding field F_i in $R<ty_1,...,ty_N>$ is fty_i
- If the type of F_i prior to taking into account the instantiation $<ty_1,...,ty_N>$ is a nominal, unsealed type then the initial type is a fresh type inference variable ty_i with a constraint ty_i :> in fty_i.
- Otherwise the initial type is fty_i.

Copy-and-update expressions elaborate as if they were a record expression written `let` *v* `=` *expr* `in { `*field-label*$_1$ `=` *expr*$_1$ `; … ;` *field-label*$_n$ `=` *expr*$_n$`;` F_1 `=` *v*`.`F_1`; ... ;` F_M `=` *v*`.`F_M `}` where F_1 `...` F_M are the fields of R not given bindings by *field-binds* and *v* is a fresh variable.

6.4.7 Function Expressions

An expression of the form `fun` pat_1 `...` pat_n `->` *expr* is a *function expression*, also known as a *lambda 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 involving only variable pattern arguments are a primitive elaborated form. Function expressions involving non-variable pattern arguments elaborate as if they had been written:

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

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

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

However if a third line is added then an exception is raised:

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

6.4.8 Object Expressions

An expression of the form

```
{ new tyâ‚€ [ args-expr ] [ as base-ident ] [ with
      val-or-member-defns end ]
  interface tyâ‚ with [
      val-or-member-defnsâ‚
  end ]
  …
  interface tyâ‚™ with [
      val-or-member-defnsâ‚™
  end ] }
```

is an *object expression*.

> Note: The use of `end` tokens is optional when lightweight syntax is used.

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() as base 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";  }

let obj5 =
   { new System.Object()
     interface System.IDisposable with
         member x.Dispose() = printfn "Dispose";  }
```

An object expression can specify additional interfaces beyond those required to fulfil the abstract slots of the type being implemented. For example `obj4` in the examples above 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. First, ty_0 to ty_n are checked and must all be 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 where ty_1 exists then the overall type is instead ty_1.

If ty_0 is a record type, then there must be no *base-ident*, no *interface-impls* and the *val-or-member-defns* must be of the form $id_1=expr_1$ `and ...and` $id_n=expr_n$. The expression is then checked as a record expression written `({` $id_1=expr_1$ `; ...;` $id_n=expr_n$ `} :` ty_0`)`.

Otherwise ty_0 must be a class or interface type. The base construction argument *args-expr* must be given if and only if ty_0 is a class type. The type must have one or more accessible constructors and the call to these constructors is resolved and elaborated using *Method Application Resolution* (see §14.4).

Apart from ty_0, each ty_i must be an interface type. For each member, an attempt is made to associate the member with a unique *dispatch slot* using *dispatch slot inference* (§14.8). 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.

The arguments patterns and expressions implementing the bodies of all implementing members are next checked one by one.

- 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* is given, then it must be named `base`, and in each member a base variable with this name is in scope. Base variables can only be used in the member implementations of an object expression and are subject to the same limitations as byref values described in §14.10.

An object expression may not implement abstract CLI events. Instead, you should override the `add_EventName` and `remove_EventName` methods that effectively implement the event.

The object must satisfy *dispatch slot checking* (§14.9) which ensures a 1:1 mapping exists between dispatch slots and their implementations.

Object expressions elaborate to a primitive form and execute by creating an object whose runtime type is compatible with all of the ty_i with a dispatch map that is the result of *dispatch slot checking*.

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 with get() = true }
```

6.4.9 Delayed Expressions

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

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

is syntactic sugar for

```
new Microsoft.FSharp.Control.Lazy (fun () -> expr)
```

The behaviour of the `Microsoft.FSharp.Control.Lazy` library type ensures expression *expr* is evaluated on demand in response to a `.Force` operation on the lazy value. The semantics of `.Force` are documented in the F# library documentation.

6.4.10 Computation Expressions

The following expression forms are all c*omputation expressions*.

```
expr { for pat in enum ... }
expr { let ... }
expr { let! ... },
expr { use ... }
expr { while ... }
expr { yield ... }
expr { yield! ... }
```

```
expr { return ... }

expr { return! ... }
```

More specifically, computation expressions are of the form *ident { comp-expr }* where *comp-expr* is, syntactically, the grammar of expressions with some additional constructs as specified below. Computation expressions are used for sequences and other non-standard interpretations of the F# expression syntax.

Note that the above grammar only lists the additional syntactic constructs that may be used in computation expressions. Some "normal" expression forms such as if/then/else and try/finally are given non-standard interpretations when used in computation expressions, as given by the translation below.

The expression

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

translates to

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

for a fresh variable `b`. The type of `b` must be a named type after the checking of *builder-expr*. If no method `Run` exists on the inferred type of `b` when this expression is checked then that call is omitted. Likewise if no method `Delay` exists on the type of `b` when this expression is checked then that call is omitted. This expression is then checked.

The translation {| _ |}C is defined recursively as follows:

```
{| let binds in cexpr |}C              = let binds in {| cexpr |}C)
{| let! pat = expr in cexpr |}C        = b.Bind(expr, (fun pat -> {| cexpr |}C))
{| do expr in cexpr |}C                = expr; {| cexpr |}C
{| do! expr in cexpr |}C               = b.Bind(expr, (fun () -> {| cexpr |}C))

{| yield expr |}C                      = b.Yield(expr)
{| yield! expr |}C                     = expr

{| return expr |}C                     = b.Return(expr)
{| return! expr |}C                    = expr

{| use pat = expr in cexpr |}C         = b.Using(expr, (fun pat -> {| cexpr |}C))
{| use! v = expr in cexpr |}C          = b.Bind(expr, (fun v ->
                                                b.Using(v,(fun v -> {| cexpr |}C)))

{| if expr then cexpr0 |}C             = if expr then {| cexpr0 |}C else b.Zero()
{| if expr then cexpr0 else cexpr1 |}C = if expr then {| cexpr0 |}C else {| cexpr1 |}C
{| match expr with pati -> cexpri |}C =  match expr with pati -> {| cexpri |}C

{| for pat in expr do cexpr |}C        = b.For({| expr |}E, (fun pat -> {| cexpr |}C))

{| while expr do cexpr |}C             = b.While((fun () -> expr), {| cexpr |}Delayed)

{| try cexpr with pati -> cexpri |}C   = b.TryWith({| cexpr |}Delayed, (fun v ->
                                                match v with
                                                | (pati:exn) -> {| cexpri |}C
                                                | _ -> raise exn)

{| try cexpr finally expr |}           = b.TryFinally( {| cexpr |}Delayed, (fun () -> expr))

{| trans-cexpr0; cexpr1 |}             = b.Combine({| trans-cexpr0 |}C, {| cexpr1 |}Delayed)

{| other-expr0 ; cexpr1 |}             = other-expr; {| cexpr1 |}C

{| other-expr |}                       = other-expr; b.Zero()
```

Where

- The auxiliary translation {| *cexpr* |}$_{\text{Delayed}}$ is `b.Delay(fun () -> {| cexpr |}C)`.
- A *trans-cexpr*$_0$ is any syntactic expression form given an explicit translation by the above rules, excluding the final rule.
- The auxiliary translation {| _ |}$_{\text{E}}$ converts *expr* to a value compatible with the type `System.Collections.Generic.IEnumerable<ty>`, for some type *ty*, using *enumerable extraction* §6.6.5.

This translation implicitly places type constraints on the expected form of the builder methods. For example, for the `async` builder found in the `Microsoft.FSharp.Control` library these correspond to implementing a builder of a type with 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 computation expression syntax in F# allows the smooth embedding of computational structures known as *monoids* and *monads*.

A monad is a way of automatically doing impedance matching between one kind of computation (e.g. lazy computations) and regular F# coding constructs. The impedance matching is known as "binding" and is inserted automatically by the computation expression machinery at points such as `let!`.

The normal rule when implementing a monad in F# is that you really only need to implement `Bind`, `Return` and (optionally) a "catch" primitive. Everything else you need for the F# computation expression syntactic sugar for a monad is just boilerplate code. An example is given below.

Implementations of the Delay, Combine, For, While, Using, TryFinally and TryWith are needed for three reasons:

1. Some operators such as Delay often have more efficient implementations than their boilerplate implementation in terms of bind and return;

2. You may wish to restrict the use of certain syntactic forms within sequence expressions. For example, try/finally can have subtle semantics and you may simply decide to only allow try/with, or neither;

3. F# computation expressions can also be used to implement monoids, where the Combine and Zero operations do not follow the same pattern, e.g. Combine appends two sequences, and Zero returns the empty sequence, and For concatenates lots of sequences. There is no “bind” in this case.

The example below shows a common approach to implementing a new computation expression builder for a monad. Here we are defining computations that can be "partially run" by running them step-by-step, e.g. up to a time limit.

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

/// 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, e.g.
    ///     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

    /// 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 rest of the operations are entirely boiler plate

    /// The delay operator
    /// This is boiler-plate in terms of “result” and “bind”. It can
    /// be more efficient to implement this directly, e.g.
    ///     let delay f = NotYetDone (fun () -> f())

    let delay f = (result ())  |> bind (fun () -> f())

    /// The tryFinally operator
    /// This is boiler-plate 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 boiler-plate 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 boiler-plate in terms of “result” and “bind”.
    let rec whileLoop gd body =
        if gd() then body |> bind (fun v -> whileLoop gd body)
        else result ()

    /// The sequanial composition operator
    /// This is boiler-plate 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 boiler-plate 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 the F# syntactic sugar
type EventuallyBuilder() =
    member x.Bind(e,k)                 = Eventually.bind k e
    member x.Return(v)                 = Eventually.result 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()
```

Once defined, computations can now be built using `eventually { ... }`:

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

6.4.11 Sequence expressions

An expression of the form

```
{ comp-expr }
```

is a *sequence expression*. By convention, the first form of sequence expression is always prefixed with the use of the identity function `seq`.

```
seq { for x in [1;2;3] -> x+x }
```

Sequence expressions are interpreted as computation expressions with a builder of type `Microsoft.FSharp.Collections.SeqBuilder`. This type is defined as follows

```
type SeqBuilder() =
    member x.Yield (v) = Seq.singleton v
    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
```

Note: This type is not actually defined in the F# library, and the elaboration of sequence expressions is handled by the F# compiler.

This means that a sequence expression generates an object of type `System.Collections.Generic.IEnumerable<ty>` for some type *ty*. This 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.

6.4.12 Range expressions

An expression of the form $expr_1$ `..` $expr_2$ is a *range expression*. Range expressions generate sequences over a given 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 $expr_1$ `..` $expr_2$ are evaluated as a call to the overloaded operator `(..)`, whose default binding is defined in `Microsoft.FSharp.Core.Operators`. This generates an `IEnumerable<_>` for the range of values between the given start ($expr_1$) and finish ($expr_2$) values, using an increment of 1. The operator requires the existence of a static member `(..)` (long name `GetRange`) on the static type of $expr_1$ with an appropriate signature.

Range expressions $expr_1$ `..` $expr_1$ `..` $expr_3$ are evaluated as a call to the overloaded operator `(.. ..)`, whose default binding is defined in `Microsoft.FSharp.Core.Operators`. This generates an `IEnumerable<_>` for the range of values between the given start ($expr_1$) and finish ($expr_3$) values, using an increment of $expr_2$. The operator requires the existence of a static member `(..)` (long name `GetRange`) on the static type of $expr_1$ with an appropriate signature.

Note: The existence of this static member is simulated for types `int`, `byte`, `int16`, `int64` and the unsigned variants of these integer types.

6.4.13 Lists via sequence expressions

An expression of the form `[` *comp-expr* `]` is a *list sequence expression*. It elaborates to `Microsoft.FSharp.Collections.Seq.to_list(seq {` *comp-expr* `})`.

6.4.14 Arrays via sequence expressions

An expression of the form `[|` *comp-expr* `|]` is an *array sequence expression*. It elaborates to `Microsoft.FSharp.Collections.Seq.to_array(seq {` *comp-expr* `})`.

6.4.15 Null expressions

An expression of the form `null` is a *null expression*. This induces a nullness constraint (§5.1.5.2, §5.3.8) on the initial type of the expression. This ensures that the type directly supports the value `null`.

Null expressions are a primitive elaborated form.

6.4.16 The AddressOf Operators

An expression of the form `&expr` or `&&expr` is an *address-of expression*. These take the address of a mutable local variable or byref-valued argument.

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

Note: Use of these operators may result in unverifiable or invalid CIL code , and a warning or error will typically be given if this is possible. Their use is recommend only to pass addresses where `byref` or `nativeptr` parameters are expected, or to pass a byref parameter on to a subsequent function.

Addresses generated by the && operator must not be passed to functions that are in tailcall position. This is not checked by the F# compiler .

Direct uses of the named types `byref` and `nativeptr` and values in the `Microsoft.FSharp.NativeInterop` module may also result in the generation of invalid or unverifiable CIL code. In particular, these types may NOT be used within named types such as tuples or function types, so their use is highly constrained. They may be used as the argument type specifications of `DllImport` annotated functions and class member signatures.

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

The overall expression is elaborated recursively by *taking the address* of the elaborated form of `expr`, written *AddressOf*(`expr`), defined section §6.4.16.1.

6.4.16.1 Taking the address of an elaborated expression

The process of computing the elaborated forms of certain expressions must compute a "reference" to an elaborated expression `expr`, written *AddressOf*(`expr`). The *AddressOf* operation is used for generating the elaborated forms of address-of expressions, assignment expressions, and method and property calls on 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>` then the elaborated form is `&path`.
- If `expr` has form $expr_a.field$ where `field` is a mutable, non-readonly-CLI field then the elaborated form is $\&(AddressOf(expr_a).field)$
- If `expr` has form $expr_a.[expr_b]$ where the operation is an array lookup then the elaborated form is $\&(AddressOf(expr_a).[expr_b])$
- Otherwise, $\&_{lda}v$ where `v` is a fresh mutable local initialized using `let v = expr` as the first binding in the overall elaborated form for the entire assignment expression. This is known as a *defensive copy* of an immutable value.

The *AddressOf* operation is computed under the assumption that the relevant elaborated form "never", "definitely" or "possibly" mutates memory using the resulting pointer. This assumption changes the errors and warnings reported.

- If the operation "definitely" mutates then an error is given if a defensive copy arises.
- If the operation "possibly" mutates then a warning (number 52) is given if a defensive copy arises.

Note that warning 52 “copy due to possible mutation of value type” is suppressed by default. This is because the majority of value types inCLI libraries are immutable.

CLI libraries do not include metadata to indicate whether if particular value type is immutable or not. Unless a value is held in arrays or locations marked mutable may be mutated, or a value type is known to be immutable to the F# compiler, F# inserts copies to ensure that inadvertent mutation does not occur.

6.4.17 'printf' Formats

Format strings are strings with "%" markers indicating format placeholders. Format strings are analyzed by the F# comiler 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 `Microsoft.FSharp.Text.Printf` module. Format strings receive special treatment 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 `Microsoft.FSharp.Text.Format<'Printer,'State,'Residue,'Result,'Tuple>`. The string is statically analyzed to fill in `Format`'s type parameters, of which `'Printer` and `'Tuple` are the most interesting:

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

A format placeholder has the following shape: %[*flags*][*width*][.*precision*][*type*]. Format placeholders in the format string give rise to variations in these types as follows:

`%b`	`bool`
`%s`	`string`
`%d`, `%i`	basic integer type `byte`, `sbyte`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `nativeint` or `unativeint`
`%u`	basic integer type
`%x`	basic integer type
`%X`	basic integer type
`%o`	basic integer type
`%e`, `%E`, `%f`, `%F`, `%g`, `%G`	`float` or `float32`
`%M`	`System.Decimal`
`%O`	`System.Object`
`%A`	a fresh variable type `'T`
`%a`	a formatter of type `'State -> 'T -> 'Residue` for a fresh variable type `'T`
`%t`	a formatter of type `'State -> 'Residue`

Valid flags are: `0`, `-`, `+` , and the space character. The `#` flag is invalid and a compile-time error will be reported if it is used.

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

6.5 Application Expressions

6.5.1 Basic Application Expressions

Application expressions involve variable names, dot-notation lookups, function applications, method applications, type applications and item lookups.

```
    | long-ident-or-op                -- long-ident lookup expression
    | expr '.' long-ident-or-op       -- dot lookup expression
    | expr expr                       -- application expression
    | expr(expr)                      -- high precedence application expression
    | expr<types>                     -- type application expression
    | expr< >                         -- type application expression (empty type list)
    | type expr                       -- simple object expression
```

Some examples of application expressions are:

```
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 exclude 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]
```

The following are also application expressions because of their expansion as syntactic sugar:

```
    | expr infix-op expr               -- infix application expression
    | prefix-op expr                  -- prefix application expression
    | expr.[expr]                     -- indexed lookup expression
    | expr.[slice-range]              -- slice expression (1D)
    | expr.[slice-range, slice-range] -- slice expression (2D)
    | expr.(expr)             -- OCaml-compat array lookup expression
```

The following are processed in a very similar way to application expressions and are covered at the end of this section:

```
    | expr <- expr                    -- assignment expression
```

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`. These are then repeatedly processed in a left-to-right starting with the use of *Unqualified Lookup* (§14.2.1). This in turn used procedures such as *Expression-Qualified Lookup* and *Method Application Resolution*.

As described in §14.2, checking an application expression results in an elaborated expression containing a series of lookups and method calls. These may include

- Uses of named values
- Uses of discriminated union cases and primitive record constructions
- Applications of functions
- Applications of static and instance methods (including those to access properties) and 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 simper primitives:

- The use of a method and or value as a first-class function may result in the addition of additional resolved let bindings and lambda expressions.
    - 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.

6.5.2 Object Construction Expressions

An expression of the form `new ty(e1 ... en)` is an *object construction expression* and 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` may not be an array, record, union or tuple type. If `ty` is a named class or struct type, then

- *ty* must not be abstract.
- If *ty* is a struct type, `n = 0` and the struct type has no constructor method taking 0 arguments, then the expression elaborates to the default “zero-bit pattern” value for the given struct type.
- Otherwise, the type must have one or more accessible constructors. The overloading between these potential constructors is resolved and elaborated using *Method Application Resolution* (see §14.4).

If *ty* is a delegate type then this is a *delegate implementation expression*.

- If the delegate type has an `Invoke` method with a actual signature $Invoke(ty_1,...,ty_n) \rightarrow rty_A$ then the overall expression must be of the form $new\ ty(expr)$ where *expr* has type $ty_1 \rightarrow ... \rightarrow ty_n \rightarrow rty_B$. 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 lambda expression must be given, i.e. the overall expression must be of the form $new\ ty(fun\ pat_1 \ldots pat_n \rightarrow expr_{body})$.

If *ty* is a type variable, then

- there must be no arguments (i.e. `n = 0`)
- the type variable is constrained with constraint

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

- the expression elaborates to a call to `Microsoft.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 nullary object constructor method for the given type. (Note that exceptions returned by this function are wrapped on return using `System.TargetInvocationException`).

6.5.3 Operator Expressions

Operator expressions are specified simply in terms of their shallow syntactic translation to other constructs. Constructs of the following forms translated as follows. For infix and prefix operators:

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

Note that when an operator that may be used as an infix or prefix operator is used in prefix position, a "twiddle" character ~ is added to the name of the operator.

In addition, the name resolution rules (§14.1) imply a resolution for uses of operators that are not otherwise given a definition. In particular, if no other definition of an operator is given, then an operator resolves to an expression that implicitly searches the types of the operator arguments for a corresponding static member with the name of the operator. This means the default definition of all operators not defined in the F# library is to search the types of the operands for a static member with the same name as the operator.

For example, the following example is permitted:

```
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
```

6.5.4 Lookup Expressions

Lookup expressions are specified in terms of their shallow syntactic translation to other constructs:

```
e1.[e2]                 → e1.Item(e2)
e1.[e2] <- e3           → e1.Item(e2,e3)
```

In addition, for the purposes of resolving expressions of this form, CIL array types of rank 1, 2, 3 and 4 are assumed to support a type extension that defines an `Item` property with 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 e_1 is a named type and that type supports the `DefaultMemberAttribute` then the member name identified by that attribute is used instead of `Item`.

6.5.5 Range Expressions

Range expressions are specified in terms of their shallow syntactic translation to other constructs:

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

6.5.6 Slice Expressions

Slice expressions are specified in terms of their shallow syntactic translation to other constructs. For 1D slices:

```
e1.[e2opt .. e3opt]      → e1.GetSlice(arg2,arg3)
e1.[*]                   → e1.GetSlice(None,None)
```

where arg_i is `Some(`e_i^{opt}`)` if e_i^{opt} is present and `None` otherwise. Likewise for 2D slices:

```
e1.[e2opt .. e3opt, e4opt .. e5opt]     → e1.GetSlice(arg2,arg3,arg4,arg5)
e1.[*, e2opt .. e3opt]                  → e1.GetSlice(None,None,arg2,arg3)
e1.[e2opt .. e3opt, *]                  → e1.GetSlice(arg2,arg3,None,None)
e1.[*,*]                                → e1.GetSlice(None,None,None,None)
```

Because this is a shallow syntactic translation, the GetSlice 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.

In addition, CIL array types of rank 1 to 4 are assumed to support a type extension that defines a method `GetSlice` with 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[,]

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[,,,]
```

6.5.7 Assignment Expressions

The expression form $expr_1$ `<-` $expr_2$ is an *assignment expression*. A modified version of *Unqualified Lookup* (§14.2.1) is applied to expression $expr_1$ using a fresh expected result type ty, producing an elaborate expression $expr_1$. This proceeds as normal except that the last qualification for $expr_1$ must resolve to one of the following constructs:

- An invocation of a (possibly indexer) property with a setter method.
    - In this case $expr_2$ is incorporated as the last argument in the method application resolution for the setter method and the overall elaborated expression is a method call to this setter property including the last argument.
- A mutable value $path$ of type ty.
    - In this case $expr_2$ is then checked using expected result type ty, producing an elaborated expression $expr_2$. The overall elaborated expression is an assignment to a value reference $\&path$ `<-`$_{stobj}$ $expr_2$.
- A reference to a value $path$ of type `byref<`ty`>`.
    - In this case $expr_2$ is then checked using expected result type ty, producing an elaborated expression $expr_2$. The overall elaborated expression is an assignment to a value reference $path$ `<-`$_{stobj}$ $expr_2$.

- A reference to a mutable field $expr_{1a}.field$ with actual result type ty.
    - In this case $expr_2$ is then checked using expected result type ty, producing an elaborated expression $expr_2$. The overall elaborated expression is an assignment to a field *AddressOf*($expr_{1a}.field$) $<-_{stobj}$ $expr_2$ (see §6.4.16.1)
- A array lookup $expr_{1a}.[expr_{1b}]$ where $expr_{1a}$ has type $ty[]$.
    - In this case $expr_2$ is then checked using expected result type ty, producing an elaborated expression $expr_2$. The overall elaborated expression is an assignment to a field *AddressOf*($expr_{1a}.[expr_{1b}]$) $<-_{stobj}$ $expr_2$ (see §6.4.16.1)

Note: the above interpretations of assignments means that local values must be mutable in order to mutate their immediate contents using primitive field assignments and array lookups, where "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)
```

6.6 Control Flow Expressions

6.6.1 Parenthesized and Block Expressions

An expression of the form `(expr)` is a *parenthesized expression* and `begin expr end` is a *block expression*.

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`.

6.6.2 Sequential Execution Expressions

An expression of the form $expr_1$`;` $expr_2$ is a *sequential execution expression*. For example:

```
printfn "Hello"; printfn "World"; 3
```

> Note: The `;` token is optional when lightweight syntax is used and the expression e2 occurs on a subsequent line starting on the same column as e1, and when the current pre-parse context resulting from the syntax analysis of the program text is a SeqBlock (see the specification of lightweight syntax later in this specification). In practice this means the token can be omitted 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 this attempt 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 $expr_1; expr_2$.

6.6.3 Conditional Expressions

An expression of the form `if` $expr_1$ `then` $expr_2$ `[ else` $expr_3$ `]` is a *conditional expression*. The *else* branch may be omitted. For example:

```
if (1+1 = 2) then "ok" else "not ok"
if (1+1 = 2) then printfn "ok"
```

The expression form is equivalent to

```
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 -> ()
```

6.6.4 Pattern Matching Expressions and Functions

An expression of the form `match` *expr* `with` *rules* is a *pattern matching expression* and evaluates the given expression and selects a rule via pattern matching (§7). For example:

```
match (3,2) with
  | 1,j -> printfn "j = %d" j
  | i,2 -> printfn "i = %d" i
  | _   -> printfn "no match"
```

An expression of the form `function` *rules* is a *pattern matching function* and is syntactic sugar for a single argument lambda expression followed by immediate matches on the argument. For example,

```
function
  | 1,j -> printfn "j = %d" j
  | _   -> printfn "no match"
```

Is syntactic sugar for:

```
fun x ->
  match x with
  | 1,j -> printfn "j = %d" j
  | _   -> printfn "no match"
```

where `x` is a fresh variable.

6.6.5 Sequence Iteration Expressions

An expression of the form `for` *pat* `in` $expr_1$ `do` $expr_2$ `done` is a *sequence iteration expression*. For example:

```
for x,y in [(1,2); (3,4)] do
    printfn "x = %d, y = %d" x y
```

> Note: The `done` token is optional when lightweight syntax is used and $expr_2$ occurs indented from the column position of the `for` and on a subsequent line. The `done` token is automatically inserted when the pre-parse context associated with the `for` token is closed.

The expression $expr_1$ is checked with a fresh expected type ty_{expr} which is then asserted to be compatible with the type `IEnumerable<`*ty*`>`, for a fresh type *ty*. If this 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() in
try
    while (v.MoveNext()) do
        match v.Current with
        | pat -> expr2
        | _ -> ()
    done
finally
    match box(v) with
    | :? System.IDisposable as d -> d.Dispose()
    | _ -> ()
```

If the assertion fails, then the type ty_{expr} may also be of any static type that satisfies the "collection pattern" of CLI libraries, in which case it is enumerated via a process known as *enumerable extraction*. In particular, ty_{expr} may be any type that has an accessible `GetEnumerator` method accepting one argument and returning a value with accessible `MoveNext` and `Current` properties. In this case the loop is evaluated in much the same way, except a dynamic check is inserted to detect if the enumerator satisfies `IDisposable`. The type of *pat* is determined by 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, then that type is used instead, and a dynamic cast is inserted to convert `v.Current` to ty_2.

6.6.6 Try-catch Expressions.

An expression of the form `try` *expr* `with` *rules* is a *try-catch expression*. For example:

```
try "1" with _ -> "2"

try
    failwith "fail"
with
   | Failure msg -> "caught"
   | :? System.InvalidOperationException -> "unexpected"
```

Expression *expr* is checked with the same expected as the overall expression. The pattern matching clauses are then checked with the same initial type and with input type `System.Exception`.

Try-catch expressions are a primitive elaborated form.

The F# library function `rethrow()` may be used only in the immediate right-hand-sides of *rules*.

```
try
    failwith "fail"
with e -> printfn "Failing"; rethrow()
```

6.6.7 Try-finally Expressions

`try` $expr_1$ `finally` $expr_2$ is a *try-finally expression*. For example:

```
try "1" finally printfn "Finally!"

try
    failwith "fail"
finally
   printfn "Finally block"
```

Expression $expr_1$ is checked with the same expected as the overall expression. Expression $expr_2$ is checked with arbitrary initial type, and a warning is given if this type can't then be asserted to be equal to `unit`.

Try-finally expressions are a primitive elaborated form.

6.6.8 While Expressions

An expression of the form `while` $expr_1$ `do` $expr_2$ `done` is a *while loop expression*. For example

```
while System.DateTime.Today.DayOfWeek = System.DayOfWeek.Monday do
    printfn "I don't like Mondays"
```

> Note: The `done` token is optional when lightweight syntax is used and $expr_2$ occurs indented from the column position of the `while` and on a subsequent line. The `done` token is automatically inserted when the pre-parse context associated with the `while` token is closed.

The overall type of the expression is `unit`. The expression $expr_1$ is checked with expected type `bool`. A warning will be reported if the body $expr_2$ of the `while` loop cannot be asserted to have type `unit`.

6.6.9 Simple for-Loop Expressions

An expression of the form `for` *var* `=` $expr_1$ `to` $expr_2$ `do` $expr_3$ `done` is a *simple for loop expression*. For example

```
for x = 1 to 30 do
    printfn "x = %d, x^2 = %d" x (x*x)
```

> Note: The `done` token is optional when lightweight syntax is used and e2 occurs indented from the column position of the `for` and on a subsequent line. The `done` token is automatically inserted when the pre-parse context associated with the `for` token is closed.

> Note: The expression form `for` *var* `=` $expr_1$ `downto` $expr_2$ `do` $expr_3$ is also permitted for compatibility with OCaml

The bounds $expr_1$ and $expr_2$ are checked with expected type `int`. The overall type of the expression is `unit`. A warning will be reported if the body $expr_3$ of the `for` loop does not have static type `unit`.

The elaborated form of a simple for loop expression is:

```
let start = expr1 in
let finish = expr2 in
for var = start to finish do expr3 done
```

for fresh variables `start` and `finish`.

For-loops over ranges specified by variables are a primitive elaborated form.

6.6.10 Assertion Expressions

An expression of the form `assert(expr)` is an *assertion expression* .

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 will not trigger unless the DEBUG conditional compilation symbol is defined.

6.7 Binding Expressions

6.7.1 Binding Expressions

An expression of the form `let` *binding*$_1$ `and ... and` *binding*$_n$ `in` *body-expr* is a *binding expression* and establishes bindings within the local lexical scope of *body-expr* and has the same overall type as *body-expr*. For example:

```
let x = 1 in x+x

let x,y = ("One", 1) in x.Length + y

let id x = x in (id 3, id "Three")

let swap (x,y) = (y,x) in List.map swap [(1,2); (3,4)]

let K x y = x in List.map (K 3) [1;2;3;4]
```

Note: The `in` token is optional when lightweight syntax is used and the body expr occurs on a subsequent line starting on the same column as the `let`. The `in` token is automatically inserted when the pre-parse context associated with the `let` token is closed due to the alignment of the body expression.

If multiple bindings are used, the variables bound are only in scope in *body-expr*. For example, the following is legal:

```
let x = 1 and y = 2 in x+y
```

Binding expressions are checked by first checking the *bindings* via the rules described in (§6.7.1.1) below. After the bindings of a "let" expression are checked, the *body-expr* is checked against the expected type of the overall expression. The resulting elaborated form of the entire expression is

```
let ident11 <typars11> = expr11 in
let ident12 <typars12> = expr12 in
...
let identnm <typarsnm> = exprnm in
body-expr.
```

where each $ident_{ij}$, $typars_{ij}$ and $expr_{ij}$ is as defined in §6.7.1.1.

6.7.1.1 Checking "let" bindings

Each $binding_i$ is either a *function definition*:

```
[inline] identi1 pati1 ... patin [ : return-typei ] = rhs-expri
```

or a *value definition*, which defines one or more values by matching a pattern against an expression:

```
[mutable] pati [ : typei ] = rhs-expri
```

Each value binding pat_i = *rhs-expr*$_i$ is processed as follows:

- The pattern pat_i is checked against a fresh expected type ty_i, or $type_i$ is present. This results in zero or more identifiers $ident_{i1}$ ... $ident_{im}$ each of type ty_{i1} ... ty_{im}.

- The expression $rhs\text{-}expr_i$ is checked against expected type ty_i, giving an elaborated form $expr_i$.

Each function binding $ident_{i1}\ pat_{i1}\ \ldots\ pat_{in} = rhs\text{-}expr_i$ is processed as follows:

- If $ident_{i1}$ is an active pattern identifier then active pattern result tags are added to the environment (§10.5.5)
- The expression $(fun\ pat_{i1}\ \ldots\ pat_{in} \rightarrow rhs\text{-}expr_i)$ is checked against a fresh expected type ty_i and reduced to an elaborated form $expr_i$.

After processing each value is generalized (§14.7) yielding generic type parameters $<typars_{ij}>$.

Note that:

- All $ident_{ij}$ must be distinct
- Function bindings may not be `mutable`. Mutable function values should be written `let mutable f = (fun args -> ...)`.
- Value definitions may not be `inline`.
- The patterns of functions may not include optional arguments (§8.12.5).

After all bindings have been processed each identifier $ident_{ij}$ is added to the name resolution environment (REF) with type ty_{ij}. The identifiers $ident_{ij}$ are not in scope in any $rhs\text{-}expr_i$.

“Let” bindings in expressions are subject to the following rules, which differ from those applied to let bindings in class definitions (§8.5.1.3), modules (§10.5) and computation expressions (§6.4.10):

- Expression bindings may not define explicit type parameters (§5.2). That is, the expression

  ```
  let f<'T> (x:'T) = x in f 3
  ```

  is rejected.
- Expression bindings are not public and are not subject to arity analysis (§14.11).
- Any custom attributes specified on the declaration, parameters and/or return arguments are ignored and a warning is given if these are present. As a result, expression bindings may not have the `ThreadStaticAttribute` or `ContextStaticAttribute` attributes.

6.7.1.2 *Ambiguities at "let" bindings*

An ambiguity exists between the two different kinds of bindings. In particular,

```
ident pat = expr
```

can be interpreted as either a function or value binding.

```
| ident pat = expr      -- function binding
| pat = expr            -- value binding
```

This ambiguity is always resolved as a function binding.

For example,

```
type foo = Id of int

let Id x = x
```

Here it may seem ambiguous if `Id x` is a pattern matching values of type `foo` or if we are a defining a function called `Id`. In F# this is always resolved as a function binding. To make this a pattern use

```
let v = if 3=4 then Id "yes" else Id "no"

let (Id answer) = v
```

6.7.1.3 *Mutable locals*

Let-bound variables 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 under the same restrictions as values of type `byref<_>` (§14.10), and are similarly implicitly dereferenced.

6.7.2 Recursive Binding Expressions.

An expression of the form `let rec` *`bindings`* `in` *`expr`* is a *recursive binding expression*.

The variables bound are available for use within their own definitions, i.e. within all of the expressions on the right-hand-side of the bindings in *`bindings`*.

Each binding may specify zero or more argument patterns. It is normal that each binding specifies a function (i.e. has at least one argument pattern). In this case the bindings define a set of recursive functions.

When one or more of the bindings specify a value, the recursive expressions are analyzed for safety (§14.12). This may give rise to warnings (including some reported as compile-time errors) and runtime checks.

Recursive bindings where the values on the right-hand-side are functions or lazy evaluations are a primitive elaborated form.

Within a set of recursive bindings any uses of a recursive binding gives rise to immediate constraints on the recursively bound construct. For example, consider the following declaration:

```
let rec f x =
    let a = f 1        // constrains "x" to be type int
    let b = f "Hello"  // constrains "x" to be type string
    x
```

This declaration is not valid because the recursive uses of `f` have given rise to inconsistent constraints on `x`.

If the type of the target has been given a full signature, including a closed set of type parameters and annotations relating the argument types to those parameters, then recursive calls at different types are permitted. For example:

```
module M =
    let rec f<'T> (x:'T) : 'T =
        let a = f 1
        let b = f "Hello"
        x
```

6.7.3 Deterministic Disposal Expressions

An expression of the form `use` *ident* `=` $expr_1$ `in` $expr_2$ is a *deterministic disposal expression*. 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* `=` $expr_1$ `in` $expr_2$ (§6.7.1), giving an elaborated expression of the form

`let` $ident_1$ `:` ty_1 `=` $expr_1$ `in` $expr_2$.

Only one variable may be defined by the binding, and the binding is not generalized. The type ty_1, is then asserted to be a subtype of `System.IDisposable`. The `Dispose` method is called on the variable's value when the variable goes out of scope, though only if the dynamic value of the expression when coerced to type `obj` is non-null. Thus the overall expression elaborates to

```
let ident = expr1 in
try expr2
finally (match (ident :> obj) with
         | null -> ()
         | _ -> (ident :> System.IDisposable).Dispose())
```

6.8 Type-Related Expressions

6.8.1 Rigid Type Annotation Expressions

An expression of the form *expr* `:` *ty* is a *type annotated expression* where *ty* is an *inflexible type constraint*. For example:

```
(1 : int)
let f x = (x : string) + x
```

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*.

> Note: This ensures information from the annotation will be used during the analysis of `expr` itself.

6.8.2 Static Coercion Expressions

An expression of the form *expr* `:>` *ty* is a *static coercion expression*, i.e. a flexible type constraint. The expression `upcast(`*expr*`)` is equivalent to *expr* `:> _`, so the target type is 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.

6.8.3 Dynamic Type Test Expressions

An expression of the form *expr* :? *ty* is a dynamic type test expression. 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 ty_e coerces to *ty*
- The type ty_e must not be sealed
- If type *ty* is sealed, or *ty* is a variable type, or type ty_e is not an interface type then *ty* :> ty_e is asserted

Dynamic type tests are a primitive elaborated form.

6.8.4 Dynamic Coercion Expressions

An expression of the form *expr* :?> *ty* is a dynamic coercion expression. downcast(*e1*) is equivalent to *expr* :?> _, so the target type is 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 ty_e coerces to *ty*
- The type ty_e must not be sealed
- If type *ty* is sealed, or *ty* is a variable type, or type ty_e is not an interface type then *ty* :> ty_e is asserted

Dynamic coercions are a primitive elaborated form.

6.9 Expression Quotations

An *expression quotation* <@ *expr* @>captures a typed abstract syntax tree form of the enclosed expression. For example:

```
<@ 1 + 1 @>

<@ (fun x -> x + 1) @>
```

The expected type of the overall expression is asserted to be of the form `Microsoft.FSharp.Quotations.Expr<ty>` for a fresh type *ty*. The expression *expr* is checked with expected type *ty*. Quotations may contain references to values, e.g.

```
let f (x:int) = <@ x + 1 @>
```

In this case the value appears in the expression tree as a node of kind `Microsoft.FSharp.Quotations.Expr.Value`.

Quotations may also contain splices `%expr` that place the given expression tree as the corresponding expression tree node:

```
let f (v:int expr) = <@ %v + 1 @>
```

The exact nodes appearing in the quotation tree are dictated by the elaborated form of *expr* arising from checking.

6.9.1 Raw Expression Quotations

A *raw expression quotation* `<@@ expr @@>` is similar to a normal expression quotation but drops the type annotation. In particular the expected type of the overall expression is asserted to be of the form `Microsoft.FSharp.Quotations.Expr`. The expression *expr* is checked with fresh expected type *ty*. For example:

```
<@@ 1 + 1 @@>

<@@ (fun x -> x + 1) @@>
```

6.10 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.

Work in progress

The execution of elaborated F# expressions results in values. Values include

- Primitive constant values
- References to object values
- The special value `null`
- Values for value types, containing a value for each field in the value type

Evaluation assumes a global pool (heap) of identified object values for reference types and boxed value types, each containing

- A runtime type
- Fields
- Method bodies
- A possible discriminated union case label
- A closure assigning values to all values referenced in the method bodies associated with the object

6.10.1 Zero Values

All ground types have a *zero value*. This is the "default" value for the type in the CLI execution apparatus:

- 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>`.

6.10.2 Evaluating Value References

For elaborated value references `v`, the value corresponding to `v` in the environment is returned.

6.10.3 Evaluating Function Applications

For elaborated applications of functions, evaluation is defined with respect to sequences of applications of the form `((f e1) ... eN)`. The constituent expressions are evaluated and the body of the function value resulting from the `f` is executed with the first formal parameter assigned the value of the first actual argument, and additional arguments are applied iteratively to further resulting function values. Argument evaluations and applications occur in an unspecified order and may be interleaved, though arguments are always evaluated prior to their corresponding applications.

6.10.4 Evaluating Method Applications

For elaborated applications of methods, the elaborated form of the expression will be either *`expr.M(args)`* or *`M(args)`*.

- The (optional) *`expr`* and *`args`* are evaluated in left-to-right order and the body of the member evaluated in an environment with formal arguments mapped to corresponding argument values.
- If *`expr`* evaluates to `null` then `NullReferenceException` is raised.
- If the method called is a virtual dispatch slot (i.e. a method declared `abstract`) then the body of the member is chosen according to the dispatch maps of the value of *`expr`*.

6.10.5 Evaluating Discriminated Union Cases

For elaborated uses of a discriminated union case *`Case(args)`* for a type *`ty`*, the arguments are evaluated in left-to-right order and an object value returned compatible with the case *`Case`*. The CLI runtime type of the object is either `ty` or some type compatible with `ty`.

If the type `ty` uses null as a representation (§5.3.8) and *`Case`* is the single, nullary discriminated union case, then the generated value is `null`.

6.10.6 Evaluating Field Lookups

For elaborated lookups of CLI and F# fields, the resolved form of the expression will be either *`expr.F`* for an instance field or *`F`* for a static field. The (optional) `expr` is evaluated and the field read. If `expr` evaluates to `null` then `NullReferenceException` is raised.

6.10.7 Evaluating Active Pattern Results

For active pattern result references (see §10.5.5) for result `i` in an active pattern with `N` possible results of types *`types`*, the elaborated expression form is a object construction of an object of type `Microsoft.FSharp.Core.Choice<`*`types`*`>` using a discriminated union case `Choice`*`N_i`*.

6.10.8 Evaluating Array Expressions

When executed, an elaborated array expression evaluates each expression in turn in left-to-right order, returning a new array containing the given values.

6.10.9 Evaluating Record Expressions

When evaluated, primitive record constructions evaluate their constituent expressions in left-to-right order and build an object of type `R<ty1,...,tyN>`.

6.10.10 Evaluating Function Expressions

Function expressions `(fun v1 … vn -> expr)` are a primitive elaborated form.

If only one variable is present, then whenever the function value is invoked at some later point the variable v_1 will be bound to the input argument, the expression *expr* will be evaluated and its value will be the result of the corresponding function invocation.

If multiple patterns are present, then the function expression evaluates to a curried function value. The result of applying the curried function value to one argument is a residual function value accepting *n-1* arguments. Whenever *n* arguments have been received in total the arguments are matched against the input patterns, any results from the match are bound and the expression `expr` is evaluated and returned as the result of the application.

The result of calling the `obj.GetType()` method on the resulting object is under-specified (see §6.10.22).

6.10.11 Evaluating Object Expressions

Object expressions `{ new ty(args) with bindings interface-impls }` are a primitive elaborated form and execute by creating an object whose runtime type is compatible with all of the ty_i and whose dispatch map maps dispatch slots to their implementing members. The base construction expression is executed as the first step in the construction of the object.

The result of calling the `obj.GetType()` method on the resulting object is under-specified (see §6.10.22).

6.10.12 Evaluating Binding Expressions

For each binding *pat = rhs-expr*, the right-hand-side expression is evaluated and then matched against the given patterns to produce a collection of bindings establishing values for the identifiers bound by the pattern.

6.10.13 Evaluating For Loops

For-loops `for var = expr1 to expr2 do expr3 done` over ranges specified by variables are a primitive elaborated form.

Expressions $expr_1$ and $expr_2$ are evaluated once, then expression $expr_3$ is evaluated repeatedly with the variable *var* bound to successive values in the range of $expr_1$ up to $expr_2$. If $expr_1$ is greater than $expr_2$ then $expr_3$ is never evaluated.

6.10.14 Evaluating While Loops

While-loops `while expr1 do expr2 done` are a primitive elaborated form.

Expression $expr_1$ is evaluated. If its value is `true` expression $expr_2$ is evaluated, and the loop is evaluated once more. If expression $expr_1$ evaluates to `false` the loop terminates.

6.10.15 Evaluating Static Coercion Expressions

At runtime, a boxing coercion is inserted if ty_e is a value type and ty is a reference type.

6.10.16 Evaluating Dynamic Type Test Expressions

Elaborated expressions of the form *expr :? ty* evaluate as follows:

- *expr* is evaluated to a value *v*.
- If *v* is `null`, then
  - If ty_e uses `null` as a representation (§5.3.8), the result is `true`
  - Otherwise the expression evaluates to `false`
- If *v* is not `null` and has dynamic type *vty*, and *vty dynamically converts to ty* (§5.3.9) then the expression evaluates to `true`.
  - An exception is made if *vty* is an enumeration type, in which case *ty* must be precisely *vty*.
  - Otherwise the expression evaluates to `false`

6.10.17 Evaluating Dynamic Coercion Expressions

Elaborated expressions of the form *expr :?> ty* evaluate as follows:

- *expr* is evaluated to a value *v*.
- If *v* is `null`
  - If ty_e uses `null` as a representation (§5.3.8) then the result is the `null` value.
  - Otherwise a `NullReferenceException` is raised.
- If *v* is not `null`
  - If *v* has dynamic type *vty*, and *vty dynamically converts to ty* (§5.3.9) then the expression evaluates to the dynamic conversion of `v` to *ty*. This means an unboxing coercion is inserted if ty_e is a reference type and *ty* is a value type.
  - Otherwise an `InvalidCastException` is raised.

Note that expressions of the form *expr :?> ty* evaluate in the same way as uses of the F# library function *unbox<ty>(expr)*.

Note: Some F# types use null as a representation for efficiency reasons (§5.3.8), most notably the option<_> type. This only happens when the `CompilationRepresentation(CompilationRepresentationFlags.UseNullAsTrueValue)` attribute is added to a discriminated union with a single nullary (zero-argument) case. For these types, boxing and unboxing can lose type distinctions, e.g.

```
> (box([]:string list) :?> int list);;
System.InvalidCastException: Unable to cast object of type
'_op_Nil[System.String]' to type
'Microsoft.FSharp.Collections.List`1[System.Int32]'.

> (box(None:string option) :?> int option);;
val it : int option = None
```

6.10.18 Evaluating Sequential Execution Expressions

Sequential expressions *e1; e2* are a primitive elaborated form. The expressions *e1* is evaluated, its result discarded, then expression *e2* is evaluated to a value *v*. The result of the overall expression is *v*.

6.10.19 Evaluating Try-catch Expressions

Try/catch expressions `try` *expr* `with` *rules* are a primitive elaborated form. The expression *expr* is evaluated and if an exception occurs then the pattern rules are executed against the resulting exception value. If no rule matches the exception is rethrown.

The special function `rethrow()` may be used in the right-hand-sides of *rules*, though may not be used inside any inner closure (REF).

```
try
    failwith "fail"
with e -> printfn "Failing"; rethrow(e)
```

When executed it continues the exception processing mechanism with the original exception information.

6.10.20 Evaluating Try-finally Expressions

Expressions of the form `try` *e1* `finally` *e2* are a primitive elaborated form. *e1* is evaluated to a value *v* and *e2* is then executed regardless of whether an exception was raised by evaluation of *e1*. The result of the overall expression is *v*.

6.10.21 Evaluating AddressOf Expressions

Note: Work in progress

The underlying CIL execution machinery used by F# supports "co-variant" arrays, witnessed by the fact that the type `string[]` *dynamically converts to* `obj[]` (§5.3.9) .

While this feature is almost never used in F#, its existence means that array assignments and taking the address of array elements may fail at runtime with an `System.ArrayTypeMismatchException` if the runtime type of the target array doesn't match the pointer type being extracted do not match.

For example, the following code will fail 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])
```

6.10.22 Types with Under-specified Object and Type Identity

The CLI and F# support operations that detect whether two object references refer to the same "physical" object. For example, `System.Object.ReferenceEquals(obj,obj)` returns true if the two objects 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 to register/de-register event handlers are based on the object identity of delegate values.

For the following F# types the results of these operations are under-specified:

- Function types
- Tuple types
- Immutable record types
- Discriminated union types
- Boxed immutable value types

For two values of these types, the results of `System.Object.ReferenceEquals` and other object-identity operations are under-specified. This means that an implementation of F# does not have to define the results of these operations for values of these types, beyond ensuring that the operations terminate and do not raise exceptions.

Likewise, for function values and objects returned by object expressions, the results of the following operations are under-specified in the same way:

- `Object.GetHashCode()`
- `Object.GetType()`

Likewise for discriminated union types the results of the following operations are under-specified in the same way:

- `Object.GetType()`

6.11 Constant Expressions

Constant expressions are

- simple constant expressions
- references to literals

The expression used to define a literal value (§10.5.3) must be a constant expression.

7 Patterns

Patterns are used to perform simultaneous case analysis and decomposition on values in conjunction with the `match`, `try...with`, `function`, `fun` and `let` expression and declaration constructs. Rules are attempted in order, top-to-bottom, left-to-right.

```
rule :=
  | pat pattern-gaurd_opt ->  expr          -- pattern, optional guard and action

pattern-guard := when expr

pat :=
  | const                                  -- constant pattern
  | long-ident pat-param_opt pat_opt       -- named pattern
  | pat as ident                           -- "as" pattern
  | pat '|' pat                            -- "or" pattern
  | pat '&' pat                            -- "and" pattern
  | pat :: pat                             -- "cons" pattern
  | [ pat ; ... ; pat ]                    -- list pattern
  | [| pat ; ... ; pat |]                  -- array pattern
  | (pat)                                  -- parenthesized pattern
  | pat,...,pat                            -- tuple pattern
  | { field-pat ; ... ; field-pat }        -- record pattern
  | _                                      -- wildcard pattern
  | pat : type                             -- pattern with type constraint
  | :? type                                -- dynamic type test pattern
  | :? type as ident                       -- dynamic type test pattern
  | null                                   -- null-test pattern
  | attributes pat                         -- pattern with attributes
  | access pat                             -- pattern with accessibility

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
  | 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* containing constructs such as the following:

- Conditionals on integers and other constants
- Switches on discriminated union cases
- Conditionals on runtime types
- Null tests
- Binding expressions
- An array of right-hand-targets referred to by index

Attributes and accessibility annotations on patterns are not permitted except on patterns that are in argument position for a let-binding in a module or a member-binding in a type.

> To do: an informal decision has been made that the process of pattern match compilation will be described in detail as part of this specification, because the compiled form is visible through expression quotation. Note that further optimizations can be made on the generated code post the generation of the elaborated form.
>
> Furthermore this pins down precisely the semantics to ascribe to active patterns.
>
> The exact reduced/elaborated form to use as the target is yet to be decided, though will be loosely based on the typed abstract syntax tree used by the current F# compiler.

7.1.1 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 -> 2
   | 1 -> 0
   | 2 -> 1
   | _ -> failwith "rotate3"
```

Any constant listed in §6.4.1 may be used as a constant pattern.

Simple constant patterns have the corresponding simple type and elaborate to a call to the F# generic equality function `Microsoft.FSharp.Core.Operators.(=)` taking the relevant match input and constant as arguments. The match succeeds if this call returns `true`, otherwise the match fails.

> Note: The use of `Microsoft.FSharp.Core.Operators.(=)` means that CLI floating point equality is used for matching floating point values, and CLI ordinal string equality is used for string matching.

7.1.2 Named Patterns

Patterns for the following forms are *named patterns*:

Long-ident

Long-ident pat

Long-ident pat-params pat

If *Long-ident* has length > 1 or begins with an uppercase identifier (i.e. System.Char.IsUpper is true and System.Char.IsLower is false on the first character), then it resolved using *Name Resolution in Patterns* (§14.1.5). This produces one of:

- A discriminated union case
- An exception label
- An active pattern case
- A literal value

If no resolution is available, then *long-ident* must be a single identifier *ident* and the pattern is a *variable binding pattern* and represents a variable that is bound by the pattern. During checking, the variable is assigned the same value and type as the value being matched.

If *Long-ident* has length > 1 and does not begin with an uppercase identifier then is it interpreted as a variable binding pattern.

7.1.3 Discriminated union patterns

If *Long-ident* resolves to a discriminated union case, then the pattern is a discriminated union pattern. For example:

```
type Data =
    | Kind1 of int * int
    | Kind2 of string * string

let data = Kind1(3,2)

match data with
| Kind1(a,b) -> a+b
| Kind2(s1,s2) -> s1.Length + s2.Length
```

Assuming *long-ident* resolves to a discriminated union case *Case*, then *long-ident* and *long-ident pat* are patterns that recognize values compatible with *Case*. The first form is used if the corresponding case takes no arguments, the second if it takes arguments.

On a successful match the data values carried by the discriminated union are matched against the given argument pattern.

7.1.4 Literal patterns

If *Long-ident* resolves to a literal value, then the pattern is a literal pattern. For example:

```
[<Literal>]
let Case1 = 100

[<Literal>]
let Case2 = 100

begin match 1 with
| Case1 -> "Case1"
| Case2 -> "Case1"
| _ -> "Some other case"
end
```

7.1.5 Active patterns

If *Long-ident* resolves to an active pattern label, then the pattern is an active pattern. For example:

```
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"

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"
```

Assuming the active recognizer case is associated with a function value *f*, then the pattern applies *f* to the input being matched. The name of *f* must be of the form `(|`id_1`|...|`id_N`|)` or `(|`id_1`|_|)` where *id* is id_k for one of the identifiers in this list. *f* must be a value of function type ty_{in} `->` ty_{out} where ty_{in} is the type of input being matched and ty_{out} must be `Microsoft.FSharp.Core.Choice<`ty_1`,...,`ty_N`>`, or `Microsoft.FSharp.Core.Option<`ty_1`>` for a partial pattern.

When executed, the pattern matches if the active recognizer returns `ChoiceN_`*k*`(`*v*`)` or `Some(`*v*`)`. If present the pattern argument *pat* is then matched against *v*.

An active recognizer is only executed if required based on a left-to-right, top-to-bottom reading of the entire pattern. For example, given

```
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
```

Then the results of the following are as indicated:

```
f 0 // 0
f 1 // 1
f 2 // failwith "x is two"
f 3 // failwith "x is three"
f 4 // failwith "got to C"
```

A pattern matching function may be executed multiple times against the same input when resolving a single overall pattern match: the precise number of times the active recognizer is executed is implementation dependent.

7.1.6 Parameterized active patterns

Assuming *long-ident* resolves to an active recognizer case associated with a function value *f*, then

long-ident pat-params pat

is a *parameterized active pattern*. For example:

```
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"
```

These are similar to unparameterized active patterns, but the *pat-params* are interpreted as expressions passed as arguments to the active recognizer function. To avoid ambiguities, syntactically the *pat-params* are patterns, and are limited to expression-like forms which are interpreted as expressions in the corresponding way. Only a limited range of expressions can be used at this point given the syntactic limitations of patterns. The expressions are evaluated as part of the evaluation of the application of the active to the input being examined.

7.1.7 'As' Patterns

The pattern *pat* `as` *ident* binds the given name to the input value and also matches the value against the given pattern. For example:

```
let t1 = (1,2)
let (x,y) as t2 = t1
printfn "%d-%d-%A" x y t2  // 1-2-(1,2)
```

7.1.8 Union Patterns

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. 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
```

7.1.9 'And' Patterns

The pattern *pat* `&` *pat* matches the input against both patterns, binding any variables that appear in either. For example:

```
let (|MultipleOf|_|) n inp = if inp%n = 0 then Some (inp/n) else None

match 56 with
| MultipleOf 4 m & MultipleOf 7 n -> printfn "x = 4*%d = 7*%d" m n
| _ -> printfn "not a multiple of both 4 and 7"
```

7.1.10 'Cons' and List Patterns

The pattern *pat* `::` *pat* is a data recognizer pattern matching the 'cons' case of F# list values. Likewise `[]` matches the empty list. Likewise `[`*pat* `;` ... `;` *pat*`]` is syntactic sugar for a series of `::` and empty list patterns. For example:

```
let rec count x =
    match x with
    | [] -> 0
    | [_] -> 1
    | [_;_] -> 2
    | _ :: t -> count t + 1
```

7.1.11 Type Annotated Patterns

The pattern *pat* `:` *type* is a type annotation enforcing the type of the value matched by the pattern to be equal to the given type. For example:

```
let rec sum (x:int list) =
    match x with
    | [] -> 0
    | (h:int) :: t -> h + sum t
```

7.1.12 Dynamic Type Test Patterns

The pattern `:?` *type* [`as` *ident*] is a *dynamic type test pattern*. This pattern matches any value whose runtime type is the given type or a subtype of the given type. For example:

```
open System
let message (x : Exception) =
    match x with
    | :? System.OperationCanceledException -> "cancelled"
    | :? System.ArgumentException -> "invalid argument"
    | _ -> "unknown error"
```

If present the identifier after `as` is bound to the value coerced to the given type, for example:

```
open System
let findLength (x : obj) =
    match x with
    | :? string as s -> s.Length
    | _ -> 0
```

A warning will be emitted if the input type cannot be statically determined to be a subtype of *type*. An error will be reported if the type test will always succeed.

If the pattern input `e` has type ty_{in}, the pattern is checked using the same conditions as both a dynamic type test expression *e* `:?` *ty* and a dynamic coercion expression *e* `:?>` *ty*.

Dynamic type test patterns are checked for redundancy taking the coercion into account:

```
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 have returned true. The value of the pattern is bound to the results of a dynamic coercion expression *e* `:?>` *ty*.

7.1.13 Record Patterns

The pattern `{ long-ident₁ = pat₁; ... ; long-identₙ = patₙ}` is a *record pattern*. 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-identᵢ` 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 be specified in the pattern.

7.1.14 Array Patterns

The pattern `[|pat ; ... ; pat|]` is an *array pattern* matching arrays of the given length. For example:

```
let checkPackets data =
    match data with
    | [| "HeaderA"; data1; data2 |] -> (data1,data2)
    | [| "HeaderB"; data2; data1 |] -> (data1,data2)
    | _ -> failwith "unknown packet"
```

7.1.15 Null Patterns

The pattern `null` is a *null pattern* that matches values 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 and hence this value is generally used only for checking values coming from CLI method calls and properties. However, some F# types do use null as a representation, see §5.3.8.

7.1.16 Guarded Pattern Rules

Rules of the form `pat when expr` are guarded rules. For example:

```
let categorize x =
    match x with
    | _ when x < 0 -> -1
    | _ when x < 0 -> 1
    | _ -> 0
```

The guards on rules are executed only once the match value has matched the pattern associated with a rule. For example

```
match (1,2) with
| (3,x) when (printfn "not printed"; true) -> 0
| (_,y) -> y
```

evaluates to `2` with no output.

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
  | attributes type-defn      -- type definition with attributes

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 type * ... * type             -- n-ary union case
  | ident : uncurried-sig                  -- n-ary union case

record-type-defn :=
  | type-name = '{' record-fields '}' type-extension-elements_opt

record-fields :=
  | record-short-field-spec ; ... ; record-short-field-spec ;_opt

record-short-field-spec :=
  | 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

object-val := as ident

class-type-body :=
  | class-inherits-decl_opt class-let-bindings_opt type-defn-elements_opt

class-inherits-decl := inherit type expr_opt

class-let-binding :=
  | attributes_opt static_opt let rec_opt bindings    -- binding
  | attributes_opt static_opt do expr                 -- side-effect binding

struct-type-defn :=
  | type-name primary-constr-args_opt object-val_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-signature

delegate-signature :=
  | delegate of uncurried-sig              -- CLI delegate definition

type-extension :=
  | type-name type-extension-elements

type-extension-elements := when type-defn-elements end

type-defn-element :=
  | field-spec
  | member-defn
  | interface-impl

type-defn-elements := when type-defn-element ... type-defn-element  end

primary-constr-args :=
  | attributes_opt access_opt (pat, ... ,  pat)

additional-constr-defn :=
  | access_opt new pat object-val = 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 val-decls in  additional-constr-expr   -- binding construction
  | additional-constr-init-expr

additional-constr-init-expr :=
  | '{' class-inherits-decl field-binds '}'    -- explicit construction
  | new type expr                              -- delegated construction

member-defn :=
  | attributes_opt static_opt member access_opt member-binding  -- concrete member
  | attributes_opt abstract access_opt member-sig              -- abstract member
  | attributes_opt override access_opt member-binding          -- override member
  | attributes_opt default access_opt member-binding           -- override member
  | attributes_opt additional-constr-defn                     -- additional constructor
  | attributes_opt field-spec                                  -- value member

member-binding :=
  | ident._opt binding                         -- method or property definition
  | ident._opt ident with bindings             -- property definition

member-sig :=
```

```
  | ident typar-defns_opt : curried-sig              -- member signature
  | ident typar-defns_opt : curried-sig with get     -- property signature
  | ident typar-defns_opt : curried-sig with set     -- property signature
  | ident typar-defns_opt : curried-sig with get,set -- property signature
  | ident typar-defns_opt : 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 :=
  | attributes_opt arg-name-spec_opt type

arg-name-spec := ?_opt ident :

field-spec :=
  | static_opt val mutable_opt access_opt ident : 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

Type definitions are one of the following kinds

- Type abbreviations (§8.2)
- Record type definitions (§8.3)
- Union type definitions (§8.4)
- Class type definitions (§8.5)
- Interface type definitions (§8.6)
- Struct type definitions (§8.7)
- Enum type definitions (§8.8)
- Delegate type definitions (§8.9)
- Exception type definitions (§8.10)
- Type extension definitions (§8.11)
- Measure type definitions (§9.4)

With the exception of type abbreviations and type extension definitions, type definitions define fresh, named types, distinct from other types.

Several type definitions or extensions can be introduced simultaneously using a *type definition group*, `type ... and ...`. For example

```
type RowVector(entries: seq<int>) =
    let entries = Seq.to_array entries
    member x.Length = entries.Length
    member x.Permute = ColumnVector(entries)

and ColumnVector(entries: seq<int>) =
    let entries = Seq.to_array entries
    member x.Length = entries.Length
    member x.Permute = RowVector(entries)
```

Exception type definitions and modules may not form part of a type definition group.

Most forms of type definitions may contain both *static* elements and *instance* elements. Static elements are accessed via the type definition. Scope constraints apply to elements of type definitions: at `static` bindings only the `static` elements are in scope. Most forms of type definitions may contain *members* (§8.12).

Type definitions are checked in a context where the type definition itself is in scope, as are all members and other accessible functionality of the type. This enables recursive references to the accessible static content of a type, and recursive references to the accessible properties of an object whose type is the same as the type definition or otherwise related to it.

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
```

In more detail, given an initial environment *env*, a type definition group is checked as follows:

- The number of generic arguments for each new type is determined.
- The basic kind of each new type definition is determined and checked, using *Type Kind Inference* if necessary (§8.1).
- The new type definitions are added to *env*.
- For type abbreviations, the type being abbreviated is determined.
- The base types, implemented interfaces, union cases, fields and abstract members of each new type definition are determined.
- The members for all new type definitions are collectively added to the environment as a recursive group (§8.12).
- The let-bindings and members of each new type definition are checked.
- The members for all new type definitions are collectively generalized (§14.7).

8.1 Type Kind Inference

A type specified using the `begin`/`end` on the right-hand-side of the '`=`' token, or with the `begin`/`end` implicit when the lightweight-syntax option is used, is called an anonymous type. In this case, the *kind* of the type is inferred. The following rules are applied to determine the kind of a type definition:

- If the type has a `ClassAttribute`, `InterfaceAttribute` or `StructAttribute` then this is used as the kind of the type.

- Otherwise if the type has any concrete elements then it is a class. Concrete elements are primary constructors, additional object constructors, `let`-bindings, non-`abstract` members or an `inherit` declaration with arguments.
- 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 then any use of the `ClassAttribute`, `InterfaceAttribute` or `StructAttribute` attributres must match the `class`/`end`, `interface`/`end` and `struct`/`end` tokens if given. These attributes may not be used with other kinds of type definitions such as type abbreviation, record, union or enum types.

8.2 Type Abbreviations

Type abbreviations define new names for existing types. For example:

```
type PairOfInt = int * int
```

Type abbreviations are processed during checking as part of a set of type declarations. 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.

Type abbreviations may not be hidden by signatures.

For example, if an implementation file contains

```
type MyInt = int
```

the signature file may not contain the following type with no abbreviation:

```
type MyInt
```

The process of repeatedly eliminating type abbreviations in favour of their equivalent types must not result in an infinite type derivation.

For example, the following are not a valid type definitions:

```
type X = option<X>

type Identity<'T> = 'T
and Y = Identity<Y>
```

Type abbreviations must have sufficient constraints to satisfy those constraints required by their right-hand-side.

For example, given the 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 may define additional constraints, so the following is permitted:

```
type F<'T when 'T :> IA and 'T :> IB> = C<'T>
```

Type abbreviations must use all declared type variables on their right-hand-side. For this purpose the order of type variables used on the right-hand-side of a type definition is determined by a left-to-right walk of the type.

For example,

```
type Drop<'T,'U> = 'T * 'T // invalid: dropped type variable
```

is not a valid type abbreviation.Note: This restriction ensure that F# type inference can forests of generalized type variables that are independent of the use of type abbreviations. This simplifies the process of guaranteeing a stable and consistent compilation to generic CLI code.

Flexible # types may not be used in type definitions except in member signatures. For example, flexible # types may not be used on the right of a type abbreviation, since 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
```

8.3 Record Types

A record type introduces a symmetric type where all inputs used to construct a value are available as properties on values of the type.

For example:

```
type R1 =
    { x : int;
      y: int }
    member this.Sum = this.x + this.y
```

The `with/end` tokens can be omitted when lightweight syntax is used as long as the `type-defn-elements` vertically aligns with the '`{`' in the `record-fields`, and likewise for union types. The semicolon '`;`' tokens can be omitted if the next *`record-short-field-spec`* vertically aligns with the previous.

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
```

Record types are implicitly sealed and may not be given the `SealedAttribute`. Record types may not be given the `AbstractClassAttribute`.

8.3.1 Members in Record Types

Union types may declare members (§8.12), overrides and interface implementations. Like all types with overrides and interface implementations, they are subject to *Dispatch Slot Checking* (§14.9).

8.3.2 Name Resolution and Record Field Labels

For a record type, the record field labels $field_1$ ... $field_N$ have module scope are added to the *FieldLabels* table of the current name resolution environment.

Record field labels play a special role in *Name Resolution for Members* (§14.1): an expression's type may be inferred from a record label, e.g.,

```
type R = { dx : int; dy: int }
let f x = x.dx // x is inferred to have type R
```

> Note: The module-scope of record field labels aids in writing succinct type-inferred implementations of basic data structures where records are used. However across larger pieces of software it can pollute the namespace of labels and give rise to unexpected resolutions.
>
> As a result a future version of F# will allow the use of the RequireQualifiedAccess attribute with record types to restrict the scope of record field labels.

8.3.3 Structural Hashing, Equality and Comparison for Record Types

Record types automatically implement the following interfaces and dispatch slots if they are not implemented by the record type author.

```
interface System.Collections.IStructuralEquatable
interface System.Collections.IStructuralComparable
interface System.IComparable
override GetHashCode : unit -> int
override Equals : obj -> bool
```

The implementations of the indicated interfaces and overrides are described later in this chapter.

8.4 Union Types

Type definitions with one or more union cases are called *union types*. For example:

```
type Message =
    | Result of string
    | Request of int * string
    member x.Name = match x with Result(nm) -> nm | Request(_,nm) -> nm
```

The `with/end` tokens can be omitted when lightweight syntax is used as long as the `type-defn-elements` vertically aligns with the first '`|`' in the `union-type-cases`. However this may not be done if the '|' tokens align with the 'type' token, e.g.

```
/// 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
```

Union case names must begin with an upper case letter, defined to mean any character where the CLI library function `System.Char.IsUpper` returns true, and `System.Char.IsLower` returns false. The discriminated union cases `Case1` ... `CaseN` have module scope and are added to the *ExprItems* and *PatItems* tables in the name resolution environment. This means they can be used both as data constructors and to form patterns.

Parentheses are significant in discriminated union definitions:

```
type CType = C of int * int
```

and

```
type CType = C of (int * int)
```

differ. The parentheses are used to indicate that the constructor takes on argument that is a first-class pair value.

The following declaration is considered to define a type abbreviation if the named type `A` exists in the name resolution environment, otherwise it is considered to define a union type.

```
type OneChoice = A
```

To disambiguate this case, use the following:

```
type OneChoice =
    | A
```

8.4.1 Members in Union Types

Union types may declare members (§8.12), overrides and interface implementations. Like all types with overrides and interface implementations, they are subject to *Dispatch Slot Checking* (§14.9) .

8.4.2 Structural Hashing, Equality and Comparison for Union Types

Union types automatically implement the following interfaces and dispatch slots if they are not implemented by the union type author.

```
interface System.Collections.IStructuralEquatable
interface System.Collections.IStructuralComparable
interface System.IComparable
override GetHashCode : unit -> int
override Equals : obj -> bool
```

The implementations of the indicated interfaces and overrides are described later in this chapter.

8.5 Class Types

Class types encapsulate values constructed via one or more object constructors. Class types have the form

```
type type-name pat_opt [as ident] _opt =
    class
        class-inherits-decl_opt
        class-let-bindings_opt
        type-defn-elements
    end
```

> The `class/end` tokens can be omitted when lightweight syntax is used, in which case Type Kind Inference (§8.1) is used to determine the kind of the type.

8.5.1 Primary Constructors in Classes

An *object constructor* represents a way of initializing an object. They can be used to create values of the type and to partially initialize an object from a subclass. Classes have object constructors through an optional *priimary 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 an *primary constructor*.

For example, the following type has an 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 with a primary constructor may, among other things, contain `let`-bindings and `let rec`-bindings.

The pattern for a primary constructor must be of the form `(simple-pat, ..., simple-pat)` for zero or more *simple-pat* where each is of the form:

```
simple-pat :=
    | ident
    | simple-pat : type
```

That is, 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 let-bindings should normally be added:

```
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 checked, the let-bindings of a primary constructor are checked and generalized in order.

When evaluated, a primary constructor evaluates the inheritance and let-bindings of the type in order.

8.5.1.1 Object References in Primary Constructors

For types with a primary constructor, the name of the `this` parameter can be bound throughout the let bindings and instance members of type definition as follows:

```
type X(a:int) as x =
     let mutable a = a
     let mutable b = a
     do x.A <- 3
     member self.A with get() =  x.A
                   and  set v =  a <- v
     member self.B =  4
```

During construction, the `this` value may not be dereferenced prior to the completion of the execution of the last `let` binding. Subsequent `do` bindings may access the `this` value.

8.5.1.2 If the variable is used anywhere prior to or including the last "let" binding then the use of this construct may result in a warning being given by the F# compiler indicating that a check will be inserted at runtime to ensure that the fields of the object are not accessed before initialization is complete. In this case, a NullReferenceException will be raised if the variable is dynamically dereferenced during the execution of the "let" and "do" bindings of the object.Inheritance Declarations in Primary Constructors

An `inherit` declaration specifies that a type extends the given type.

- If no `inherit` declaration is given for a class then the default is `System.Object`.
- The `inherit` declaration of a type must have arguments if and only if the type has a primary constructor.

8.5.1.3 "let" and "do" Declarations in Primary Constructors

Classes with primary constructors may include "let" and "do" bindings.

- Each let or do binding may be marked static (see below). If not marked static the binding is called an instance let or do binding.
- Instance let bindings are lexically scoped (and thus implicitly private) to the object being defined.
- let bindings for non-function values may be marked `mutable`.
- A group of let bindings may be marked `rec`.
- Let bindings are generalized
- Let bindings in classes may not have attributes
- The compiled representation used for let-bounds values in classes is implementation dependent. Non-function let bindings are generally represented as instance fields in the corresponding class. However there is no guarantee of this.

8.5.1.4 Static "let" and "do" Declarations in Primary Constructors

Classes with primary constructors may have "let" and "do" bindings marked as "static":

- Static `let` bindings are lexically scoped (and thus implicitly private) to the type being defined.
- Each let may be marked `mutable`.
- A group of let bindings may be marked `rec`.
- Static let bindings are generalized
- Static let bindings in classes may not have attributes
- The compiled representation used for static let-bounds values in classes is implementation dependent. Static non-function let bindings are generally represented as static fields in the generated class. However there is no guarantee of this.
- Static let bindings are computed once per-generic-instantiation.

- Static let-bindings 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 §13.1.1.

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)
```

8.5.2 Members in Classes

Class types may declare members (§8.12), overrides and interface implementations. Like all types with overrides and interface implementations, they are subject to *Dispatch Slot Checking* (§14.9) .

8.5.3 Additional Object Constructors in Classes

While the use of primary object constructors is generally preferred, additional object constructors may also be specified. These are needed

- to define classes with more than one constructor; or
- if explicit `val` fields are specified without the `DefaultValueAttribute` .

For example, this example adds a second constructor to a class with 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, then 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, then additional constructors must initialize those `val` fields of the object that do not have the `DefaultValueAttribute`, and also specify a call to a base class constructor for any inherited class type. No call to a base class constructor is required if the base class is `System.Object`.

The use of additional object constructors and `val` fields is particularly required when multiple object constructors exist that must each call different base class constructors. For example:

```
type BaseClass =
    val s1 : string
    new (s) = { s1 = s }
    new () = { s = "default" }

type SubClass =
    inherit BaseClass
    val s2 : string
    new (s1,s2) = { inherit BaseClass(s1); s2 = s2 }
    new (s2) = { inherit BaseClass(); s2 = s2 }
```

Additional object constructors are implemented using a restricted subset of expressions that ensure 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"
```

For classes without a primary constructor, the name of the `this` 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 will be given if `x` occurs syntactically in or before the *additional-constr-init-expr* of the construction expression. Any evaluation of a reference to this variable prior to the completion of execution of the *additional-constr-init-expr* within the *additional-constr-expr* throws a `NullReferenceException`.

8.5.4 Additional Fields in Classes

Additional field members indicate a value stored in an object. They are generally used only for classes without a primary constructor, or for mutable fields with default initialization, and often only occur in generated code. For example:

```
type PairOfIntegers =
    val x : int
    val y : int
    new(x,y) = {x = x; y = y}
```

And a static field in an explicit class type:

```
type TypeWithADefaultMutableBooleanField =
    [<DefaultValue>]
    static val mutable ready : bool
```

A `val` specifications in a type with a primary constructor must be marked mutable and have the `DefaultValueAttribute`, e.g.

```
type X() =
   [<DefaultValue>]
   val mutable x : int
```

This attribute takes a parameter `check` that indicates if checking should be applied to ensure that unexpected null values are not created by this mechanism. This parameter defaults to `true`. If this parameter is true, the type of the field must admit default initialization (§5.3.8).

For example, the following type is rejected:

```
type MyClass<'T>() =
    [<DefaultValue>]
    static val mutable uninitialized : 'T
```

However, in compiler generated and hand-optimized code it is sometimes essential to be able to emit fields that are completely uninitialized, e.g.

```
type MyNullable<'T>() =
    [<DefaultValue>]
    static val mutable ready : bool

    [<DefaultValue(false)>]
    static val mutable uninitialized : 'T
```

At runtime, such a field is initially assigned the zero value for their type (§6.10.1). 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
```

8.6 Interface Types

Interface type definitions are hierarchically arranged types implemented by objects. They are made up only of 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)
```

The `interface/end` tokens can be omitted when lightweight syntax is used, in which case Type Kind Inference (§8.1) 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, e.g. 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. There may be no circular references between `inherit` declarations based on the named types of the inherited interface types.

8.7 Struct Types

Structs are type definitions whose instances are stored inline inside the stack frame or object of which they are a part. They are represented a CLI struct, 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
```

Structs undergo type kind inference (§8.1), so 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 with 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)
```

Structs fields may be mutable. However, this is only possible for structs without 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. Like all types with overrides and interface implementations, they are subject to *Dispatch Slot Checking* (§14.9) .

Structs may not have `inherit` declarations.

Structs may not have `let` or `do` bindings. For example, the following is not valid

```
[<Struct>]
type S (def : int) =
  do System.Console.WriteLine("Structs cannot use 'do'!")
```

This is because the default constructor for structs would not execute these bindings.

Structs may have `static let` or `static do` bindings. For example, the following is valid

```
[<Struct>]
type S (def : int) =
    static do System.Console.WriteLine("Structs can use 'static do'")
```

A struct type must be realizable according to the CLI rules for structs, in particular recursively constructed structs are not permitted unless reference-typed indirections are used . For example, the following type definition is not permitted, because the size of `S` would be infinite

```
[<Struct>]
type BadStruct1 =
    val data: float;
    val rest: BadStruct1
    new (data, rest) = { data = data; rest = rest }
```

Likewise, the implied size of the following struct would be infinite:

```
[<Struct>]
type BadStruct2 (data: float, rest: BadStruct2) =
  member s.Data = data
  member s.Rest = rest
```

Some struct types have an *implicit default constructor* which initializes all fields to the default value. This applies to struct types whose fields all have types which admit default initialization. For example, the Complex type defined above admits default initialization.

```
let zero = Complex()
```

> Note: The existence of the implicit default constructor for structs is not recorded in CLI metadata and is an artefact for the CLI specification and implementation itself. A CLI implementation admits default constructors for all struct types, though F# does not permit their direct use for F# struct types unless all field types admit default initialization. The is similar to the way that F# considers some types to have null as an abnormal value.
>
> Public struct types for use from other .NET languages should be authored with the existence of the default zero-initializing constructor in mind.

8.8 Enum Types

Occasionally the need arises to represent a type that compiles as a CLI enumeration. This is done by giving integer constants in a type definition that otherwise has the same form as a discriminated union type. 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"
```

Each case must be given a constant value of the same type. The constant values dictate the *underlying type* of the enum must be one of the following types:

- `sbyte`, `byte`, `int16`, `int32`, `uint64`, `uint16`, `uint32`, `char`

The declaration of an enumeration type in an implementation file has the following effects on the typing environment:

- it brings a named type into scope
- it 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>` for their underlying type.

The 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
```

> Rationale: This is standard name scoping rules for CLI enumerations

Unlike discriminated 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 not in the above enumeration can be generated by using the `enum` function from the F# library:

```
let unknownColor : Color = enum<Color>(7)
```

Rationale: This is behaviour for CLI enumerations

8.9 Delegate Types

Occasionally the need arises to represent a type that compiles as a CLI delegate type. This is done by using the `delegate` keyword with a member signature. For example:

```
type Handler<'T> = delegate of obj * 'T -> unit
```

Delegates are often used when interfacing to CLI libraries via P/Invoke:

```
type ControlEventHandler = delegate of int -> bool

[<DllImport("kernel32.dll")>]
extern void SetConsoleCtrlHandler(ControlEventHandler callback, bool add)
```

8.10 Exception Definitions

Exception definitions in modules define a new way of constructing values of type `exn` (an abbreviation for `System.Exception`). Exception definitions define new discriminated union cases that generate and pattern match on values of type `exn` (an abbreviation for `System.Exception`).

Note: Exception values may also be generated by defining and using classes that extend `System.Exception`.

For example:

```
exception Error of int * string
```

The discriminated union case can now be used to generate values of type `exn`:

```
raise (Error(3, "well that didn't work did it"))
```

The discriminated union case can now be used to pattern match on values of type `exn`:

```
try
    raise (Error(3, "well that didn't work did it"))
with
    | Error(sev, msg) -> printfn "severity = %d, message = %s" sev msg
```

Exception definitions may also 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))
```

An exception definition also generates a type with name *`idException`*.

8.11 Type Extensions

Type extensions associate additional dot-notation members with an existing type. For example:

```
// intrinsic extension
type System.String with
    member x.IsLong = (x.Length > 1000)
```

The `end` token to match `with` is optional when lightweight syntax is used.

Type extensions may be given for any accessible type definition except those defined by type abbreviations.

If the type being extended is in the same module or namespace declaration group as the type definition for the type then it is called an *intrinsic extension*.

Otherwise the type extension must be in a module, and it is called an *optional extension*. Opening a module containing an optional extension extends the name resolution of the "." syntax for the type that is extended. That is, optional extensions are only accessible if the module containing the extension has been opened.

Name resolution for members defined in type extensions behaves as follows:

- Members defined in intrinsic extensions follow the same name resolution and other language rules as members defined as part of the original definition of the type.
- Members defined in optional extensions are considered last in the name resolution process. If a member defined in an optional extension has the same name as any member in the original type definition, including inherited members, then it will be inaccessible.

For example, the following illustrates the definition of one intrinsic and one optional extension 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 =

      // optional extension
      type Numbers.Complex with
          member x.Magnitude = ...
          member x.Phase = ...
```

Extensions may define both instance members and static members.

Extensions are checked as follows

- The bindings in an extension are checked along with the members in the group of type definitions of which it is a part
- No two intrinsic extensions may contain conflicting members. This is implicit in the fact that 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.
- Optional extensions must be in modules
- Optional extension members are syntactic sugar for static members. Uses of optional extension members elaborate to calls to static members with encoded names where the object is passed as the

first argument. The encoding of names is not specified in this release of F# and is not necessarily compatible with C# encodings of C# extension members

8.12 Members

Member definitions describe functions 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 any particular object. 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
```

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 x.StaticProperty2
        with get() = 3 + staticArray.Length

    static member x.MutableStaticProperty
        with get()         = adjustableStaticValue
        and  set(v:string) = adjustableStaticValue <- v

    static member x.StaticIndexer
        with get(idx) = staticArray.[idx]

    static member x.StaticIndexer2
        with get(idx1,idx2) = staticArray2.[idx1].[idx2]

    static member x.MutableStaticIndexer
        with get (idx1) = staticArray.[idx1]
        and  set (idx1) (v:string) = staticArray.[idx1] <- v
```

Members across a set of mutually type definitions are checked as a mutually recursive group. As with collections of recursive functions, inconsistent type constraints may arise from recursive calls to potentially-generic methods:

```
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 full type annotation on a target method makes it eligible for early generalization (§14.7).

```
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)
```

As with collections of recursive functions, a mutually recursive set of members shares the same *floating type variable environment*. For example:

```
type Test() =
   member t.M1 (x: 'T) = ()

   member t.M2 (y: 'T list) =  t.M1(y) // this gives rise to an error
```

Here the two floating type variable references to `'T` refer to the “same” `'T`. This gives rise to the inconsistent type constraint `'T = 'T list`.

8.12.1 Property Members

A *property member* is a *`member-binding`* of 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 [1]
```

A property member of the form

```
static_opt member ident._opt ident with get pat_1 = expr_1 and set pat_2a [pat_2b] = expr_2
```

is equivalent to two property members of the form:

```
static_opt member ident._opt ident with get pat_1 = expr_1
static_opt member ident._opt ident with set pat_2a [pat_2b] = expr_2
```

Furthermore

```
static_opt member ident._opt ident = expr
```

is equivalent to:

```
static_opt member ident._opt ident with get () = expr
```

Also

```
static_opt member ident._opt ident with set pat = expr_2
```

is equivalent to:

```
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 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 taking one argument is accessed using `x.[i]`, and when taking two arguments via `x.[i,j]` etc.Setter properties must return type `unit`.

Property members may be declared abstract. If a property has both a getter and a setter, then either both must be abstract or neither.

Each property member has an implied property type given by the type of the value returned by a getter property or the type of the value accepted by a setter property. If a property has both a getter and a setter, and neither is an indexer, then the property type implied by the signature of each property must be identical.

> Note: C# permits the .[] to be given an underlying name other than “Item”. Given the current specification, F# does not.

Static property members are evaluated every time the member is invoked. For example, given

```
type C () =
    static member Time = System.DateTime.Now
```

The body of the member is evaluated each time `C.Time` is evaluated.

---

[1] The ordering of the setter and the getter does not matter – this is equally valid:
`[static] member ident._opt ident with set pat_2a [pat_2b] = expr_2 and get pat_1 = expr_1`

Note: a static property member may also be written with an explicit `get` method:

```
static member ComputerName
    with get() = System.Environment.GetEnvironmentVariable("COMPUTERNAME")
```

8.12.2 Method Members

A *method member* is a *member-binding* of the form:

$static_{opt}$ member $ident._{opt}$ $ident$ pat_1 ... pat_n = $expr$

The $ident._{opt}$ must be present if and only if the property member is an instance member. The identifier *ident* is bound to the "this" (or "self") variable associated with the object within the expression *expr*.

Arity analysis (§14.11) applies to method members. This is because F# members must compile to CLI methods and be compatible with the expected compiled form of CLI methods, which accept only a single fixed collection of arguments.

8.12.3 Curried Method Members

Methods may be written taking multiple arguments in iterated ("curried") form. For example:

```
static member StaticMethod2 s1 s2 =
    sprintf "In StaticMethod(%s,%s)" s1 s2
```

The rules of arity analysis (§14.11) determine the compiled form of these members.

8.12.4 Named Arguments to Method Members

Calls to methods (but not 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 associated with a method declaration are derived from the names used in the first pattern of a member binding, 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 only be used with the arguments that correspond to the arity of the member. That is, because members only have an arity 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 using named arguments is specified in *Method Application Resolution* (see §14.4). The rules in that section describe how, during resolution, named arguments are associated with matching formal arguments of the same name of “settable” return properties of the same name. For example, the following code:

```
System.Windows.Forms.Form(Text="Hello World")
```

resolves the named argument to a settable property. If an ambiguity exists, assigning the named argument to a formal argument takes precedence over settable return properties.

The *Method Application Resolution* (§14.4) rules ensure that:

- Named arguments must appear after all other arguments, including optional arguments matched by position

For example, the following code is invalid:

```
// error: unnamed args after named

System.Console.WriteLine(arg0="World","Hello {0}")
```

After named arguments have been assigned, the remaining required arguments are called the *required unnamed arguments*. These must be a prefix of the original required arguments.

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 names of members may be listed in signatures and on the signatures used for abstract members, e.g.

```
type C() =
    static member ThreeArgs : arg1:int * arg2:int * arg3:int -> int
    abstract TwoArgs : arg1:int * arg2:int -> int
    default x.TwoArgs(arg1,arg2) = arg1 + arg2
```

8.12.5 Optional Arguments to Method Members

Members (but not `let`-declared functions) may have optional arguments. These must come at the end of the argument list. An optional argument is marked with a `?` before its name. Inside the member the argument has type `option<argType>`. On the callside the argument is provided with a value of type *`argType`*, though there is a way to pass a value of type `option<argType>` if necessary. For example:

```
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 using the following techniques:

- By name, e.g. `arg2=1`
- By propagating an existing optional value by name, e.g. `?arg2=None` or `?arg2=Some(3)` or `?arg2=arg2`.

  Note: This can be useful when building one method that passes numerous optional arguments on to another.

- By using normal, unnamed arguments 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 using optional arguments is specified in *Method Application Resolution* (see §14.4).

Optional arguments may not be used in member constraints.

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, e.g. `?ValueTitle = Some (…)`.

For example, below is a fragment of a call to a Microsoft Excel COM automation API using 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")
```

Note that CLI optional arguments are not passed as values of type `Option<_>`. If the optional argument is present, its value is passed. If it is not, 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 otherwise for other CLI optional arguments with no default value.

The compiled representation of optional arguments is fragile, in the sense that the addition of further optional arguments to a member signature will result in a compiled form that is not binary compatible with the previous compiled form.

8.12.6 Overloading of Members

Multiple methods of the same name may appear in the same type definition or extension. For example

```
type MyForm() as self =
    inherit System.Windows.Forms.Form()

    member x.ChangeText(text: string) =
        self.Text <- text

    member x.ChangeText(text: string, reason: string) =
        self.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 must be uniquely identified by name and number of arguments, or else annotated with an `Microsoft.FSharp.Core.OverloadIDAttribute`. Adding an `OverloadID` attribute to a member permits it to be part of a group overloaded by the same name and arity.

The string argument to the attribute must be a unique name amongst those in the overload set. The `OverloadID` must be specified in both signature and implementation files if a signature file is provided for the definition.

```
type MyForm() as self =
    inherit System.Windows.Forms.Form()

    [<OverloadID("1")>]
    member x.ChangeText(text: string) = self.Text <- text

    [<OverloadID("2")>]
    member x. ChangeText(i: int) = self.Text <- string i
```

8.12.7 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, ignoring return types. This does not apply to members named `op_Implicit` or `op_Explicit`.

8.12.8 Conditional Compilation of Member Calls

CLI languages respect the presence of the `System.Diagnostics.ConditionalAttribute` as follows:

- The `ConditionalAttribute("symbol")` may only be applied to methods
- Methods with the `ConditionalAttribute` must have return type `unit` . This may be checked on either use of the method or the definition of the method.
- If *symbol* is not in the current set of conditional compilation symbols, then Application Expressions that get resolved to calls to members with the `ConditionalAttribute` are eliminated by the compiler as follows, ensuring arguments are not evaluated.
    - Static members: `Type.M(args)` → `()`
    - Instance members: `expr.M(args)` → `()`
- First class uses are eliminated as follows:
    - Static members: `Type.M(args)` → `(fun args -> ())`
    - Instance members: `expr.M` → `let _ = expr in (fun args -> ())`

Note: In theory, conditional compilation of member calls also applies to calls that arise from desugaring syntax, e.g. computation expressions. However these almost never have return type “unit” so this is rarely used in practice.

8.12.9 Members Represented as Events

Events are the CLI notion of a “listening point”, that is, a configurable object holding 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, i.e., are objects that mediate the addition and removal of listeners from a backing list of listeners. The F# library supports a type `Microsoft.FSharp.Control.IEvent<_,_>` and a module `Microsoft.FSharp.Control. Event` that contains operations for mapping, folding, creating and composing events. The definition of the type is 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>
```

A sample use of events is shown below:

```
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 -> Printf.printf "OnRedraw\n")
form.Activate()
Application.Run(form)
```

Events from CLI languages are revealed as object properties of type `Microsoft.FSharp.Control.IEvent<_,_>`. The type arguments to this type `IEvent` are determined by the F# compiler.

Event declarations are not built-in to the F# language, and `event` is not a keyword. However, property members marked with the `CLIEventAttribute` and whose type coerces to `Microsoft.FSharp.Control.IDelegateEvent<_>` are compiled to include extra CLI metadata and methods that mark the property name as a CLI event.

For example, in the following the “ChannelChanged” property is currently compiled as a .NET 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 member ChannelChanged : IEvent<ChannelChanged,int>


type ImplI() =
    let channelChanged = new Event<ChannelChanged,_>()
    interface I with
        [<CLIEvent>]
        member self.ChannelChanged = channelChanged.Publish
```

8.12.10 Members Represented as Static Members

Most members are represented as their corresponding CLI method or property.

In certain situations even instance members may be compiled as static methods. This happens

- when the type definitions use `null` as a representation (REF), as given by placing flag on a `CompilationRepresentation(CompilationRepresentationFlags.UseNullAsTrueValue)` on the type.
- when the member is an optional extension member (REF)

This 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.UseNullAsTrueValue)>]
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.

8.13 Abstract Members and Interface Implementations

8.13.1 Abstract Members

An *abstract member* in a type represents a promise that an object will provide an implementation for a dispatch slot. For example:

```
type IX =
   abstract M : int -> int
```

A class definition may contain abstract members, but must be labelled with the `AbstractClass` attribute:

```
[<AbstractClass>]
type X() =
   abstract M : int -> int
```

An abstract member has the form

```
abstract ident typar-defns_opt

      : args-spec_1 -> ... -> args-spec_n -> type [ with get | set | get,set ]_opt
```

For n >=2 then $args\text{-}spec_2 ... args\text{-}spec_n$ must all be patterns without attribute or optional argument specifications.

If `get` or `set` is specified then the abstract member is a *property member*, and if both are specified then it is equivalent to two abstract members, one with `get` and one with `set`.

8.13.2 Members Implementing Abstract Members

An *implementation member* is of the form:

```
override ident.ident pat1 ... patn = expr
default ident.ident pat1 ... patn = expr
```

Implementation members implement dispatch slots. Members that implement dispatch slots on the base type must be declared using `override` or `default`, and those that implement dispatch slots on interface types must be declared under an interface implementation and may use `member`, `override` or `default`.. The keywords `override` or `default` are synonyms.

> The combination of an `abstract` slot declaration and a default implementation of that slot is the equivalent of a "virtual" method in some other languages.

For example:

```
type IX =
    abstract P : int

type X() =
    override x.ToString() = "I'm an X"
    interface IX with
        member x.P = 3+4
```

> You should use `default` for the original declaration of the abstract member, and `override` for implementations in subclasses. This not currently checked but may be checked in a later version of the language.

Property members override corresponding getter or setter dispatch slots.

```
type X() =
    abstract P : int with get
    default x.P with get() = 3
    member x.P with set(v) = failwith "Ha! P is not really settable"

type Y() =
    inherit X()
    override x.P with get() = 4
```

8.13.3 Interface Implementations

An *interface implementation* specifies how objects of the given type support the given interface. An interface in a type definition indicates that objects of the given type support the given 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
```

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
```

All interface implementations are made explicit. That is, interfaces are not implicitly implemented by other method declarations nor by inherited method declarations.

Each member of an interface implementation is checked as follows:

- The member must be an instance member definition
- *Dispatch Slot Inference* (§14.8) is applied
- The member is checked under the assumption that the "this" variable has the enclosing type.

For example, 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
```

8.14 Generated Equality, Hashing and Comparison

Functional programming in F# frequently involved the use of structural equality, hashing and comparison. For example:

```
(1,1+1) = (1,2)
```

evaluates to `true`, because tuple types support structural equality. Likewise these two function calls return identical values:

```
hash (1,1+1)
hash (1,2)
```

Likewise 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 fields 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)
```

In order to facilitate this, by default, record, union, struct and exception type definitions implicitly include compiler-generated declarations for structural equality, hashing and comparison. These implicit declarations are divided into those for structural equality/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) = ...
```

and those for 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) = ...
```

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.

Note: The .NET interface System.Collections.IStructuralEquatable and the System.Object overrides Equals and GetHashCode allow the specification of two different equality relations, called the ER (equality-relation) and PER (partial-equality-relation). These nearly always coincide except if the relations involve comparing floating point values, where the special value NaN has unusual semantics:

ER = equality relation, so (NaN = NaN) = true

PER = partial equality relation (NaN = NaN) = false

For he vast majority of F# code this distinction is not relevant. However, should you manually implement equality for structural types which must themselves have equality semantics, you should preserve these relations, e.g.

> Object.Equals(object) should implement ER semantics

> IStructuralEquatable.Equals(AnyComparerWhichImplementsERSemantics) should structural ER semantics

> IStructuralEquatable.Equals(AnyComparerWhichImplementsPERSemantics) should structural PER semantics

---

8.14.1 Controlling the Generation of Structural Equality, Hashing and Comparison Implementations

The inclusion of structural equality, hashing and comparison declarations can be controlled by the use of the following attributes

```
Microsoft.FSharp.Core.ReferenceEquality Microsoft.FSharp.Core.StructuralEquality
Microsoft.FSharp.Core.StructuralComparison
```

For example, given

```
type R1 =
    { myData : int }
    static member Create() = { myData = 0 }

[<ReferenceEquality>]
type R2 =
    { mutable myState : int }
    static member Fresh() = { myState = 0 }

[<StructuralEquality(true); StructuralComparison(false) >]
type R3 =
    { someType : System.Type }
    static member Make() = { someType = typeof<int> }
```

then

```
R1.Create() = R1.Create()
not (R2.Fresh() = R2.Fresh())
R3.Make() = R3.Make()
```

all evaluate to `true`. If present, these attributes may only be used in the following combinations:

```
[<ReferenceEquality(true)>]

[<StructuralEquality(false); StructuralComparison(false)>]

[<StructuralEquality(true); StructuralComparison(false)>]

[<StructuralEquality(true); StructuralComparison(true)>]
```

Note: the value carried by these attributes defaults to true if the attribute is used, hence for the purposes of this specification

```
[<ReferenceEquality>]
```

is considered equivalent to

```
[<ReferenceEquality(true)>]
```

Note: `[<StructuralEquality(false); StructuralComparison(false)>]` is used when writing customizing hash/eq implementations, or if no custom implementations are given then reference equality is inherited from the System.Object defaults.

`[<StructuralEquality(true); StructuralComparison(false)>]` is used when writing customizing IComparable implementations, or if no custom implementations are given then reference equality is assumed

Note: the last of these is equivalent to having none of these attributes at all, and just serves as a comment that the type has structural equality/hash/compare semantics.

Note: A future revision of F# may give a warning if structural comparison is specified or auto-generated for types that contain fields where `compare` is known to fail for all non-null values of the given field type.

Note: A future revision of F# may allow you to put attributes on fields that indicate they should be ignored for the purposes of structural eq/hash/compare generation.

Note that

```
R3. Make() < R3. Make()
```

will raise an exception at runtime, since the type `R2` doesn't support structural comparison. However, a future revision of F# may give a warning if structural comparison operators are used in conjunction with types where `compare` is known to fail for all non-null values of the given type.

For a given type, the structural equality and hashing declarations are generated if:

- The `ReferenceEquality(true)` attribute has not been specified in the attributes of the type.
- The corresponding member or interface is not given an explicit implementation in the type.

For a given type, the structural comparison declarations are generated if:

- The `ReferenceEquality(true)` attribute has not been specified in the attributes of the type.
- The `StructuralComparison(false)` attribute has not been specified in the attributes of the type.
- The corresponding member or interface is not given an explicit implementation in the type.

The above attributes may not be used on class, interface, delegate or enum type definitions.

8.14.2 Behaviour of the generated Object.Equals implementation

For a type `T`, the behaviour of the generated `override x.Equals(y:obj) = ...` declaration is as follows. First, if an explicit implementation of interface `System.IComparable` has been given then just call `System.IComparable.CompareTo`:

```
override x.Equals(y:obj) =
    ((x :> System.IComparable).CompareTo(y) = 0)
```

Otherwise,

- The `y` argument is converted to type `T`.
- If `T` is a reference type, and `y` is null, then `false` is returned

- When `T` is a struct or record type, `Microsoft.FSharp.Core.Operators.(=)` is invoked on each corresponding pair of fields of `x` and `y` in declaration order, stopping at the first `false` result and returning `false`.
- When `T` is a union type, `Microsoft.FSharp.Core.Operators.(=)` is invoked first on the index of the discriminated union cases for the two values, then on each corresponding field pair of `x` and `y` for the data carried by the union case, stopping at the first `false` result and returning `false`.
- When `T` is an exception type, `Microsoft.FSharp.Core.Operators.compare` is invoked on the index of the tags for the two values, then on each corresponding field pair for the data carried by the exception stopping at the first `false` result and returning `false`.

The first lines of this code can be written:

```
override x.Equals(y:obj) =
      let y = (y :?> T) in
      match y with
      | :? null -> 1
      | _ -> ...
```

8.14.3 Behaviour of the generated CompareTo implementations

For a type `T`, the behaviour of the generated `System.IComparable.CompareTo` function is as follows:

- The `y` argument is converted to type `T`
- If `T` is a reference type, and `y` is null, then `1` is returned
- When `T` is a struct or record type, `Microsoft.FSharp.Core.Operators.compare` is invoked on each corresponding pair of fields of `x` and `y` in declaration order, returning the first non-zero result.
- When `T` is a union type, `Microsoft.FSharp.Core.Operators.compare` is invoked first on the index of the discriminated union cases for the two values, then on each corresponding field pair of `x` and `y` for the data carried by the union case, returning the first non-zero result.
- When `T` is an exception type, `Microsoft.FSharp.Core.Operators.compare` is invoked on the index of the tags for the two values, then on each corresponding field pair for the data carried by the exception, returning the first non-zero result.

Note: The first lines of this code can be written:

```
interface System.IComparable with
    member x.CompareTo(y:obj) =
        let y = (y :?> T) in
        match y with
        | :? null -> 1
        | _ -> ...
```

8.14.4 Legacy generation of Object.Equals implementation

For backwards compatibility with earlier versions of F#, class types that provide an implementation of the `System.IComparable` interface and no implementation of `override x.Equals(y:obj) = ...` are automatically provided with a generated declaration of the following form:

```
override x.Equals(y:obj) =
    ((x :> System.IComparable).CompareTo(y) = 0)
```

Note: A deprecation warning is given when this occurs

8.14.5 Behaviour of the generated GetHashCode implementations

TBD: describe the behaviour of these implementations

8.14.6 Behaviour of hash, (=) and compare

The generated equality, hashing and comparison declarations described above make use of the F# library functions hash, (=) and compare. The behaviour of these library functions is defined by the pseudo code below. This code ensures

- Ordinal comparison for strings
- Structural comparison for arrays
- Natural ordering for native integers (which do not support System.IComparable)

8.14.6.1 Pseudo code for Microsoft.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 byte arrays and obj[].

```
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(...))
```

8.14.6.2 Pseudo code for Microsoft.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 byte arrays and obj[].

```
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)
```

8.14.6.3 Pseudo code for Microsoft.FSharp.Core.Operators.hash

Note: TBD

9 Units Of Measure

F# supports static checking of *units of measure*. Units of measure, or *measure*s 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, they differ from types in that they play no role at run-time (in fact, they are *erased*), they obey special rules of *equivalence* (so `N m` can be interchanged with `m N`) and are supported by special syntax.

Measures are supported through the use of a special attribute `Measure` on types (to introduce base units-of-measure, or derived units-of-measure) and on type parameters (to parameterize types and members on units-of-measure). The primitive types `int8`, `int16`, `int32`, `int64`, `float`, `float32` and `decimal` have non-parameterized (dimensionless) and parameterized versions. Here is a simple example:

```
[<Measure>] type m                          // base measure: metres
[<Measure>] type s                          // base measure: seconds
[<Measure>] type sqm = m^2                  // derived measure: square metres
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
```

Furthermore, as with ordinary types, F# can *infer* the types of functions that are generic in their units. For example, here is an interactive session:

```
> let sqr (x:float<_>) = x*x;; // Need to say float<_> to resolve overloading

val sqr : float<'U> -> float<'U ^ 2>

> let sumsqr x y = sqr x + sqr y;;

val sumsqr : float<'U> -> float<'U> -> float<'U ^ 2>
```

Measure expressions have a special syntax that includes the use of `*` and `/` for product and quotient of measures, juxtaposition as shorthand for product, and `^` for integer powers. The syntax `1` denotes "dimensionless" or "without units", but is rarely needed, as non-parameterized types such as `float` are aliases for the parameterized type with `1` as parameter, i.e. `float = float<1>`.

9.1 Measure Constants

The syntax of constants (Section 4.3) is extended to support floating-point constants with units-of-measure.

```
measure-literal-atom :=
  | long-ident                              -- named measure e.g. kg
  | ( measure-literal-simp )                -- parenthesized measure e.g. (N m)

measure-literal-power :=
  | measure-literal-atom
  | measure-literal-atom ^ int32            -- power of measure e.g. m^3

measure-literal-seq :=
  | measure-literal-power
  | measure-literal-power measure-literal-seq

measure-literal-simp :=
  | measure-literal-seq                     -- implicit product e.g. m s^-2
  | measure-literal-simp * measure-literal-simp      -- product e.g. m * s^3
  | measure-literal-simp / measure-literal-simp      -- quotient e.g. m/s^2
  | / measure-literal-simp                  -- reciprocal e.g. /s
  | 1                                       -- dimensionless

measure-literal :=
  | _                                       -- anonymous measure
  | measure-literal-simp                    -- simple measure e.g. 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
```

Atomic literal measures are long identifiers such as `SI.kg` or `MyUnits.feet`. Arbitrary literal measures are built from atomic measures using juxtaposition or `*` for product of measures, `/` for quotient, and `^` for integer powers. As with floating point expressions, `*` and `/` have the same precedence, and associate to the left, but juxtaposition binds tighter than both, and `^` binds tighter still. The symbol `/` can also be used as a unary reciprocal operator. The measure `1` denotes “no units-of-measure” and is used for dimensionless quantities.

Floating point constants can be annotated with their measure by following the constant by a literal measure in angle brackets. The special measure syntax `_` can be used when the measure can be inferred by the compiler from the context. It is particularly useful is reserved for the constant zero, which can be assigned an arbitrary measure, including one involving unit parameters.

Here are some examples:

```
let earthGravity = 9.81f<m/s^2>
let atmosphere = 101325.0<N m^-2>
let zero = 0.0f<_>
```

Constants with units-of-measure are assigned a `float` or `float32` type with a measure parameter as specified in the suffix. 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>`.

9.2 Measure Type Annotations

The syntax of types is extended to support syntax for units-of-measure. This is the same as for measure annotations on constants, except that measure variables are supported, which have the same syntax as ordinary type variables.

```
measure-atom :=
  | typar                                  -- variable measure e.g 'U
  | long-ident                             -- named measure e.g. kg
  | ( measure-simp )                       -- parenthesized measure e.g. (N m)

measure-power :=
  | measure-atom
  | measure-atom ^ int32                   -- power of measure e.g. m^3

measure-seq :=
  | measure-power
  | measure-power measure-seq

measure-simp :=
  | measure-seq                            -- implicit product e.g. 'U 'V^3
  | measure-simp * measure-simp            -- product e.g. 'U * 'V
  | measure-simp / measure-simp            -- quotient e.g. 'U / 'V
  | / measure-simp                         -- reciprocal e.g. /'U
  | 1                                      -- dimensionless measure (no units)

measure :=
  | _                                      -- anonymous measure
  | measure-simp                           -- simple measure e.g. 'U 'V

type :=
  | ...
  | measure
```

9.3 Equivalence of measures and constraint solving

Internally, the F# type-checker maintains measures in the following form:

```
measure-int := 1 | long-ident | typar | measure-int measure-int | / measure-int
```

Powers of measures are expanded, so for example `kg^3` is equivalent to `kg kg kg`.

In contrast to ordinary types, syntactically distinct measures may be treated by the type-checker as equivalent. For example, `kg m / s^2` is the same as `m kg / s^2`. To be precise: the type-checker does not distinguish between measures that 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, e.g. `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 there is in scope a measure abbreviation of the form `[<Measure>] type` *long-ident* `=` *measure*

(Note: These are the laws of abelian groups together with expansion of abbreviations.)

For presentation purposes (in error messages, and in Visual Studio), measures are presented in a normalized form, as specified by the *measure* grammar above, 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 of checking equality of measures: given two measure expressions *measure-int*$_1$ and *measure-int2*, reduce each to normalized form using the rules of commutativity, associativity, identity, inverses and abbreviation, and then compare the syntax.

9.3.1 Constraint solving

The mechanism described in Section 14.6 is extended to support equational constraints between measure expressions. These arise from equations between parameterized types, i.e. when *type*`<`*tyarg*$_{11}$`,...,` *tyarg*$_{1n}$`>` `=` *type*`<`*tyarg*$_{21}$`,...,` *tyarg*$_{2n}$`>` is reduced to a series of constraints *tyarg*$_{1i}$ `=` *tyarg*$_{2i}$. For those arguments that are measures, rather than types, the rules listed above 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, then a type error is produced. For example, the expression `fun (x:float<m^2>,y:float<s>) -> x+y` would (eventually) result in a constraint `m^2 = s`, which cannot be solved, indicating a type error.

9.3.2 Generalization

Analogous to the process of generalization of type variables described in Section 14.7, there is a generalization procedure that produces measure variables over which a type can be generalized.

9.4 Measure Definitions

Measure definitions define new named units-of-measure.

```
measure-defn :=
  | prim-measure-defn                    -- primitive measure definition
  | abbrev-measure-defn                  -- measure abbreviation

prim-measure-defn := [<Measure>] type-name
abbrev-measure-defn := [<Measure>] type-name = measure
```

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, distinct from other measures. Measure abbreviations, like type abbreviations, define new names for existing measures. Also like type abbreviations, repeatedly eliminating measure abbreviations in favour of their equivalent measures must not result in infinite measure expressions. For example, the following is not a valid measure definition:

```
[<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 Section 5.2). If an explicit parameter definition is used, the parameter name is prefixed by the special attribute `[<Measure>]`. 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 is used when checking the actual arguments to types and other parameterized definitions, rejecting ill-formed types such as `float<int>` and `IEnumerable<m/s>`.

The built-in unparameterized floating-point and decimal types can be considered as abbreviations for the built-in parameterized floating-point types:

```
type float = float<1>
type float32 = float32<1>
type decimal = decimal<1>
```

9.6 Measure Parameter Erasure

In contrast to *type* parameters on generic types, *measure* parameters are not exposed in the metadata that is interpreted by the runtime: they are *erased*. This has a number of consequences:

- Casting is with respect to erased types
- Overload resolution is with respect to erased types
- Reflection is with respect to erased types

9.7 Static members on floating-point types

The parameterized floating point types `float32`, `float` and `decimal` are assumed to have additional static members with measure types as follows (here `F` is either `float32` or `float` and `D` is either `float32`, `float` or `decimal`):

- `Sqrt` has type `F<'U^2> -> F<'U>`
- `Atan2` has type `F<'U> -> F<'U> -> F<1>`.
- `op_Addition` and `op_Subtraction` and `op_Modulus` have type `D<'U> -> D<'U> -> D<'U>`
- `op_Multiply` has type `D<'U> -> D<'V> -> D<'U 'V>`
- `op_Division` has type `D<'U> -> D<'V> -> D<'U/'V>`
- `Abs` and `op_UnaryNegation` and `op_UnaryPlus` have type `D<'U> -> D<'U>`
- `Sign` has type `D<'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`.

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, the types/modules may contain member/let bindings, which finally contain expressions.

Declaration elements are processed with respect to an *environment*. The definition of the various elements of an environment is found in §14.1.

```
namespace-decl-group :=
  | namespace long-ident module-elems          -- modules within a namespace

module-defn :=
  | attributes_opt module access_opt ident = module-defn-body

module-defn-body :=
  | begin module-elems_opt end

module-elem :=
  | module-let-binding                    -- top level let or do bindings
  | 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-let-binding :=
  | attributes_opt let rec_opt bindings          -- binding
  | attributes_opt do expr                       -- side-effect binding

import-decl := open long-ident

module-abbrev := module ident = long-ident

compiler-directive-decl := # ident string ... string
module-elems := module-elem ... module-elem
```

10.1 Namespace Declaration Groups

With respect to naming, modules and types in an F# program program are organized into *namespaces*. New components may contribute entities to existing namespaces. Each such contribution to each namespace is called a *namespace declaration group*. For example

```
namespace MyCompany.MyLibrary

   module Values1 =
      let x = 1
```

A namespace declaration group is the basic declaration unit within F# implementation files and contains a series of module and type definitions contributed to the indicated namespace. A file may contain multiple namespace declaration groups, e.g.,

```
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
```

A namespace declaration group can only contain types and modules, but not values, e.g.,

```
namespace MyCompany.MyLibrary

   type MyType() =        // allowed
      let x = 1
      member v.P = x+2

   let id x = x + 1        // not allowed: values are not allowed in namespaces

   module MyInnerModule = // allowed
      let myValue = 1
```

When a namespace declaration group `namespace N declarations` is checked under an input environment env_0, the individual declarations are checked in order and an overall *signature* N_{sig} is inferred for the module. An entry for N is then added to the *ModulesAndNamespaces* table in the original environment env_0 (see §14.1.9).

Like module declarations, namespace declaration groups are processed linearly rather than simultaneously, so that later namespace declaration groups are not in scope when processing earlier ones.

```
namespace Utilities.Part1

   module Module1 =
      let x = Utilities.Part2.StorageCache()  // error (Part2 not yet declared)

namespace Utilities.Part2

   type StorageCache() =
      member cache.Clear() = ()
```

Within a namespace declaration group, the namespace itself is implicitly opened if any preceding namespace declaration groups or referenced assemblies contribute to this namespace, e.g.

```
namespace MyCompany.MyLibrary

   module Values1 =
      let x = 1

namespace MyCompany.MyLibrary

   // Implicit open of MyCompany.MyLibrary bringing Values1 into scope

   module Values2 =
      let x = Values1.x
```

However this only opens the namespace as constituted by preceding namespace declaration groups.

10.2 Module Definitions

Module definitions are named collections of declarations. For example:

```
module MyModule =
    let x = 1
    type Foo = A | B
    module MyNestedModule =
        let f y = y + 1
        type Bar = C | D
```

> The `begin/end` tokens for modules can be omitted when lightweight syntax is used.

When a module definition `module M = declarations` is checked under an input 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 in the original environment env_0.

Like namespace declaration groups, module declarations are processed linearly rather than simultaneously, so that later namespace declaration groups are not in scope when processing earlier ones.

```
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. Adding the attribute `[<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix)>]` adds the suffix `Module` to the name of a module for the purposes of this check.

10.3 Import Declarations

Entities in namespaces and modules can be made accessible via shortened long-identifiers using an *import declaration*. For example:

```
open Microsoft.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* (see §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.

Each F_i is added to the environment successively using the technique specified in §14.1.9.

A warning is given if any F_i is a module has the `RequireQualifiedAccessAttribute` attribute.

10.4 Module Abbreviations

Module abbreviations define a local name for a module long identifier. For example:

```
module Ops = Microsoft.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 `module` *`ident`* `=` *`Long-ident`* is processed by first resolving the *`Long-ident`* to a list of modules by *Name Resolution in Module and Namespace Paths* (see §14.1). These are then appended to the set of names associated with *`ident`* in the *ModulesAndNamespaces* table.

Note: older versions of F# allowed a module abbreviation to point to a namespace. This is no longer supported. For example, the following abbreviation results in a warning, and is ignored:

```
module FSCollections = Microsoft.FSharp.Collections
```

10.5 "let" Bindings in Modules

"Let" bindings in modules introduce named values and functions.

`let` rec_{opt} $binding_1$ `and ... and` $binding_n$

For example:

```
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)
```

Values and functions defined by "let" declarations are referred to simply as "values". "Let" declarations 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 non-function value with explicit type variables is called a type function (§10.5.4).

"Let" declarations may specify attributes for values.

```
    [<System.Obsolete("Don't use this")>]
    let pear v = (v, v)
```

If more than one value is defined by a "let" definition through pattern matching then the attributes apply to each value.

```
    [<System.Obsolete("Don't use this")>]
    let (a, b) = (1, 2)
```

Attributes may be given immediately prior to each value defined in a pattern:

```
    let ([<System.Obsolete("Not warm enough? Consider Mercury?")>] venus,
         [<System.Obsolete("Not cold enough? Consider Pluto?")>] mars) =
                  ("hot", "cold")
```

Values may be declared mutable:

```
let mutable count = 1
let FreshName() = (count <- count + 1; count)
```

10.5.1 Processing of "let" Bindings in Modules

Let bindings in modules are processed in the same way as `let` and `let rec` bindings in expressions (§6.7.1.1) with the following adjustments:

- Each value defined may have an accessibility annotation (§11.1). The default accessibility annotation of a let-binding in a module is *public*.
- Each value defined is *externally accessible* if it has an accessibility annotation of public and is not hidden by an explicit signature. Externally accessible values have a guaranteed compiled CLI representation of these bindings in compiled CLI binaries.
- Each value defined can be used to satisfy the requirements of any signature for the module (§12.2).
- Each value defined is subject to arity analysis (§14.11)
- Values may have attributes, including the `ThreadStaticAttribute` or `ContextStaticAttribute` attributes.

10.5.2 "do" bindings

A "do" binding within a module may have the forms

```
do expr
expr
```

The expression *expr* is checked with an arbitrary initial type *ty*. After checking $expr_1$, *ty* is asserted to be equal to `unit`. If this attempt fails, a warning rather than an error is reported. This warning is suppressed for plain expressions without "do" in script files (i.e. `.fsx` and `.fsscript` files).

"do" bindings may be given attributes, e.g.

```
let main() =
    let form = new System.Windows.Forms.Form()
    System.Windows.Forms.Application.Run(form)

[<STAThread>]
do main()
```

10.5.3 Literals

Let bindings in modules may be given the `LiteralAttribute` attribute. For example,

```
[<Literal>]
let PI = 3.141592654
```

Literal values may be used in custom attributes and pattern matching, e.g.

```
[<Literal>]
let StartOfWeek = System.DayOfWeek.Monday
[MyAttribute(StartOfWeek)]
let feeling(day) =
    match day with
    | StartOfWeek -> "rough"
    | _ -> "great"
```

Such a value is subject to the following restrictions:

- It may not be marked `mutable` or `inline`, nor given the `ThreadStaticAttribute` or `ContextStaticAttribute` attributes.
- The right-hand-side expression must be a constant expression (See §6.11)

10.5.4 Type Functions

Let bindings of values within modules may be given explicit type parameters. For example,

```
let empty<'T> : (list<'T> * Set<'T>) = ([], Set.empty)
```

A value with explicit type parameters but arity `[]` (i.e. no explicit function parameters) is called a *type function*. Type functions are rarely used in F# programming, though are very convenient when necessary. Some example type functions from the F# library are:

```
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 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, e.g. empty sets and maps.

Type functions are treated specially with respect to generalization (§14.7) and signature conformance (§12.2), and are generally attributed with one of the `RequiresExplicitTypeArgumentsAttribute` and `GeneralizableValueAttribute` attributes. Type functions may not be defined inside types, expressions and computation expressions because “let” declarations at these points may not include explicit type parameters.

While type functions should, as a rule, only be used for “pure” computations, type functions may still perform computations. For example:

```
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 as a function definition taking 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 be invocations of such a function.

10.5.5 Active Pattern Bindings

Let bindings of values with an *active-pattern-op-name* within modules introduce pattern matching tags into the environment when that 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.

11 Accessibility, Attributes and Reflection

```
access :=
  | private
  | internal
  | public

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
```

11.1 Accessibility Annotations

Accessibilities may be specified on many declaration elements. The permitted user-specified accessibilities are:

`public`	No requirements on access
`private`	Access is only permitted from the enclosing type `C`, module `M` or namespace declaration group `N`.
`internal`	Access is only permitted from the enclosing assembly `A`.

The default accessibilities are always “public”, i.e.:

- Let-bindings, 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

However, some bindings have restricted lexical scope, in particular:

- Let-bindings in classes are only lexically available within the class being defined, and only from their point of their definition onward
- Module type abbreviations are only lexically available within the module or namespace declaration group being defined, and only from their point of their definition onward
- If an implementation file has a signature, then only the items in the signature are available to other implementation and signature files.

Note that

- “`private`” on a member means “private to the enclosing type or module”

- "`private`" on a let-binding 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: Declaring family/protected specifications are not supported

11.1.1 Permitted Locations of Accessibility Modifiers

An accessibility modifier always comes before `mutable` and `inline` in let-bindings in modules:

```
let private x = 1
let private inline f x = 1
let private mutable x = 1
```

In a concrete module, it comes before the identifier:

```
module private M =
    let x = 1
```

Similarly for concrete type definitions:

```
type private C = A | B
type private C<'T> = A | B
```

- Not on `inherit` definitions (always same as enclosing class)
- Not on `interface` definitions (always same as enclosing class)
- Not on `abstract member` definitions (always same as enclosing class)
- Not on individual union cases

On "val" definitions in classes

```
val private x : int
```

On explicit "new" definitions in classes

```
    new private () = { inherit Base }
```

On implicit "new" definitions in classes

```
    type C private() = ...
```

On members in classes

```
    member private x.X = 1
```

On explicit property get/set in classes

```
  member v.Item
     with private get i = 1
     and  private set i v = ()
```

On type representations:

```
type Cases =
    private
        | A
        | B
```

11.2 Custom Attributes

CLI languages support the notion of *custom metadata 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 only be applied to certain target language constructs according to the `AttributeUsageAttribute` found on the attribute class itself. A warning will be given if an attempt is made to attach an attribute to an incorrect language construct.

Attributes are not permitted on let-bindings in expressions, classes or computation expressions. Attributes on parameters are given as follows:

```
let foo([<SomeAttribute>] a) = a + 5
```

Attributes on return values are given as follows:

```
let foo a : [<SomeAttribute>] = a + 5
```

Attributes on primary constructors are given prior to 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
```

Attributes may be attached to items in F# signature files (`.fsi` and `.mli` files) and these are incorporated into any F#-specific metadata associated with the generated assembly and the CLI IL metadata for the generated assembly. Attributes found in signature files are not attached to the compiled form of F# declarations unless they are also present in the implementation file. If signature files are used then attributes that are relevant to F# type checking must be placed on both signatures and implementations. For example, the `Obsolete` attribute should be placed in both signature and implementation. Attributes from signatures must be duplicated in implementation files if they are needed to typecheck the implementation file.

Custom attributes are mapped to compiled CLI metadata as follows:

- Attributes map to the element specified by their target, if a target is given
- An attribute on a type *`type`* becomes an attribute on the corresponding CLI type definition, whose `System.Type` object is returned by `typeof<`*`type`*`>`
- An attribute on a record field `F` for a type `T` by default become attributes on the CLI property for the field, whose name is `F`, unless the target of the attribute is "`field`", when they become attributes on the underlying backing field for the CLI property, whose name is `_F`.
- An attribute on a union discriminated union case `ABC` for a type `T` becomes an attribute on a static method on the CLI type definition `T`. This method is called:
    - `get_ABC` if the union discriminated union case takes no arguments
    - `ABC` otherwise
- Attributes on arguments are only propagated for arguments of member definitions, and not for let-bound function definitions
- Attributes on generic parameters are not propagated

Attributes placed immediately prior to `do` bindings in modules anywhere in an assembly are attached to one of

- The `main` entry point of the program
- The compiled module
- The compiled assembly

Attributes are attached to the main entry point if it is legitimate for them to be attached to a method according to the `AttributeUsageAttribute` found on the attribute class itself, and likewise for the assembly (the main method takes precedence if it is legitimate for the attribute to be attached to either).

For example, the `STAThreadAttribute` should be placed immediately prior to a top-level `do` binding.

```
let main() =
    let form = new System.Windows.Forms.Form()
    System.Windows.Forms.Application.Run(form)

[<STAThread>]
do main()
```

11.3 Reflected Forms of Declaration Elements

The `typeof` and `typedefof` F# library operators return a `System.Type` object for an F# type definition, which, according to typical implementations of the CLI execution environment, 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 declaration elements will have corresponding `System.Reflection` objects returned via this route:

- All member declarations will be 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 in such a way as to provide a corresponding compiled CLI type declaration and `System.Type` object, though this is not accessible using the typeof operator. However:
- Internal and private module "let" 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 consistent with the erasure of F# type abbreviations and F# unit of measure annotations.
- The definition of new units of measure do give rise to corresponding compiled CLI type declarations with an associated `System.Type`.

12 Signatures

Signature types give a precise specification of the functionality implemented by a module or namespace declaration group. They also act as a way to hide functionality contained within an implementation file.

```
namespace-decl-group-spec :=
  | namespace long-ident module-spec-elements  -- modules within a namespace
  | module long-ident module-spec-elements     -- named module
  | module-spec-elements                          -- anonymous module

module-spec =
  | module ident = module-spec-body

module-spec-element :=
  | val mutable_opt curried-sig              -- value signature
  | val binding                              -- literal value signature
  | type type-specs                          -- type(s) signature
  | exception exception-spec                 -- exception signature
  | module-spec                              -- submodule signature
  | module-abbrev                            -- locally alias a module
  | import-decl                              -- locally import contents of a module

module-spec-elements := module-spec-element ... module-spec-element

module-spec-body =
  | begin module-spec-elements end

type-spec :=
  | abbrev-type-spec
  | record-type-spec
  | union-type-spec
  | anon-type-spec
  | class-type-spec
  | struct-type-spec
  | interface-type-spec
  | enum-type-spec
  | delegate-type-spec
  | type-extension-spec

type-specs := type-spec ... and ... type-spec

type-spec-element :=
  | attributes_opt access_opt new : uncurried-sig        -- constructor signature
  | attributes_opt member  access_opt member-sig          -- member signature
  | attributes_opt abstract access_opt member-sig         -- member signature
  | attributes_opt 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-spec := type-name '=' type

union-type-spec := type-name '=' union-type-cases type-extension-elements-spec_opt

record-type-spec := type-name '=' '{' record-fields '}' type-extension-elements-
spec_opt

anon-type-spec := type-name '=' begin type-elements-spec end

class-type-spec := type-name '=' class type-elements-spec end

struct-type-spec := type-name '=' struct type-elements-spec end

interface-type-spec := type-name '=' interface type-elements-spec end

enum-type-spec := type-name '=' enum-type-cases

delegate-type-spec :=  type-name '=' delegate-signature

type-extension-spec := type-name type-extension-elements-spec
```

```
type-extension-elements-spec := with type-elements-spec end
```

The `begin/end` tokens are optional when lightweight syntax is used.

Like module declarations, signature declarations are processed linearly rather than simultaneously, so that later signature declarations are not in scope when processing earlier ones.

```
namespace Utilities.Part1

   module Module1 =
      val x : Utilities.Part2.StorageCache  // error (Part2 not yet declared)

namespace Utilities.Part2

   type StorageCache =
      new : unit -> unit
```

12.1 Signature Types

Work in progress. This section doesn't document all constructs that may be used in signatures.

Value specifications in signatures indicate the existence of a value in the implementation. For example in the signature of a module:

```
module MyMap =
    val mapForward : index1: int * index2: int -> string
    val mapBackward : name: string -> (int * int)
```

Type specifications in signatures indicate the existence of a corresponding type definition in the implementation.

Member specifications in signatures indicate the existence of a corresponding member on the corresponding type definition in the implementation. Member specifications must specify argument and return types, and optionally specify names and attributes for parameters. For example in an interface type:

```
type IMap =
    interface
        abstract Forward : index1: int * index2: int -> string
        abstract Backward : name: string -> (int * int)
    end
```

12.2 Signature Conformance

Work in progress. This section doesn't document all checks made by the implementation.

Constructs can be hidden by signatures, with the following exceptions:

- Type abbreviations may not be hidden by signatures.
- Any type whose representation is a record or discriminated union must reveal either all or none of its fields/constructors, in the same order as that specified in the implementation. Types whose representations are classes may reveal some, all or none of their fields in a signature.
- Any type which is revealed to be an interface, or a type which is a class or struct with one or more constructors may not hide its `inherit` declaration, abstract dispatch slot declarations or abstract interface declarations.

12.2.1 Conformance for values

If a value with a given name is present in both signature and implementation then:

- The declared accessibilities, `inline` and `mutable` modifiers must be identical
- If either has the `[<Literal>]` attribute then both must, and furthermore the declared literal value for each must be identical
- The number of inferred and/or explicit generic type parameters must be identical
- The types and type constraints must be identical up to alpha-conversion of inferred and/or explicit generic type parameters
- The arities must match (see below)

12.2.1.1 Arity Conformance for values

Arities of values must conform between implementation and signature. Arities of values are implicit in module signatures. A signature containing:

```
val F : ty1,1...ty1,A1 * ... * tyn,1...tyn,An -> rty
```

will be result in an arity $[A_1...A_n]$ for `F`. Arities in a signature must be equal or shorter than corresponding arities in an implementation, and the prefix must match. This means 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 may (or may not) be a first-class computed expression that computes to a function value, rather than a compile-time function value. The first can only be satisfied by a true function, i.e., the implementation must be a lambda value as in

```
let F x = x + 1
```

Note: because arity inference also permits r.h.s. lambdas, the implementation may currently also be:

```
let F = fun x -> x + 1
```

The signature

```
val F: (int -> int)
```

can be satisfied by any value of the appropriate type, e.g.,

```
let f =
    let myTable = Hashtbl.create 4 in
    fun x ->
        match (Hashtbl.tryfind myTable x) with
        | None -> let res = x*x in Hashtbl.add myTable x res; res
        | Some(x) -> x
```

or

```
let f = fun x -> x + 1
```

or

```
// throw an exception as soon as the module bindings are executed
let f : int -> int = failwith "failure"
```

> Note: In either case 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 rationale for this interpretation of top-level types is that CLI interoperability requires that F# function be compiled to methods, rather than to fields which are function values. Thus we inevitably need some kind of information in signatures to reveal the desired arity of a method as it is revealed to other CLI programming languages. We could use other annotations, e.g an explicit attribute syntax. However that would mean that these annotations would be required in the normal case where functions are implemented as true methods. F# is thus biased toward a mechanism that achieves interoperability without requiring a wealth of annotations to do so.

12.2.1.2 Conformance for Type Functions

If a value is a type function then its corresponding signature element must also be a type function signature, i.e. a signature with 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>
```

> Rationale: The second signature indicates that the value is, by default, generalizable (§14.7). A type function is not generalizable unless marked with the GeneralizableValueAttribute.

12.2.2 Conformance for members

- Abstract members must be present in the signature if a representation is given for a type
- If one is an extension member then both must be
- The `OverloadIDAttribute` attributes must result in identical resolved names
- The `static`, `abstract` and `override` qualifiers must match precisely
- If one is a constructor then both must be
- If one is a property then both must be
- The types must be identical up to alpha-conversion (as for values)

13 Program Structure and Execution

F# programs are made up of a collection of assemblies. F# assemblies are made up of a set of static references to existing assemblies, called the *referenced assemblies*, plus either:

- an interspersed sequence of signature (`.fsi`) files and implementation (`.fs`) files; or
- a single script (`.fsx` or `.fsscript`) file

Script files may also be used as if they are implementation files, though are then subject to the rules of implementation files.

```
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-spec ... namespace-decl-group-spec

named-module :=
  | module long-ident module-elems              -- named module

anonynmous-module :=
```

13.1 | *module-elems* -- anonymous module Implementation Files

Implementation files are made up of a number of *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
```

A single namespace declaration group and containing module can be specified implicitly at the head of a file using a `module` declaration with a long path, for example:

```
module MyCompany.MyLibrary.MyModule

let x = 1
```

This is equivalent to:

```
namespace MyCompany.MyLibrary

module MyModule =
    let x = 1
```

The final identifier is treated as the module name, the preceding identifiers as the namespace.

Anonymous implementation files are those without either a leading `module` or `namespace` declaration. These may 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. The extension is removed and the first letter capitalized.

> Note: if the file name contains characters that are not part of an F# identifier then the resulting module name is unusable.

Implementation files are checked as follows. Given an initial environment env_0,

- A new constraint solving context is created
- Each *namespace-decl-group*$_i$ is checked in turn and added to the environment, giving a new environment env_i.
- Default solutions (REF) are applied to any remaining type inference variables that include `default` constraints. These are applied in the order that the type variables appear in the type-annotated text of the checked namespace declaration groups.
- The inferred signature (REF) of the implementation file is checked against any required signature using *Signature Conformance* (REF). The *resulting signature* of an implementation file is the required signature, if present, else it is the inferred signature.
- The resulting signature must not contain any free inference type variables, else a “value restriction” error is reported.
- Arbitrary choice solutions (REF) are applied to any remaining type inference variables

After each implementation file is checked, a new environment is created by adding the namespace declaration groups with their resulting signatures to env_0.

Like module declarations, namespace declaration groups are processed linearly rather than simultaneously, so that later namespace declaration groups are not in scope when processing earlier ones. This rules out invalid recursive definitions, e.g.

```
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
```

13.1.1 Initialization Semantics for Implementation Files

Each implementation file gives rise to a *static initializer*. This consists of the evaluation of each of the bindings in each of the modules in the file. The static initializer is executed “on demand”. This means it is executed at an unspecified point prior to the first point where evaluation dereferences a value in the module whose computation may involve a visible side effect. This is called *forcing* the static initializer.

> Note: The execution of an expression *expr* is considered to have a visible side effect if there is a program context `C[-]` (i.e. a program with a single place holder) such that `C[let v = expr in ()]` and `C[()]` evaluate to different results. When assessing whether an expression causes a side effect, F# compilers may assume that program contexts do not measure operational information arising from the evaluation of expressions such as processor instruction counts and timing information.

F# Interactive executes the static initializer for each program fragment immediately.

For EXEs, the static initializer for the last module specified on the command-line when building the .EXE is forced immediately on startup.

For DLLs, no static initializer is forced when the DLL is loaded.

If the execution environment supports the execution of multiple threads of F# code then each top-level binding in each static initializer may be run in a mutual exclusion region which ensures that threads that attempt to access the results of the static initialization are paused until those results are available. The granularity of mutual exclusion is under-specified, but may be as large as an entire static initializer for an implementation file.

Modules and types within implementation files are given their own static initializers. The static initializer for each module forces each static initializer for each module that occurs before the module in the implementation file. The static initializer for the implementation file forces the static initializer for each module in the file.

The static initializers for types are not forced by the static initializer of the enclosing implementation file or module. Instead they are forced prior to the first point where evaluation dereferences a static value in the type whose computation may involve a visible side effect. The static initializer for each type forces each static initializer for each module that occurs before the type in the implementation file.

> Note: Traditionally ML dialects have used an "evaluate everything" semantics for the initialization order of top-level bindings. However, the semantics above is useful when executing code in a multi-language, dynamic loading environment.

13.1.2 Explicit "Main" Entry Point

The last file specified in the compilation order for a .EXE may additionally contain an explicit entry point, indicated by annotating a function in a module with `EntryPointAttribute`.

- The attribute can be applied only to a let-bound function in a module. It may not be a member.
- Only one function may be given this attribute, and this 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`" prior to being checked. If this assertion fails an error is reported.
- At runtime the arguments passed on startup are an array containing the same as entries as `System.Environment.GetCommandLineArgs()`, minus the first entry in that array.

The function becomes the entry point to the program. It immediately forces the static initializer for the file in which the function exists. It will then run the body of the function.

13.2 Signature Files

Signature files give a precise specification of the functionality implemented by a corresponding implementation file and contain a single signature type. After the implementation file is processed it is checked to see if it conforms to the signature.

The inclusion of a signature file in compilation implicitly applies that signature type to the contents of a corresponding implementation file.

Anonymous signature files are those without either a leading `module` or `namespace` declaration. The names of these module specifications are similarly derived from the name of the source file that contains the module.

13.3 Compiler Directives

Compiler directives are declarations in non-nested modules or namespace declaration groups of the form `# id string ... string`.

The lexical directives `#if`, `#else`, `#endif` and `#indent "off"` are similar to compiler directives but are dealt with elsewhere in this specification.

The following directives are valid in all files

Directive	Example	Short Description
`#nowarn`	`#nowarn "54"`	Turn off warnings within this lexical scope

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"`
		Reference a DLL within this entire script

`#I` `#Include`	`#I @"Nunit.Core.dll"`	Add a path to the search paths for DLLs used within this entire script
`#load`	`#load "library.fs"` `#load "core.fsi" "core.fs"`	Load a set of signature and implementation files into the script execution engine

`#time`	`#time`
	`#time "on"`
	`#time "off"`
		Toggle or set timing on or off

`#help`	`#help`	Ask the script execution environment for help
`#q` `#quit`	`#q` `#quit`	Request the script execution environment to halt execution and exit

14 Inference Procedures

14.1 Name Resolution

14.1.1 Name Environments

Each point in the interpretation of an F# program is subject to an environment. This encompasses:

- The full set of referenced external DLLs (assemblies)
- *ModulesAndNamespaces* : a table mapping `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 should there be any module called `System.Collections` declared or in a referenced assembly.

  If multiple assemblies are referenced then the fragments are presented in the reverse of the order in which the references are specified on the command line. This only matters if ambiguities arise in referencing the contents of assemblies, e.g. if two assemblies define the type `MyNamespace.C`.

- *ExprItems* : a table of mapping names to items, where an item is a:
  - value
  - discriminated union case (for use when constructing data)
  - active pattern result tag (for use when returning results from active patterns)
- *FieldLabels* : a table of mapping names to sets of field references for record types
- *PatItems* : a table of mapping names to items, where an item is a:
  - discriminated union case (for use when pattern matching on data)
  - active pattern choice tag (for use when specifying active patterns)
- *Types*: a table mapping names to type definitions. Two queries are supported on this table:
  - Find a type by name alone. This may return multiple types, e.g. in the default type checking environment, resolving `Microsoft.FSharp.Core.Tuple` will return multiple tuple types.
  - Find a type by name and generic arity `n`. This will return at most one type, e.g. in the default type checking environment, resolving `Microsoft.FSharp.Core.Tuple` with `n = 2` will return a single type.
- *ExtensionsInScope* : a table of mapping 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

Overview: 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. A list of modules and namespace declaration groups is returned.

Details: A prefix of *long-ident* is resolved to a list of modules and namespace declaration groups by consulting the *ModulesAndNamespaces* table. The remaining identifiers recursively consult the declared modules and sub-modules of these fragments and all results are concatenated together.

For example, if the environment contains two referenced DLLs, each with namespace declaration groups for namespaces `System`, `System.Collections` and `System.Collections.Generic`, then *Name Resolution in Module and Namespace Paths* for `System.Collections` will return the two entries for that namespace.

14.1.3 Name Resolution in Expressions

Overview: Given an input *long-ident*, environment *env* and an (optional) count *n* of the number of subsequent type arguments `<_,...,_>`, *Name Resolution in Expressions* computes the result of interpreting a prefix of *long-ident*`<_,...,_>` as a value or other expression item and a residue path *rest*.

Details:

If *long-ident* is a single identifier *ident* then

- Lookup *ident* in the *ExprItems* table and return the result and empty *rest*
- Otherwise lookup *ident* in the *Types* table, with generic arity matching *n* if available, returning this type and empty *rest*.
- Otherwise fail

If *long-ident* is made up of more than one identifier *ident.rest*:

- If *ident* exists as a value in the *ExprItems* table then return the result, with *rest* as the residue.
- Otherwise, consider the following backtracking search:
    - Consider each division of *long-ident* into [*namespace-or-module-path*].*ident*[.*rest*], with the *namespace-or-module-path* becoming successively longer
    - For each such division, consider each module and namespace declaration group *F* in the list produced by resolving *namespace-or-module-path* using *Name Resolution in Module and Namespace Paths*.
    - For each such *F* attempt to resolve *ident*[.*rest*] in the following order. If any resolution succeeds terminate the search:
        - A value in *F*, returning this item and *rest*
        - A discriminated union case in *F*, returning this item and *rest*
        - An exception constructor in *F*, returning this item and *rest*
        - An type in *F*. If *rest* is empty then return this type, otherwise resolve using *Name Resolution for Members*.
        - A [sub-]module in *F*, recursively resolving *rest* against the contents of this module
- Otherwise lookup *ident* in the *Types* table:
    - If a generic arity matching *n* is available then simply lookup any type matching *ident* and *n*.
    - If no generic arity *n* is available, and *rest* is non-empty, then
        - If the Types table contains a type *ident* without generic arguments, then resolve to this type
        - If the Types table contains a unique type *ident* with generic arguments, then resolve to this type. However, if the overall result of the *Name Resolution in Expressions*

operation is an item where the generic arguments do not appear in either the return or argument types of the item then a warning is given that the generic arguments can not be inferred from the type of the item.

- Otherwise an error is given

If *rest* is empty then return this type, otherwise resolve using *Name Resolution for Members*.

- Otherwise lookup *ident* in the *ExprItems* table and return the result and residue *rest*
- Otherwise if *ident* is a symbolic operator name and then resolve to an item indicating an implicitly resolved symbolic operator
- Otherwise fail

Ambiguities are resolved by the first result returned by the above process.

For example consider the following pathological 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 discriminated 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 doesn’t have a property Prop
let test7 = M.E.Prop2    // resolves to M.E, and then a property lookup
```

The following example shows the resolution behaviour for type lookups that are ambiguous by generic arity arity:

```
module M =
    type C<'T>() =
        static member P = 1

    type C<'T,'U>() =
        static member P = 1

let _ = new M.C()          //  gives error
let _ = new M.C<int>()     //  no error, resolves to C<'T>
let _ = M.C()              //  gives 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              //  give 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 behaviour 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<'T>
```

The following example shows a case where where a warning is given for an incomplete types, because the type parameter `'T` can not be inferred from the use `M.C.P`, as it 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
```

14.1.4 Name Resolution for Members

Overview: *Name Resolution for Members* is a sub-procedure used when resolving *.member-ident*[*.rest*] to a member, in the context of a particular type *type*.

Details: The *member-ident* is resolved as follows:

- Search the hierarchy of the type from Object to the given type
- At each type:
    - Try to resolve *member-ident* to a discriminated union cases of *type*
    - Try to resolve *member-ident* to a property group of *type*
    - Try to resolve *member-ident* to a method group of *type*
    - Try to resolve *member-ident* to a field of *type*

- Try to resolve *member-ident* to an event of *type*
- Try to resolve *member-ident* to a property group of extension members of *type*, by consulting the ExtensionsInScope table.
- Try to resolve *member-ident* to an method group of extension members of *type*, by consulting the ExtensionsInScope table.
- Try to resolve *member-ident* to a nested type *type-nested* of *type*, recursively resolving *.rest* if present, otherwise returning *type-nested*.

- At any type, the existence of a property, event, field or union case called *member-ident* causes any methods or other entities of that same name from base types to be hidden.

14.1.5 Name Resolution in Patterns

Overview: *Name Resolution for Patterns* is used when resolving *long-ident* in the context of pattern expressions. The *long-ident* must resolve to a discriminated union case, exception label, literal value or active pattern label. If it does not, the *long-ident* may represent a new variable binding in the pattern.

Details: The *member-ident* is resolved using the same process as *Name Resolution in Expressions* except the *PatItems* table is consulted instead of the *ExprItems* table.

This means values do not pollute the namespace 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
```

will result in "matched, C = 4", because C is *not* present in the *PatItems* table, and hence becomes a bound variable. In contrast,

```
[<Literal>]
let C = 3

match 4 with
| C -> sprintf "matched, C = %d" C
| _ -> sprintf "no match, C = %d" C
```

will result in "matched, C = 3", because C *is* present in the *PatItems* table, because it is a literal.

14.1.6 Name Resolution for Types

Overview: *Name Resolution for Types* is used when resolving *long-ident* in the context of a syntactic type. A generic arity matching *n* is always available. The result is a type definition and a possible residue *rest*.

Details: Given *ident*[*.rest*] then lookup *ident* in the *Types* table, with generic arity *n*, and return the result and residue *rest*.

If not present in this table,

- Consider each division of *long-ident* into [*namespace-or-module-path*].*ident*[*.rest*], with the *namespace-or-module-path* becoming successively longer
- For each such division, consider each module and namespace declaration group *F* in the list produced by resolving *namespace-or-module-path* using *Name Resolution in Module and Namespace Paths*.
- For each such *F* attempt to resolve *ident*[*.rest*] in the following order. If any resolution succeeds terminate the search:
  - An type in *F*. Return this type and residue *rest*.

- A [sub-]module in *F*, recursively resolving *rest* against the contents of this module

For example, given:

```
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")
```

the name `C` on the last line resolves to the named type `M.C<_,_>`.

14.1.7 Name Resolution for Type Variables

Whenever syntactic types and expressions are processed we assume a context that contains a mapping from identifiers to inference type variables. This is used to ensure multiple uses of the same type variable name map to the same type inference variable, e.g. for

```
let f x y = (x:'T), (y:'T)
```

then `x` and `y` are assigned the same static type, i.e. the same type inference variable associated with the name `'T`. The full inferred type of the function is:

```
val f<'T> : 'T -> 'T -> 'T * 'T
```

The mapping is threaded through the processing of expressions and types as they are processed in a left-to-right fashion. It is initially empty for any member or any other top-level construct containing expressions and types. Entries are eliminated from the map once they are generalized, so

```
let f () =
    let g1 (x:'T) = x
    let g2 (y:'T) = (y:string)
    g1 3, g1 "3", g2 "4"
```

checks correctly: `g1` is generalized: this can be seen by the fact it is applied to both integer and string types. The type variable `'T` on the second line refers to a different type inference variable, which is eventually constrained to be type `string` (and a warning is given, see below).

14.1.8 Field Label Resolution

Overview: *Field Label Resolution* specifies how we resolve identifiers such as *field1* in `{ field1=expr; ... fieldN=expr }`.

Details: Lookup all fields in all available types in the *Types* table and return the set of field declarations.

14.1.9 Opening Modules and Namespace Declaration Groups

When a module or namespace declaration group *F* is opened, items are added to name environment as follows:

- Each exception discriminated union case for each exception type definition (§8.10) in *F* is added to the *ExprItems* and *PatItems* tables based on the original order of declaration in *F*.
- Each value is added based on the original order of declaration in *F*.
    - The value is added to the *ExprItems* table

- If any value is an active pattern, then the tags of that active pattern are added to the *PatItems* table based on the original order of declaration.
- If the value is a literal it is added to the *PatItems* table.

- Each type definition is added based on the original order of declaration in F. Adding a type definition involves:
  - If the type definition is a record, add any record field labels to the *FieldLabels* table.
  - If the type is a discriminated union, add any discriminated union cases to the *ExprItems* and *PatItems* tables.
  - Add the type to the *TypeNames* table. If the type has a CLI mangled generic name such as `List`1` then an entry is added under both `List` and `List`1`.
- The member contents of each type externsion in F_i is added to the *ExtensionsInSope* table based on the original order of declaration in F_i.
- Each sub-module or sub-namespace-decl-group in F_i is added to the *ModulesAndNamespaces* table based on the original order of declaration in F_i.
- Any sub-modules marked with `Microsoft.FSharp.Core.AutoOpenAttribute` are themselves opened.

14.2 Resolving Application Expressions

Application expressions such as *x.Y<int>.Z(g).H.I.j* that make use of the "dot"-notation are resolved using a set of rules that take into account the many possible shapes and forms of these expressions and the ambiguities that may arise in their resolution. This section specifies the exact algorithmic process used to resolve these expressions.

To check an application expression, the expression is first repeatedly decomposed into a leading expression *expr* and a list of projections `projs`. The projections are each of the form

```
.long-ident-or-op      -- dot lookup projection
(expr)                 -- application projection
<types>                -- 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)`.

Note: in this specification we write sequences of these by juxtaposition, e.g. *(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, giving *projs* if *rest* is empty, and *.rest projs* otherwise.

After decomposition:

- If *expr* is a long identifier expression *long-ident*, then apply *Unqualified Lookup* on *long-ident* with projections *projs*.
- If not, check the expression against an arbitrary initial type *ty*, giving an elaborated expression *expr*. Then process *expr* , *ty* and *projs* using *Expression-Qualified Lookup* .

14.2.1 Unqualified Lookup

Overview: 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 will 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`.

Details: To compute *Unqualified Lookup*, *Long-ident* is first resolved using *Name Resolution in Expressions* (§14.1). This returns a *name resolution item* *item* and a *residue long identifier* *rest*.

> For example, Name Resolution in Expressions `v.X.Y` may resolve to 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`.

Note: *Name Resolution in Expressions* also takes as input the presence and count of subsequent type arguments in the first projection. For example, given `N.C` with projections `<int>` and `.P` then the name `N.C` is resolved under the knowledge that the subsequent projection is a type application with 1 type argument.

We then apply *Item-Qualified Lookup* for *item* and (*rest* + *projs*).

14.2.2 Item-Qualified Lookup

Overview: Given an input item *item* and projections *projs*, *Item-Qualified Lookup* computes the projection *item.projs*. This is often a recursive process: the first resolution will make use of a prefix of the information in *item.projs*, and recursive resolutions will resolve remaining projections.

Details: The *item* must be one of:

- A named value
- A discriminated 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

If not, an error occurs.

> Note: Static CLI events are accessed using the explicit `add_Handler` and `remove_Handler` methods.

If the first projection is *<types>* then we say the resolution is *has a type application* *<types>* with remaining projections *projs'*.

Once `item` is generated by name resolution, checking proceeds as follows:

- For a value reference `v`
  - The type scheme of `v` is *instantiated* giving a type *ty*.
    - If the first projection is *<types>* then *types* are processed and the results used as the arguments when instantiating the type scheme.
    - Otherwise the type scheme is *freshly instantiated*.

        - If the value is labelled with the `RequiresExplicitTypeArgumentsAttribute` attribute then the first projection must be *`<types>`*.
        - If the value has type "`byref<`ty_2`>`" then add a ByRef-dereference to the elaborated expression.
        - Insert automatic flexibility (REF) for the use of the value
    - Apply *Expression-Qualified Lookup* for type *`ty`* and any remaining projections.
- For type name, where *`projs`* begins with *`<types>.long-ident`*:
    - The *`types`* are processed and the results used as the arguments to instantiate the named type reference, generating a type *`ty`*.
    - We then apply *Name Resolution for Members* to *`ty`* and *`long-ident`*. This generates a new *`item`*.
    - Apply *Item-Qualified Lookup* to *`item`* and any remaining projections.
- For a group of type names where *`projs`* begins with *`<types>`*, *`args`* or just *`args`*:
    - The *`types`* are processed and the results used as the arguments to instantiate the named type reference, generating a type *`ty`*.
    - The object construction *`ty(args)`* is processed as an object constructor call as if it had been written `new` *`ty(args)`*.
    - Apply *Expression-Qualified Lookup* to *`item`* and any remaining projections.
- A group of method references or property indexer references
    - Apply *Method Application Resolution* for the method group. For indexer properties the underlying getter indexer methods are used 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.
        - If the result of Method Application Resolution is labelled with the `RequiresExplicitTypeArgumentsAttribute` attribute then explicit type arguments must have been given.
    - Let *`fty`* be the actual return type resulting from *Method Application Resolution*. Apply *Expression-Qualified Lookup* to *`fty`* and any remaining projections.
- A static field reference
    - Check the field for accessibility and attributes
    - Let *`fty`* be the actual type of the field (taking into account the type *`ty`* via which the field was accessed)
    - Apply *Expression-Qualified Lookup* to *`fty`* and *`projs`*.
- For a discriminated union case (i.e. a discriminated union constructor tag, an exception constructor tag or active pattern result element tag):
    - TBD

- A CLI event reference
    - TBD
- For an implicitly resolved symbolic operator name *op*
    - If the operator is a binary operator, resolve to the expression

      ```
      (fun (x:^a) (y:^b) ->
           ((^a or ^b) : static member (op) : ^a * ^b -. ^c) (x,y)).projs
      ```

      and re-check this entire expression.

14.2.3 Expression-Qualified Lookup

Overview: Given an elaborated expression *expr* of type *ty*, and projections *projs*, *Expression-Qualified Lookup* computes the "lookups or applications" for *expr.projs*.

Details:

If *projs* is empty, the type of the overall, original application expression is asserted to be *ty* and checking is complete.

If the projections start with:

- *(expr2)*, then apply *Function Application Resolution*
- *<types>*, then fail, e.g. with "uninitial type application". Types may not be applied to arbitrary expressions, only generic types, generic methods, generic values etc.

Otherwise the projections start with *.long-ident*. In this case, resolve *long-ident* using *Name Resolution for Members* (§14.1). This returns a name resolution item *item* and a residue long identifier *rest*.

> For example, for *ty* = `string` and *long-ident* = `Length`, Name Resolution for Members will return a property reference to the CLI instance property `System.String.Length`.

The *item* must be one of:

- 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

If not, an error is reported.

Checking then proceeds as follows:

- If *item* is a group of methods or a group of indexer properties, apply *Method Application Resolution* for the method group. For indexer properties the underlying getter indexer methods are used 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.

  Let *fty* be the actual return type resulting from *Method Application Resolution*. Apply *Expression-Qualified Lookup* to *fty* and any remaining projections.

- If `item` is a non-indexer getter property, *Method Application Resolution* is invoked for the method group containing only the getter method for the property, using no type arguments and one `()` argument.
- If `item` is an instance IL or F# field `F`,
    - Check the field for accessibility and attributes
    - Let `fty` be the actual type of the field (taking into account the type `ty` via which the field was accessed)
    - Assert that `ty` is a subtype of the actual containing type of the field.
    - Produce an elaborated form for `expr.F`. If `F` is a field in a value type then take the address of `expr` by using *AddressOf* operation §6.4.16.1.
    - Apply *Expression-Qualified Lookup* to `fty` and `projs`.

14.3 Function Application Resolution

Overview: Given expressions `f` and `expr` where `f` has type `ty`, and given subsequent projections `projs`, *Function Application Resolution* does the following:

- Attempt to assert that `f` has type ty_1 `->` ty_2 for new inference variables ty_1 and ty_2.
- Check `expr` with the initial type ty_1.
- Process `projs` using *Expression-Qualified* against ty_2.

If the first assertion failed, and `expr` has form `{ computation-expr }`, then

- Check the expression as the computation expression form `f { computation-expr }`, giving result type ty_1.
- Process `projs` using *Expression-Qualified Lookup* against ty_1.

14.4 Method Application Resolution

Overview: Given a method group `M`, optional type arguments `<tyargs>`, an optional syntactic argument `obj`, an optional syntactic argument `arg` and overall expected type `ty` *Method Application Resolution* resolves the overloading based on the partial type information of available. It also

- resolves optional and named arguments
- resolves out arguments
- resolves post-hoc property assignments
- applies method overload resolution
- inserts *ad hoc* conversions that are only applied for method calls

---

Note: If no syntactic argument is given then we are resolving a use of a method as a first class value, e.g. the method call in `List.map System.Environment.GetEnvironmentVariable ["PATH"; "USERNAME"]`

Details:

- Restrict the candidate method group `M` to those methods that are *accessible* from the point of resolution.

- If an argument *arg* is given, then determine the sets of *unnamed* and *named actual arguments*, *UnnamedActualArgs* and *NamedActualArgs*
  - Decompose arg into a list of arguments as follows
    - If arg is a syntactic tuple arg1 ,..., argN then use these arguments.
    - If arg is a syntactic unit value () then use a zero-length list of arguments.
  - For each argument,
    - If arg is a binary expression of the form name=expr then it is a named actual argument.
    - Otherwise arg is an unnamed actual argument.
  - If there there are no named actual arguments, and there is only one candidate method in M, accepting only one non-optional argument, then the decomposition of *arg* to tuple form is ignored and there is one named actual arg which is *arg* itself.
  - All named arguments must occur 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

- Determine the named and unnamed *prospective actual argument types*, called *ActualArgTypes*.
  - If an argument *arg* is given, the prospective actual argument types are given by fresh type inference variables for each unnamed and named actual argument.
  - If no argument *arg* is given, then
    - If the method group contains a single accessible method, then the prospective unnamed argument types are one fresh type inference variable for each non-optional, non-out parameter accepted by that method.
    - Otherwise, the expected overall type of the expression is asserted to be a function type *dty -> rty*. If *dty* is a tuple type (*dty1 * .. * dtyN*), then the prospective argument types are (*dty1, .. ,dtyN*). If *dty* is unit then the prospective argument types are empty. Otherwise the prospective argument types are *dty* alone.
    - 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
- For each candidate method in M, attempt to produce zero, one or two *prospective method call* $M_{possible}$ as follows
  - If the candidate method is generic and has been generalized, generate fresh type inference variables for its generic parameters, resulting in the *FormalTypeArgs* for $M_{possible}$

- Determine the *named* and *unnamed formal arguments*, called *NamedFormalArgs* and *UnnamedFormalArgs* respectively, by splitting the formal arguments for M into those that have a matching argument in *NamedActualArgs* and those that do not.
- If *#UnnamedFormalArgs* exceeds *#UnnamedActualArgs*, then modify *UnnamedFormalArgs* as follows
  - If all formal arguments in the suffix are "out" arguments with byref type, then remove the suffix and call it *ImplicitlyReturnedFormalArgs*.
  - If all formal arguments in the suffix are "optional" arguments, then remove the suffix and call it *ImplicitlySuppliedFormalArgs*.
- If the last element of *UnnamedFormalArgs* has the ParamArrayAttribute and type *pty[]* for some *pty*, then modify *UnnamedActualArgs* as follows
  - if *#UnnamedActualArgs* exceeds the number of *#UnnamedFormalArgs-1*, then produce a prospective method call with the excess of *UnnamedActualArgs* removed and called *ParamArrayActualArgs*
  - if *#UnnamedActualArgs* equals *#UnnamedFormalArgs-1*, then produce two prospective method calls
    - one with an empty *ParamArrayActualArgs*
    - one with no *ParamArrayActualArgs*
  - If *ParamArrayActualArgs* has been produced, then $M_{possible}$ is said to *use ParamArray conversion with type pty*
- Each *name=arg* in *NamedActualArgs* is then associated with a *target*. A target is a *named formal argument*, a *settable return property* or a *settable return field* as follows
  - If an argument exists in *NamedFormalArgs* with name *name*, then that argument is the target
  - If the return type of M, prior to the application of any type arguments tyargs, contains a settable property *name*, then it is the target
  - If the return type of M, prior to the application of any type arguments tyargs, contains a settable field *name*, then it is the target
- No prospective method call is generated if
  - Any named argument can not be associated with a target
  - *#UnnamedActualArgs* is less than *#UnnamedFormalArgs* (after the above processing steps)
  - *#ActualTypeArgs*, if given, doesn't precisely equal *#FormalTypeArgs* for M.
  - The candidate method is static and obj is provided, or the candidate method is instance and obj is not provided.

➢ Next, attempt to eagerly apply expected types prior to argument checking. If only one prospective method call $M_{possible}$ exists, then *assert* $M_{possible}$ by performing the following steps:

- If present, each $ActualTypeArg_i$ is asserted to be equal to its corresponding $FormalTypeArg_i$
- If present, the type of *obj* is asserted to be a subtype of the containing type of the method M.
- For each $UnnamedActualArg_i$ and $UnnamedFormalArg_i$, the corresponding *ActualArgType* is asserted to coerce to the type of the corresponding argument of M.

- If $M_{possible}$ uses ParamArray conversion with type pty then for each $ParamArrayActualArg_i$, the corresponding $ActualArgType$ is asserted to coerce to pty.
- For each of $NamedActualArg_i$ with an associated formal argument target, the corresponding $ActualArgType$ is asserted to coerce to the type of the corresponding argument of M.
- For each of $NamedActualArg_i$ with an associated property or field setter target, the corresponding $ActualArgType$ is asserted to coerce to the type of the property or field.
- The *prospective formal return type* is asserted to coerce to the *expected actual return type*. If the method M has return type rty, then the *formal return type* is defined as follows
  - If the prospective method call contains $ImplicitlyReturnedFormalArgs$ with type ty_1, ..., ty_N, then 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.

Note: The effect of the eager application of uniquely applicable methods is to flow information from the expected argument types to the checking of the actual arguments. This type information flow will not happen if more than one applicable method exists, i.e. if the method is overloaded where multiple overloads accept the same number of arguments.

- Check and elaborate argument expressions. If arg is present,
  - Check and elaborate each unnamed actual argument expression arg_i using an expected type given by the corresponding type in $ActualArgTypes$.
  - Check and elaborate each named actual argument expression arg_i, using an expected type given by the corresponding type in $ActualArgTypes$.
- Choose a unique $M_{possible}$ by the following rules:
  - For each $M_{possible}$, determine if the method is applicable. This is done by attempting to assert $M_{possible}$ (see above). If an inconsistent constraint set is detected (§14.6) by any step of this process the method is not applicable. In any case, the overall constraint set is left unchanged as a result of determining the applicability of each $M_{possible}$.
  - If there exists a unique applicable $M_{possible}$ then choose that method. Otherwise, choose the unique *best* $M_{possible}$ according to the following rules:
    - Prefer candidates that don't use ParamArray conversion. If two candidates both use ParamArray conversion with types pty_1 and pty_2 and pty_1 feasibly subsumes pty_2 then prefer the second, i.e. use the candidate with the more precise type.
    - Otherwise, prefer candidates that don't have $ImplicitlyReturnedFormalArgs$.
    - Otherwise, prefer candidates that don't have $ImplicitlySuppliedFormalArgs$.
    - Otherwise, prefer candidates that are non-generic over candidates that are generic, i.e. those that have empty $ActualArgTypes$
    - Otherwise, prefer candidates that are non-generic over candidates that are generic
    - Otherwise, if two candidates have unnamed actual argument types ty_{11} ... ty_{1n} and ty_{21} ... ty_{2n} where each ty_{1i} feasibly subsumes ty_{2i} then prefer the second. That is, prefer any candidate with the more specific actual argument types.
  - If there is no unique better method then an error is reported
- Once a unique best $M_{possible}$ is chosen, commit that method.

- Apply attribute checks
- Then build the resuting elaborated expression.

  Note, this section is not yet complete.

  In brief, this means building an elaborated expression through the following steps:

  - If necessary, take the address of `obj` using the *AddressOf* operation (§6.4.16.1).
  - Build the argument list by
    - Passing each argument corresponding to an `UnamedFormalArgs` where the argument is an optional argument as a `Some(_)` value.
    - For each argument corresponding to an `ImplicitlySuppliedFormalArgs` pass a `None` value.
    - Apply coercion to arguments.
  - 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.
  - For each `NamedActualArgs` whose target is a settable property or field, assign the value into the property
  - If `arg` is not present, return a lambda expression representing a first class function value.

Two additional rules are applied when checking arguments:

- If a formal parameter is of delegate type `D`, and an actual argument is syntactically a function value `(fun ...)`, then the parameter is interpreted as if it had been written `new D(fun ...)`.
- If a formal parameter is an out parameter of type `byref<ty>`, and an actual argument type is not a byref type, then the actual parameter is interpreted as if it had type `ref<ty>`, i.e. an F# reference cell can be passed where a `byref<ty>` is expected.

Note: One effect of the above rules is that

```
let r = new Random()

let roll = r.Next;;
```

(i.e. using r.Next as a first class function value) gives

```
val roll : int -> int
```

despite the fact that System.Random.Next is overloaded. Note the spec on this topic (in sec 12) is mostly just a recantation of what the implementation does, so that's not much help. So, we resolve "r.Next" with initial type of "'?ty" is indeed "if there is no information in the initial type about the number of arguments, then assume there's one argument". This is why the overload resolution chooses the middle one.

14.5 Implicit Insertion of Flexibility for Uses of Values and Members

At each point a data constructor, named value or member is used and forms an expression, flexibility is implicitly added to the expression associated with the use of the value or member, according to the inferred type of the expression.

The flexibility added is as follows:

- The type of the value or member is decomposed to be of the form

  $$ty_{11} * \ldots * ty_{1n} \rightarrow \ldots \rightarrow ty_{m1} * \ldots * ty_{mn} \rightarrow rty$$

  If the type is not of 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 an unsealed type, and is not a variable type, the type is replaced by a fresh type variable ty'_{ij} with a coercion constraint $ty'_{ij} :> ty_{ij}$.

- The expression elaborates to an expression of type

  $$ty'_{11} * \ldots * ty'_{1n} \rightarrow \ldots \rightarrow ty'_{m1} * \ldots * ty'_{mn} \rightarrow rty$$

  but otherwise semantically equivalent to the first expression.

This means 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)
```

14.6 Constraint Solving

Overview: Constraint solving is the process of processing ("solving") non-primitive constraints down to primitive constraints on type variables. It is invoked every time a constraint is added to the set of current inference constraints at any point during checking.

Details: Given a type inference context, 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*,...,*type*] : (*member-sig*)

typar : (new : unit -> 'T)

typar : struct

typar : not struct

typar : enum<*type*>

typar : delegate<*type*, *type*>

Each newly introduced constraint is solved as indicated in the following sections.

14.6.1 Solving Equational Constraints

New constraints *typar* = *type* or *type* = *typar* where *typar* is a type inference variable cause *typar* to be eliminated from the constraint problem and replaced by *type*. Other constraints associated with *typar* are then no longer primitive and are re-solved.

New constraints of the form $type\langle tyarg_{11},\ldots,tyarg_{1n}\rangle = type\langle tyarg_{21},\ldots,tyarg_{2n}\rangle$ are reduced to a series of constraints $tyarg_{1i} = tyarg_{2i}$ on identical named types and re-solved.

14.6.2 Solving Subtype Constraints

Primitive constraints of the form *typar* :> obj are discarded.

New constraints $type_1 :> type_2$ where $type_2$ is a sealed type are reduced to the constraint $type_1 = type_2$ and re-solved.

New constraints of the form *type*<*tyarg*$_{11}$,..., *tyarg*$_{1n}$> :> *type*<*tyarg*$_{21}$,..., *tyarg*$_{2n}$> or *type*<*tyarg*$_{11}$,..., *tyarg*$_{1n}$> = *type*<*tyarg*$_{21}$,..., *tyarg*$_{2n}$> are reduced to the constraints *tyarg*$_{11}$ = *tyarg*$_{21}$ ... *tyarg*$_{1n}$ = *tyarg*$_{2n}$ and re-solved.

> Note: F# generic types do not support co-variance or contra-variance. That is, while single dimensional array types in the CLI are effectively covariant, F# treats these types as invariant for the purpose of constraint solving. Likewise, F# considers CLI delegate types as invariant, and any CLI variance type annotations on generic interface types and generic delegate types are ignored.

New constraints of the form *type*$_1$<*tyarg*$_{11}$,..., *tyarg*$_{1n}$> :> *type*$_2$<*tyarg*$_{21}$,..., *tyarg*$_{2n}$> 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 re-solved.

> 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 re-solved, e.g., then the constraint `'T = int` will eventually be derived.

> Note: Subtype constraints on single dimensional array types `ty[] :> ty` are reduced to residual constraints, based on the fact that these types are considered to subtype `System.Array`, `System.Collections.Generic.IList<'T>`, `System.Collections.Generic.ICollection<'T>` and `System.Collections.Generic.IEnumerable<'T>`. multi-dimensional array types `ty[]` are also subtypes of `System.Array`.
>
> Note: types from other CLI languages may, in theory, support multiple instantiations of the same interface type, e.g., `C : I<int>, I<string>`. This makes it more difficult to solve a constraint such as `C :> I<'T>`. This is rarely used in practice in F# coding. Such a constraint is reduced to a constraint `C :> I<'T>` where `I<'T>` is the first interface type that occurs in the tree of supported interface types, from most derived to least derived, iterating left-to-right in the order of the declarations in the CLI metadata.
>
> Note: CLI variance type annotations on interfaces are ignored.

New constraints of the form *type* :> 'b are re-solved as *type* = 'b.

> Note: These constraints typically only arise when calling generic code from other CLI languages where a method accepts a parameter of a 'naked' variable type, e.g., a C# 2.0 function with a signature such as `T Choose<'T>(T x, T y)`.

14.6.3 Solving Nullness, Struct and other Simple Constraints

New constraints of the forms

```
type : null

type : (new : unit -> 'T)

type : struct

type : not struct

type : enum<type>
```

type : delegate<*type*, *type*>

where *type* is not a variable type are reduced to further constraints and resolved according to the requirements for each kind of constraint listed in §5.1.5, §5.3.8 and elsewhere.

14.6.4 Solving Member Constraints

Work in progress (being updated for latest operator overloading implementation)

New constraints of the forms

(*typar* or ... or *typar*) : (*member-sig*) are solved as *member constraints*. A static type *ty* *satisfies a member constraint* ([static] member *ident* : *arg-type*$_1$ * ... * *arg-type*$_n$ -> *ret-type*) if:

- *ty* is a named type whose type definition contains a member ([static] member *ident* : *formal-arg-type*$_1$ * ... * *formal-arg-type*$_n$ -> *ret-type*) of the given name, taking *n* arguments
- the presence of `static` matches that of the constraint
- asserting type inference constraints between the arguments and return types does not lead to a type inference error;

14.6.4.1 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 contain the definition of these operators. In particular, given the following groups of types:

- the *integral* types `byte`, `sbyte`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `nativeint` and `unativeint`
- the *signed integral* CLI types `sbyte`, `int16`, `int32`, `int64` and `nativeint`
- the *floating point* CLI types `float32` and `float`

the members assumed are:

- The *integral* types are assumed to define static members `op_BitwiseAnd`, `op_BitwiseOr`, `op_ExclusiveOr`, `op_LeftShift`, `op_RightShift`, `op_UnaryPlus`, `op_UnaryNegation`, `op_Increment`, `op_Decrement`, `op_LogicalNot` and `op_OnesComplement`.
- The *signed integral* types are assumed to define a static member `op_UnaryNegation`.
- The *floating point* types are assumed to define static members `Sin`, `Cos`, `Tan`, `Sinh`, `Cosh`, `Tanh`, `Atan`, `Acos`, `Asin`, `Exp`, `Ceiling`, `Floor`, `Round`, `Log10`, `Log`, `Sqrt`, `Atan2` and `Pow`.
- The *floating point* and *integral* types are assumed to define static members `op_Addition`, `op_Subtraction`, `op_Multiply`, `op_Division`, `op_Modulus` and `op_UnaryPlus`.
- The *floating point* and *signed integral* types are assumed to define static member `op_UnaryNegation`.
- The *floating point* and *signed integral* and `decimal` types are assumed to define static member `Sign`.
- The *floating point* and *signed integral* types are assumed to define a static member `Abs`.
- The *floating point*, *integral* and the *string* type `string` are assumed to define static members `ToByte`, `ToSByte`, `ToInt16`, `ToUInt16`, `ToInt32`, `ToUInt32`, `ToInt64`, `ToUInt64`, `ToSingle`, `ToDouble` and `ToDecimal`.
- The *floating point* and *integral* types are assumed to define static members `ToIntPtr` and `ToUIntPtr`.

Note: The decimal type is only included in one of the items above: for the Sign static member. This is deliberate: in CLI `System.Decimal` includes the definition of static members such as `op_Addition` and the existence of these methods does not need to be simulated by the F# compiler.

This mechanism is used to implement the extensible conversion and math functions of the F# library including `sin`, `cos`, `int`, `float`, `(+)` and `(-)`.

14.6.5 Over-constrained user type annotations

An implementation of F# should give a warning if a type inference variable arising from a user type annotation is constrained to be a type other than another type inference variable. For example,

```
let f (x:'T) = (x:string)
```

gives a warning because `'T` has been constrained to be precisely `string`.

14.7 Generalization

Overview: Generalization is the process of giving constructs generic types if possible, thereby making the construct re-usable at multiple different types. Generalization is applied by default at all `let`, `member` and `let rec` bindings, with the exceptions listed below. Generalization is also applied to member bindings that implement generic virtual methods in object expressions, and for field bindings implementing fields with first-class generic types.

Generalization is applied simultaneously to all bindings in a recursive group (e.g. a group of values defined using `let rec` or the collected members of a group of type definitions connected by `and`). This determines an overall set of generalized variables. These are then generalized independently for each item in that recursive group.

Details: Generalization takes a set of bindings of values, functions and members, and an environment *`env`*. It generalizes all type inference variables that are either:

- present in the inferred types of the values, functions and members; OR
- explicitly declared as generic type parameters on an item.

and which are not ungeneralizable. Ungeneralizable type inference variables are:

- Any type inference variable *`^typar`* that is part of the inferred or declared type of a binding, unless that binding is marked `inlined`.
- Any type inference variable in an inferred type in the *ExprItems* or *PatItems* tables of *`env`*, or in any inferred type of any module in the *ModulesAndNamespaces* table in *`env`*.
- Any type inference variable that is part of the inferred or declared type of a binding where the elaborated right-hand-side of the binding is not a *generalizable expression*. 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. An expression is generalizable if it is:
    - a lambda expression; or
    - an object expression implementing an interface; or
    - a delegate expression; or
    - a let binding expression where both the right-hand-side of the binding and the body itself are generalizable; or
    - a let-rec binding expression where both the right-hand-sides of all the bindings and the body of the expression are generalizable; or

- a tuple expression, all of whose elements are generalizable; or
- a record expression, all of whose elements are generalizable, where the record contains no mutable fields; or
- a discriminated union case expression, all of whose arguments are generalizable; or
- an exception expression, all of whose arguments are generalizable; or
- an empty array expression; or
- a constant expression; or
- an application of a type function labelled with the `GeneralizableValueAttribute` attribute.

- Any type inference variables appearing in a constraint that itself refers to a type inference variable that may not be generalized.

Note the generalizable type variables can be computed by a greatest-fixed-point computation, i.e

(1) start with all variables that are candidates for generalization

(2) determine a set of variables U that may not be generalized because they are free in the environment or present in ungeneralizable bindings

(3) remove these from the set

(4) add to U any inference variables with a constraint involving one of the variables from U

(5) Repeat from (2).

A known limitation exists that type parameters involving constraints are not generalized at "let" bindings in classes. For example, given

```
type C() =
  class
    let f x = (x :> System.Windows.Forms.Form)
    member this.M(x) = (x :> #System.Windows.Forms.Form)
  end;;
```

The binding for 'f' is not generalized, i.e. is not generic. However the binding for "M" is generic.

Explicit type parameter definitions on value and member definitions can affect the process of type inference and generalization (REF). In particular, a declaration that includes explicit type parameters will not be generalized beyond those type parameters. For example, consider the function

```
let f<'T> (x:'T) y = x
```

During inference this will result in a function of the following type, where `'_b` is an as-yet-to-be-resolved type inference variable.

```
f<'T> : 'T -> '_b -> '_b
```

To permit generalization at these bindings either reove the explicit type parameters (if they can be inferred), or use the required number of parameters:

```
let throw<'T,'U> (x:'T) (y:'U) = x
```

14.8 Dispatch Slot Inference

Overview: *Dispatch Slot Inference* is applied to object expressions and type definitions prior to processing members. In both cases the input is a type ty_0 being implemented, a set of members `override x.M(`$arg_1 ... arg_N$`)`, a set of additional interface types ty_1 ... ty_n , and for each ty_i a further set of members `override x.M(`$arg_1 ... arg_N$`)`. The aim of dispatch slot inference is to associate members with a unique abstract member or interface member defined or inherited by the collected types ty_i.

Details: 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 is given.

For example, given:

```
type IA = interface abstract P : int end
type IB = interface inherit IA end
type ID = interface inherit IB end
```

Then the following object expression is legal: the implementation of `IB` is the most-specific implemented type encompassing `IA`, and hence 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 gives an error because the interface `IA` is included in both `IB` and `IC`, and hence the implementation mapping for `P` would be 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`.

Next, an attempt is made to associate each member with a dispatch slot based on name and number of arguments. This is called *dispatch slot inference*.

- For each binding `member x.M(`$arg_1 ... arg_N$`)` in type ty_i, attempt to find a single dispatch slot `abstract M :` $aty_1 ... aty_N$ `->` rty with name M, argument count N and most-specific implementing type ty_i.
    - Argument counts are determined by syntactic analysis of patterns looking for tuple and unit patterns. So these members has argument count 1, despite the argument type being `unit`:

```
member obj.ToString(() | ()) = ...

member obj.ToString():unit) = ...

member obj.ToString(_:unit) = ...
```

- Members 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 inferred to both be `int`.

14.9 Dispatch Slot Checking

Overview: *Dispatch Slot Checking* is applied to object expressions and type definitions to check consistency properties such as ensuring that all abstract members are implemented.

Details: After all bodies of all methods are checked, the implementation relation is checked for the types being implemented. A 1:1 mapping must exist 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 gives rise to an elaborated *dispatch map* keyed by dispatch slot. Dispatch slots are qualified by the declaring type of the slot, so a type may supply different implementations for `I.m()` and `I2.m()`.

The construction of the dispatch map for any particular type is as follows:

- where the type definition or extension gives an implementation of an interface then mappings are added for each member of the interface,
- where the type definition or extension gives a `default` or `override` member then a mapping is added for the associated abstract member slot.

14.10 Byref Safety Analysis

Overview: "ByRef" arguments are possibly-stack-bound pointers used to pass large inline data structures and non-escaping mutable locations to procedures in CLI languages. ByRef pointers are generally unused in F# because of the use of tuple values for multiple return values and reference cells for mutable store. However, ByRef values can arise when overriding CLI methods that have signatures involving byref values.

Details: The following checks are made:

- Byref types may not be used as generic arguments
- Byref values may not be used in inner lambdas.

Strict restrictions are imposed to ensure that ByRef arguments do not escape the scope of the implementing method except by being dereferenced. This means they cannot be used inside inner closures within the implementing method - they should be dereferenced first, stored in an local value (which can be used by inner closures), and copied back at the exit of the method. In this context a "method" consists of all constructs within the implementing expression except those enclosed by a function, lambda expression or one of the implementation functions of an object expression.

14.10.1 Passing ref to methods expecting byref values.

When calling a function that accepts a byref parameter a value of type `ty ref` may be passed. The interior address of the heap-allocated cell associated with such a parameter is passed as the pointer argument.

```
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; }
    }

F# client code:
   let res1 = ref 0 in C.IntegerOutParam(res)
   // res1.contents now equals 3

   let x = {new D() with IntegerOutParam(res : int byref) = res <- 4} in
   let res2 = ref 0 in
   x.IntegerOutParam(res2);
   // res2.contents now equals 4
```

14.11 Arity Inference

During checking, members within types and let bindings within modules are inferred to have an *arity*. An arity is:

- The number of iterated (curried) arguments *n*; and
- A tuple length for each of these argument types for the inferred type $[A_1;...;A_n]$. A tuple length of zero indicates the corresponding argument type is `unit`.

Arities are inferred as follows. A function definition of form

```
let ident pat1 ... patn = ...
```

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. For example

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

is given arity `[1;2]`.

Note: arities are also inferred from lambda expressions appearing on the immediate right of a let binding, e.g. this gets an arity of [1]:

```
let f = fun x -> x + 1
```

and

```
let f x = fun y -> x + y
```

has an arity of [1;1]

Arity inferences is applied partly to help define the elaborated form of function definitions, which in turn will be the form seen by other CLI languages. In particular:

- A function value `F` in a module with arity $[A_1;...;A_n]$ and type $ty_{1,1} * ... * ty_{1,A1} -> ... -> ty_{n,1} * ... * ty_{n,An} -> rty$ will be elaborated 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 with arity [A_1] and type $ty_{1,1}$ * ... * $ty_{1,A1}$ -> rty will be elaborated to a CLI instance (respectively static) method definition with signature rty F($ty_{1,1}$, ..., $ty_{1,A1}$).

For example, a function

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

in a module will be compiled as a CLI static method with C# signature

```
int AddThemUp(int x, int y, int z);
```

14.12 Recursive Safety Analysis

Overview: F# permits recursive bindings of non-function values, e.g.

```
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
```

This raises the possibility of invalid recursive cycles exercised by strict evaluation, e.g.

```
let rec x = x + 1
```

Recursive Safety Analysis describes the process used to partially check the safety of these bindings and to convert them to a form established by lazy initialization, where runtime checks are inserted to locally enforce initialization.

Details: A right-hand-side expression is *safe* if it is a:

- any function expression (including ones whose bodies include references to variables being defined recursively)
- any object expression implementing an interface (including ones whose member bodies include references to variables being defined recursively)
- a `lazy` delayed expression
- a record, tuple, list or data construction expression whose field initialization expressions are each safe
- a value other than one of those being recursively bound
- a value being recursively bound where the value appears in one of the following positions:
    - as a field initializer for a field of a record type where the field is marked “mutable”
- any expression that refers only to variables earlier variables defined by the sequence of recursive bindings

Other right-hand-side expressions are made semi-safe by adding a new delayed computation bindings. If the original binding is

```
u = expr
```

then a fresh variable (say `v`) is generated with a delayed binding:

```
v = lazy expr
```

and occurrences of the variables in the right hand side are replaced by `Lazy.force v`. The following binding for the original variable is added after the `let rec`.

```
u = expr.Force()
```

Bindings are then established by executing the right-hand-sides in order. Delayed computations are covered later in this specification.

Explanatory text: This means that F# permits the recursive bindings where the mutual-references are hidden inside delayed values such as inner functions, other recursive functions, anonymous `fun` lambdas, lazy computations, and the methods of object-implementation expressions. In particular the expressions on the right hand side of a “let rec” can be applications, method calls, constructor invocations and other computations.

Recursive bindings that involve computation to evaluate the right-hand-sides of the bindings are executed as an “initialization graph” of delayed computations. Some recursive references may be runtime checked because there is a possibility that the computations involved in evaluating the bindings may actually take delayed computations and execute them. The F# compiler gives a warning for "let rec" bindings that may be involve a runtime check and inserts delays and thunks so that if runtime self-reference does occur then an exception will be raised.

Recursive bindings that involve computation are often used 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 have this behaviour. 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 ->
                      Printf.printf "Text = %s\n" menuItem.Text),
                 Shortcut.CtrlH)
```

A compiler warning is given for this code because in theory the `new MenuItem(...)` constructor could evaluate the callback as part of the construction process, though because the System.Windows.Forms library is well-designed this will not happen in practice.

---

15 Lexical Filtering

15.1 The Lightweight Syntax Option

F# supports the optional use of lightweight syntax through the use of whitespace to make indentation significant. This feature is the default for F# files with extension `.fs`, `.fsx`, `.fsi` and `.fsscript`.

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 `;;` by having the parser take indentation into account. This can make a surprising difference to the readability of code. Compiling your code with the indentation-aware syntax option is useful even if you continue to use explicit tokens, as it reports many indentation problems with your code and ensures a regular, clear formatting style.

For example:

```
let f x =
    let y = x + 1   // the ‘in’ token is not required on this line
    y + y
```

> Note: Do not use ;; with this option, except on a single line of its own to terminate entries to F# Interactive. The use of ;; is not needed.

In this documentation we will call the indentation-aware syntax option the "light" syntax option. When the light syntax option is enabled, comments are considered pure whitespace. This means the indentation position of comments is irrelevant and ignored. Comments act entirely as if they were replaced by whitespace characters. TAB characters may not be used when the light syntax option is enabled.

> Note: You should ensure your editor is configured to replace TAB characters with spaces, e.g., in Visual Studio 2008 go to "Tools\Options\Text Editor\F#\Tabs" and select "Insert spaces".

15.1.1 Basic lightweight syntax rules by example.

The basic rules applied when the light syntax option is activated are shown below, illustrated by example.

`// When the light syntax option is`
`// enabled top level expressions do not`
`// need to be delimited by ';;' since every construct`
`// starting at first column is implicitly a new`
`// declaration. NOTE: you still need to enter ';;' to`
`// terminate interactive entries to fsi.exe, though`
`// this is added automatically when using F#`
`// Interactive from Visual Studio.`
`printf "Hello"`
`printf "World"`
	`// Without the light syntax option the`
	`// source code must contain ';;' to separate top-level`
	`// expressions.`
	`//`
	`//`
	`#indent "off"`
	
	`printf "Hello";;`
	`printf "World";;`

`// When the light syntax option is`
`// enabled 'in' is optional. The token after the '='`
`// of a 'let' definition begins a new block, where`
`// the pre-parser inserts an implicit separating 'in'`
`// token between each 'let' binding that begins at`
`// the same column as that token.`
`let SimpleSample() =`
`    let x = 10 + 12 - 3`
`    let y = x * 2 + 1`
`    let r1,r2 = x/3, x%3`
`    (x,y,r1,r2)`
	`// Without the light syntax option 'in'`
	`// is very often required. The 'in' is optional when`
	`// the light syntax option is used.`
	`//`
	`#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)`

`// When the light syntax option is`
`// enabled 'done' is optional and the scope of`
`// structured constructs such as match, for, while`
`// and if/then/else is determined by indentation.`
	`// Without the light syntax option`
	`// 'done' is required`
	`#indent "off"`

```
let FunctionSample() =
    let tick x = printf "tick %d\n" x
    let tock x = printf "tock %d\n" 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
    printf "done!\n"
```

```
let FunctionSample() =
    let tick x = printf "tick %d\n" x in
    let tock x = printf "tock %d\n" x in
    let choose f g h x =
        if f x then g x else h x in
    for i = 0 to 10 do
        choose (fun n -> n%2 = 0) tick tock i
    done;
    printf "done!\n"
```

```
// When the light syntax option is
// enabled the scope of if/then/else is implicit from
// indentation.

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
    printf "done!\n"
```

```
// Without the light syntax option
// 'begin'/'end' or parentheses are often needed
// to delimit structured language constructs
#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;
    printf "done!\n"
```

Here are some examples of the offside rule being applied to F# code:

```
// '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)
```

```
// The '|' markers in patterns must align.
// The first '|' should always be inserted.
markers.
let f () =
    match 1+1 with
    | 2 -> printf "ok"
  | _ -> failwith "no!"   <-- syntax error
```

15.1.2 Inserted Tokens

Some hidden tokens are inserted by lexical filtering. These are shown below.

```
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 and are translations of the corresponding input tokens used to give 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 binding $in expr
    | while expr do expr $done
    | if expr then $begin expr $end
       [elif expr $then $begin expr $end]
       [else $begin expr $end]
    | for pat in 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”

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-spec-elements :=
  | $begin module-spec-element ... module-spec-element $end

module-spec :=
  | module ident = $begin module-spec-body $end
```

15.1.4 Offside lines

Indentation-aware syntax is sometimes called the "offside rule". In F# code offside lines occur at column positions. For example, a `=` token associated with `let` introduces an offside line at the column of the first token after the `=` token.

Offside lines are also introduced by other structured constructs, in particular at the column of the first token after the `then` in an `if`/`then`/`else` construct, and likewise after `try`, `else`, `->` and `with` (in a `match`/`with` or `try`/`with`) and `with` (in a type extension). "Opening" bracketing tokens `(`, `{` and `begin` also introduce an offside line. In all these cases the offside line introduced is determined by the column number of the first token following the significant token. Offside lines are also introduced by `let`, `if` and `module`. In this cases the offside line occurs at the start of the identifier.

15.1.5 The Pre-Parse Stack.

The "light" syntax option is implemented as a pre-parse of the token stream coming from a lexical analysis of the input text (according to the lexical rules above), and uses a stack of *contexts*.

- When a column position becomes an offside line a "context" is pushed.
- "Closing" bracketing tokens ("`)`", "`}`" and "`end`") automatically terminate offside contexts up to and including the context introduced by the corresponding "opening" token.

15.1.6 Full List of Offside Contexts.

The full list of contexts kept on the pre-parse stack is as follows. First, the following context is the primary context of the analysis:

- **SeqBlock .** This indicates a sequence of items which must be columned aligned, and where a delimiter replacing the regular 'in' and ';' tokens is automatically inserted as necessary between the elements. This context is pushed when
  - immediately after the start of a file, excluding lexical directives such as #if
  - immediately after a '=' token is encountered in a Let or Member context
  - immediately after a Paren, Then, Else, WithAugment, Try, Finally, Do context is pushed
  - immediately after any infix token is encountered.
  - immediately after a '->' token is encountered when in a MatchClauses context
  - immediately after an 'interface', 'class', or 'struct' token is encountered in a type declaration

Here "immediately after" refers to the fact that the column position associated with the SeqBlock is first token following the significant token.

Note, that according to the current specification, SeqBlock contexts are NOT pushed in the following situations:

> immediately after a “=” token in a field binding in a record expression

> immediately after a “lazy” token

This means

```
type R = { f : int }
{ f = let x = 1
      let y = 2
      x + y }
```

doesn’t parse correctly, where

```
type R = { f : int }
{ f = (let x = 1
       let y = 2
       x + y) }
```

does. Likewise

```
lazy let x = 1
     let y = 2
     x + y
```

does not parse, but this does:

```
lazy (let x = 1
      let y = 2
      x + y)
```

The following contexts are associated with particular nested constructs introduced by particular keywords:

- **Let** when a `'let'` keyword is encountered
- **If** when an `'if'` or `'elif'` keyword is encountered
- **Try** when a `'try'` keyword is encountered

- **Fun** when an `'fun'` keyword is encountered
- **Function** when an `'function'` keyword is encountered
- **WithLet** when a `'with'` is encountered as part of a record expression or an object expression whose members use the syntax`{ new Foo with M() = 1 and  N() = 2 }`
- **WithAugment** -- pushed when a 'with' is encountered 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** -- pushed when an `'match'` keyword is encountered
- **For** -- pushed when a `'for'` keyword is encountered
- **While** -- pushed when a `'while'` keyword is encountered
- **Then** -- pushed when a `'then'` keyword is encountered
- **Else** -- pushed when a `'else'` keyword is encountered
- **Do** -- pushed when a `'do'` keyword is encountered
- **Type** -- pushed when a `'type'` keyword is encountered
- **Namespace** -- pushed when a `'namespace'` keyword is encountered
- **Module** -- pushed when a `'module'` keyword is encountered
- **Member** -- pushed when a `'member'`, `'abstract'`, `'default'` or `'override'` keyword is encountered, though only when not already in a Member context, as multiple tokens may be present. Also pushed when a `'new'` keyword is encountered and if the next token is `'('.` This distinguishes the member declaration `new(x) = ...` from the expression `new x()`
- **Paren(token)** pushed when a `'('`, `'begin'`, `'struct'`, `'sig'`, `'{'`, `'['`, `'[|'` or `quote-op-left` is encountered
- **MatchClauses** -- pushed when a `'with'` keyword is encountered when in a Try or Match context immediately after a `'function'` keyword is encountered
- **Vanilla** -- pushed whenever an otherwise unprocessed keyword is encountered in a SeqBlock context

15.1.7 Balancing rules

When processed, the following tokens cause contexts to be popped of the offside stack until a condition is reached. When a context is popped extra tokens may be inserted to indicate the end of the construct.

```
end             pop until enclosing context is one of:
                        WithAugment
                        Paren(interface)
                        Paren(class)
                        Paren(sig)
                        Paren(struct)
                        Paren(begin)
;;              pop all
else            pop until If
elif            pop until If
done            pop until Do
in              pop until For or Let
with            pop until Match, Member, Interface, Try, Type
finally         pop until Try
)               pop until Paren(()
}               pop until Paren({)
]               pop until Paren([)
|]              pop until Paren([|)
quote-op-right  pop until Paren(quote-op-left)
```

15.1.8 Offside Tokens, Token Insertions and Closing Contexts

When a token occurs on or prior to the *offside limit* for the current offside stack and that token (which is the rightmost offside line for the offside contexts on the context stack), and a *permitted undentation* does not apply, then enclosing contexts are *closed* until the token is no longer offside. This may result in extra delimiting tokens being inserted.

Contexts are closed as follows:

- When a **Fun** context is closed, a `$end` token is inserted.
- When a **SeqBlock**, **MatchClauses**, **Let** or **Do** context is closed, a `$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 forcing the close is not `done`, then a `$done` token is inserted.
- When a **Member** context is closed, a `$end` token is inserted.
- When a **WithAugment** context is closed, a `$end` token is inserted.

If a token is offside and a context can not be closed then an "undentation" warning or error is given, indicating that the construct is badly formatted.

> Note: offside warnings and errors are usually simple to remove by adding extra indentation and applying standard structured formatting to your code.

Tokens also are inserted in the following situations.

- When a **SeqBlock** context is pushed, a `$begin` token is inserted, with the exception of the first **SeqBlock** pushed for the start of the file.
- When a token occurs directly on the offside line of **Let** context, and the token is not `and`, and the next surrounding context is a **SeqBlock**, then an `$in` token is inserted.
- When a token occurs directly on the offside line of a **SeqBlock** (for the second or subsequent lines of the block), then a `$sep` token is inserted. This plays the same role as `;` in the grammar rules.

For example, consider the source text

```
let x = 1
x
```

The raw token stream is `let`, `x`, `=`, `1`, `x` and an end-of-file marker `eof`. An initial **SeqBlock** is pushed immediately after the start of the file, i.e. 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 `$end` token. It is directly aligned with the **CtxtSeqBlock** context, so a `$seq` token is inserted.

15.1.9 Exceptions to when tokens are offside

A set of exceptions are made when determining if a token is offside from the current context:

- When in a **SeqBlock** context, an infix token may be offside by the size of the token plus one. That is, in the following examples the infix tokens :

```
let x =
        expr + expr
      + expr + expr
  let x =
        expr
    |> f expr
    |> f expr
```

  Similarly, when in a `SeqBlock` context, any infix token is permitted to align precisely with the offside line of the `SeqBlock`, without being considered offside, e.g.,:

```
let someFunction(someCollection) =
    someCollection
    |> List.map (fun x -> x + 1)
```

  In particular, the infix token `|>` that begins the last line is not considered to be 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-parenthetical operators. For example,

```
new MenuItem("&Open...",
             new EventHandler(fun _ _ ->
                         ...
                 ))
```

  The first ')' token here does not indicate a new element in a sequence of items, despite the fact that it's precisely aligned with the sequence block started at the start of the argument list.

- When in a `Let` context, an `and` token is permitted to align precisely with the `let`, without being considered offside, e.g.,:

```
let x = 1
and y = 2
x+y
```

- When in a `Type` context, `}`, `end`, `and` and `|` tokens are permitted to align precisely with the `type`, without being considered offside, e.g.,:

```
type X =
| A
| B
with
    member x.Seven = 21/3
end
and Y = {
    x:int
}
and Z() = class
    member x.Eight = 4+4
end
```

- When in a `For` context, a `done` token is permitted to align precisely with the `for`, without being considered offside, e.g.,:

```
for i = 1 to 3 do
    expr
done
```

- When entering a `SeqBlock; Match` context i.e. on the right-hand-side of an arrow for a match expression), a token is permitted to align precisely with the `match`, without being considered offside. This allows the “last” expression to be inline with the match, meaning that a long series of matches doesn’t cause increased indentation, e.g.,:

```
match x with
| Some(_) -> 1
| None ->
match y with
| Some(_) -> 2
| None ->
3
```

- When in a `Interface` context, an `end` token is permitted to align precisely with the `interface`, without being considered offside, e.g.,:

```
interface IDisposable with
    member x.Dispose() = printf "disposing!\n"
done
```

- When in a `If` context, `then`, `elif` and `else` tokens are permitted to align precisely with the `if`, without being considered offside, e.g.,:

```
if big
then callSomeFunction()
elif small
then callSomeOtherFunction()
else doSomeCleanup()
```

- When in a `Try` context, `finally` and `with` tokens are permitted to align precisely with the `try`, without being considered offside, e.g.,:

```
try
    callSomeFunction()
finally
    doSomeCleanup()
```

and

```
try
    callSomeFunction()
with Failure(s) ->
    doSomeCleanup()
```

- When in a `Do` context, a `done` token is permitted to align precisely with the `do`, without being considered offside, e.g.,:

```
for i = 1 to 3
   do
       expr
   done
```

15.1.10 Permitted Undentations.

In general, nested expressions must occur at increasing column positions in indentation-aware code, called the "incremental indentation" rule. Warnings or syntax errors will be given where this is not the case. However, for certain constructs "undentation" is permitted. In particular, undentation is permitted in the following situations.

15.1.10.1 Bodies of anonymous functions may be undented.

The bodies of functions may be undented from the 'fun' or 'function' symbol. This means the symbol is ignored when determining whether the body of the function satisfies the incremental indentation rule.

```
let HashSample(tab: Collections.HashTable<_,_>) =
    tab.Iterate (fun c v ->
        printf "Entry (%O,%O)\n" c v)
```

The block may not undent past other offside lines, so the following is not accepted because the second line breaks the offside line established by the “=”:

```
let x = (function (s, n) ->
    (fun z ->
        s+n+z))
```

Constructs enclosed by bracketing may be undented.

The bodies of a '(' ... ')' or 'begin' ... 'end' may be undented when the expressions follow a 'then' or 'else'. may not undent further than the 'if'.

```
let IfSample(day: System.DayOfWeek) =
    if day = System.DayOfWeek.Monday then (
        printf "I don't like Mondays"
    )
```

Likewise the bodies of modules and module types delimited by 'sig' ... 'end', 'struct' ... 'end', or 'begin' ... 'end' may be undented, e.g.,

```
module MyNestedModule = begin
   let one = 1
   let two = 2
end
```

Likewise the bodies of classes, interfaces and structs delimited by '{' ... '}', 'class’ ... 'end', 'struct' ... 'end', or'interface' ... 'end' may be undented, e.g.,

```
type MyNestedModule = interface
   abstract P : int
end
```

15.2 High Precedence Application

The entry "f x" in the precedence table from Section 4 refers to function application where the function and argument are separated by spaces. The entry `"f(x)"` indicates that in expressions and patterns, identifiers followed immediately by a left-parenthesis without intervening whitespace form a "high precedence" application. These are parsed with higher precedence (i.e. binding more tightly) 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

```
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 as long as the arguments of method applications are parenthesized and come immediately after the method name, without spaces, e.g.,

```
e.Meth1(arg1,arg2).Prop1.[3].Prop2.Meth2()
```

Although strictly allowed by the grammar and the precedence rules above, a sanity check ensures that high-precedence application expressions may not be used as directly arguments, and must instead be surrounded by parentheses, e.g.,

```
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, e.g.,

```
f  e.Prop1 e.Prop2.[3]
```

15.3 Lexical analysis of type applications

The entry `f<types> x` in the precedence table (§3.7) refers to any identifier followed immediately by a `<` symbol and a sequence of:

- `_`, `,`, `*`, `'`, `[`, `]`, 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 made up only of `>` characters (e.g. `>`, `>>` or `>>>`) is treated as if it is just a series of individual `>` tokens. Likewise any token made up only of `>` characters followed by a '`.`' (e.g. `>.`, `>>.` or `>>>.`) is treated as if it is just a series of individual `>` tokens followed by a '`.`'.

If an identifier is followed by a sequence of tokens of this kind then lexical analysis marks the construct as a "high precedence type application" and subsequent grammar rules ensure 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 (B(e)).C
```

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.

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 Special Attributes and Types

This chapter documents attributes and types of special significance to the F# compiler.

16.1.1 Custom Attributes Imported 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 F# imported assemblies or in assemblies authored in other CLI languages.

Attribute	Description
`System.ObsoleteAttribute`	Indicate that the construct is obsolete and give a warning or error depending on the settings in the attribute. This attribute may be used in both F# and imported assemblies.
`System.ParamArrayAttribute`	When applied to an argument of a method, indicates that special language rules apply to the processing of method applications that allow the method to act as variable-argument method. This attribute may be used in both F# and imported assemblies.
`System.ThreadStaticAttribute`	Mark a mutable static value in a class as thread static. This attribute may be used in both F# and imported assemblies.
`System.ContextStaticAttribute`	Mark a mutable static value in a class as context static. This attribute may be used in both F# and imported assemblies.
`System.AttributeUsageAttribute`	Indicates the attribute usage targets for an attribute. This attribute may be used in both F# and imported assemblies.
`System.Diagnostics.ConditionalAttribute`	Ensures that calls to the method are only emitted when the corresponding conditional define is made. This attribute may be used in both F# and imported assemblies.
`System. Reflection.AssemblyInformationalVersionAttribute`	Attaches version metadata to the compiled form of the assembly. This attribute may be used in both F# and imported assemblies.
`System. Reflection.AssemblyFileVersionAttribute`	Attaches version metadata to the compiled form of the assembly. This attribute may be used in both F# and

	imported assemblies.
System. Reflection.AssemblyDescriptionAttribute	Attaches metadata to the compiled form of the assembly, e.g. 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	Attaches metadata to the compiled form of the assembly, e.g. 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	Attaches metadata to the compiled form of the assembly, e.g. 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	Attaches metadata to the compiled form of the assembly, e.g. 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	Attaches metadata to the compiled form of the assembly, e.g. the “LegalCopyright” attribute in the Win32 version resource for the assembly. This attribute may be used in both F# and imported assemblies.
System. Reflection. AssemblyProductAttribute	Attaches metadata to the compiled form of the assembly, e.g. the “FileDescription” attribute in the Win32 version resource for the assembly. This attribute may be used in both F# and imported assemblies.
System. Reflection. AssemblyProductAttribute	Attaches metadata to the compiled form of the assembly, e.g. the “FileDescription” attribute in the Win32 version resource for the assembly. This attribute may be used in both F# and imported assemblies.
System. Reflection. AssemblyKeyFileAttribute	Indicates to the F# compiler how to sign an assembly. This attribute may be used in both F# and imported assemblies.
System.Reflection.DefaultMemberAttribute	When applied to a type, indicates the name of the indexer property for that type. This attribute may be used in both F# and imported assemblies.
System.Runtime.CompilerServices.InternalsVisibleTo	Directs an 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.TypeForwardedTo	Indicates a type redirection. This attribute may only be used in imported non-F# assemblies. Its use in F# code is not permitted.
System.Runtime.CompilerServices.ExtensionAttribute	Indicates the compiled form of a C# extension member. This attribute may only be used in imported non-F# assemblies. Its use in F# code is not permitted.
System.Runtime.InteropServices.DllImportAttribute	When applied to a function definition in a module, ignore the implementation of the binding and compile it as a CLI P/Invoke stub declaration. This attribute may be used in both F# and imported assemblies.
System.Runtime.InteropServices.MarshalAsAttribute	When applied to a parameter or return type, indicate the marshalling attribute for a CLI P/Invoke stub declaration. This attribute may be used in both F# and imported assemblies. Note, however, that the specification of "custom" marshallers are not supported by F#.
System.Runtime.InteropServices.InAttribute	When applied to a parameter, indicate the CLI [in] attribute. This attribute may be used in both F# and imported assemblies. However, it has no specific meaning in F# beyond changing the corresponding attribute in the CLI compiled form.
System.Runtime.InteropServices.OutAttribute	When applied to a parameter, indicate the CLI [out] attribute. This attribute may be used in both F# and imported assemblies, but has no specific meaning in F# beyond changing the corresponding attribute in the CLI compiled form.
System.Runtime.InteropServices.OptionalAttribute	When applied to a parameter, indicates the CLI optional attribute. This attribute may be used in both F# and imported assemblies, but has no specific meaning in F# beyond changing the corresponding attribute in the CLI compiled form.
System.Runtime.InteropServices.FieldOffsetAttribute	When applied to a field, indicate the field offset of the underlying CLI field. This attribute may be used in both F# and

	imported assemblies.
System.NonSerializedAttribute	When applied to a field, indicate the not serialized bit should be set for the underlying CLI field. This attribute may be used in both F# and imported assemblies.
System.Runtime.InteropServices.StructLayoutAttribute	Indicate the StructLayout of a CLI type. This attribute may be used in both F# and imported assemblies.
Microsoft.FSharp.Core.AutoOpenAttribute	When applied to an assembly and given a string argument, auto-open the namespace or module when the assembly is referenced. When applied to a module without a string argument, auto-open the module when the enclosing namespace or module is opened. This attribute may only be used in F# assemblies.
Microsoft.FSharp.Core.CompilationRepresentationAttribute	Control the compiled form of an F# Language construct. This attribute may only be used in F# assemblies.
Microsoft.FSharp.Core.DefaultAugmentationAttribute	When applied to an F# union type, indicate that the default CLI-visible augmentation of Is, Get and other members should not be generated for that type. This attribute may be used in F# code.
Microsoft.FSharp.Core.DefaultValueAttribute	When applied to an F# field, mark the field as not-needing-initialization, i.e. it will be initialized to the zero value. The field must be mutable. This attribute may be used in F# code.
Microsoft.FSharp.Core.GeneralizableValueAttribute	When applied to an F# value, indicates that uses of the attribute can give rise to generic code through the process of type inference. For example, Set.empty. The value must normally be a type function whose implementation has no observable side effects. This attribute may only be used in F# assemblies.
Microsoft.FSharp.Core.LiteralAttribute	When applied to a binding, compile the value as a CLI literal. This attribute may only be used in F# assemblies.
Microsoft.FSharp.Core.NoDynamicInvocationAttribute	When applied to a function or member binding, replaces the generated code with a stub that will throw an exception at runtime. Used to replace the default generated implementation of unverifiable inlined members with a

	verifiable stub. This attribute may only be used in F# assemblies.
Microsoft.FSharp.Core.OCamlCompatibilityAttribute	When applied to an F# construct, indicate that the construct is for OCaml compatibility and that its use should give a warning unless OCaml compatibility is enabled. This attribute may only be used in F# assemblies.
Microsoft.FSharp.Core.OverloadIDAttribute	Give a unique name to an F# method in an overload set. This attribute may only be used in F# assemblies.
Microsoft.FSharp.Core.StructAttribute	Indicates that a type is a struct type This attribute may only be used in F# assemblies.
Microsoft.FSharp.Core.ClassAttribute	Indicates that a type is a class type This attribute may only be used in F# assemblies.
Microsoft.FSharp.Core.InterfaceAttribute	Indicates that a type is an interface type This attribute may only be used in F# assemblies.
Microsoft.FSharp.Core.MeasureAttribute	Indicates that a type or generic parameter is a unit of measure definition or annotation This attribute may only be used in F# assemblies.
Microsoft.FSharp.Core.ReferenceEqualityAttribute	When applied to an F# record or union type, controls whether the type should use reference equality for its default equality implementation. This attribute may only be used in F# assemblies.
Microsoft.FSharp.Core.ReflectedDefinitionAttribute	Make the quotation form of a binding is available at runtime through the Microsoft.FSharp.Quotations. Expr.GetReflectedDefinition method. This attribute may only be used in F# assemblies.
Microsoft.FSharp.Core.RequireQualifiedAccess	When applied to an F# module, give a warning if an 'open' is used on that module name. This attribute may only be used in F# assemblies.
Microsoft.FSharp.Core.RequiresExplicitTypeArgumentsAttribute	When applied to an F# value, indicates that the value must be given explicit type arguments when used, e.g. typeof<int>. This attribute may only be used in F# assemblies.

Microsoft.FSharp.Core.StructuralComparisonAttribute	When applied to an F# record or union type, controls whether the type should use structural comparison for its default comparison implementation. This attribute may only be used in F# assemblies.
Microsoft.FSharp.Core.StructuralEqualityAttribute	When applied to an F# record or union type, controls whether the type should use structural equality for its default equality implementation. This attribute may only be used in F# assemblies.

16.1.2 Custom Attributes Emitted by F#

The following custom attributes are emitted by the F# compiler:

Attribute	Descriptions
System.Diagnostics.DebuggableAttribute	May be emitted by an F# compiler to improve debuggability of F# code
System.Diagnostics.DebuggerHiddenAttribute	May be emitted by an F# compiler to improve debuggability of F# code
System.Diagnostics.DebuggerDisplayAttribute	May be emitted by an F# compiler to improve debuggability of F# code
System.Diagnostics.DebuggerBrowsableAttribute	May be emitted by an F# compiler to improve debuggability of F# code
System.Runtime.CompilerServices.CompilationRelaxationsAttribute	May be emitted by an F# compiler to permit extra JIT optimziations
System.Runtime.CompilerServices.CompilerGeneratedAttribute	May be emitted by an F# compiler iun compiled code
System.Reflection.DefaultMemberAttribute	May be emitted by an F# compiler to indicate the name of the indexer property for a class
Microsoft.FSharp.Core.CompilationMappingAttribute	May be emitted by an F# compiler to indicate how a CLI construct relates to an F# source language construct
Microsoft.FSharp.Core.FSharpInterfaceDataVersionAttribute	May be emitted by an F# compiler to indicate the schema number for the embedded binary resource for F#-specific interface and optimization data
Microsoft.FSharp.Core.OptionalArgumentAttribute	May be emitted by an F# compiler to indicate optional arguments to F# members

16.1.3 Custom Attributes Not Recognized by F#

The following custom attributes are defined in some CLI implementations and may, at first sight, seem to have meaning relevant to F#. However do not effect the behaviour of the F# compiler, or their use in F# code may be flagged as an error.

Attribute	Descriptions
`System.Runtime.CompilerServices.DecimalConstantAttribute`	The F# compiler will ignore this attribute. However if used in F# code it will have the effect of making some other .NET languages see a decimal constant as a compile-time literal.
`System.Runtime.CompilerServices.RequiredAttributeAttribute`	Do not use this attribute in F# code, and the F# compiler will ignore it or give an error if it is used
`System.Runtime.InteropServices.DefaultParameterValueAttribute`	Do not use this attribute in F# code, and the F# compiler will ignore it or give an error if it is used
`System.Runtime.InteropServices.UnmanagedFunctionPointerAttribute`	Do not use this attribute in F# code, and the F# compiler will ignore it or give an error if it is used
`System.Runtime.CompilerServices.FixedBufferAttribute`	Do not use this attribute in F# code, and the F# compiler will ignore it or give an error if it is used
`System.Runtime.CompilerServices.UnsafeValueTypeAttribute`	Do not use this attribute in F# code, and the F# compiler will ignore it or give an error if it is used
`System.Runtime.CompilerServices.SpecialNameAttribute`	Do not use this attribute in F# code, and the F# compiler will ignore it or give an error if it is used

16.2 Exceptions Thrown by F# Language Primitives

The following exceptions are thrown by certain F# language or primitive library operations.

Attribute	Descriptions
`System.ArithmeticException`	A base class for exceptions that occur during arithmetic operations, such as System.DivideByZeroException and System.OverflowException
`System.ArrayTypeMismatchException`	Thrown when a store into an array fails because the actual type of the stored element is incompatible with

	the actual type of the array.
System.DivideByZeroException	Thrown when an attempt to divide an integral value by zero occurs.
System.IndexOutOfRangeException	Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array.
System.InvalidCastException	Thrown when an explicit conversion from a base type or interface to a derived type fails at run time.
System.NullReferenceException	Thrown when a null reference is used in a way that causes the referenced object to be required.
System.OutOfMemoryException	Thrown when an attempt to allocate memory (via new) fails.
System.OverflowException	Thrown when an arithmetic operation in a checked context overflows
System.StackOverflowException	Thrown when the execution stack is exhausted by having too many pending method calls; typically indicative of very deep or unbounded recursion.
System.TypeInitializationException	Thrown when a F# initialization code for a type throws an exception, and no catch clauses exists to catch it.

17 The F# Library FSharp.Core.dll

The CLI base library `mscorlib.dll` is referenced by all compilations.

The F# base library `FSharp.Core.dll` is referenced by all compilations.

The following namespaces are automatically opened for all F# code:

```
open Microsoft.FSharp
open Microsoft.FSharp.Core
open Microsoft.FSharp.Core.LanguagePrimitives
open Microsoft.FSharp.Core.Operators
open Microsoft.FSharp.Text
open Microsoft.FSharp.Collections
open Microsoft.FSharp.Core.ExtraTopLevelOperators
```

Additional namespaces may be opened due to the presence of `AutoOpenAttribute` declarations attached to referenced F# DLLs.

See also the online documentation at
http://research.microsoft.com/fsharp/manual/FSharp.Core/Microsoft.FSharp.Core.html

17.1 Basic Types (Microsoft.FSharp.Core)

17.1.1 Basic Type Abbreviations

Type Name	Short 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

17.1.2 Types Accepting Unit of Measure Annotations

Type Name	Short Description
`int8<_>`	Underlying representation `System.SByte`, but accepting a unit of measure
`int16<_>`	Underlying representation `System.Int16`, but accepting a unit of measure
`int32<_>`	Underlying representation `System.Int32`, but accepting a unit of measure
`int64<_>`	Underlying representation `System.Int64`, but accepting a unit of measure
`float32<_>`	Underlying representation `System.Single`, but accepting a unit of measure
`float<_>`	Underlying representation `System.Double`, but accepting a unit of measure
`decimal`	Underlying representation `System.Decimal`, but accepting a unit of measure

17.1.3 nativeptr<_>

In compiled IL code `nativeptr<type>` is represented as `System.IntPtr`, except in method argument or return position, when it is generated as a CLI pointer type `type*`.

Note: CLI point types are rarely used. The unsafe object constructors for the CLI type System.String is one place where pointer types appear in CLI metadata.

Note: You can convert between `System.UIntPtr` and `nativeptr<'T>` using the inlined unverifiable functions in `Microsoft.FSharp.NativeInterop.NativePtr`.

Note: nativeptr<_> compiles in different ways because CLI makes restrictions exist about where pointer types can appear.

17.2 Basic Operators and Functions (Microsoft.FSharp.Core.Operators)

17.2.1 Basic Arithmetic Operators

The following operators are defined in `Microsoft.FSharp.Core.Operators`:

Operator/Function Name	Expression Form	Short Description
`(+)`	`x + y`	Overloaded addition
`(-)`	`x - y`	Overloaded subtraction
`(*)`	`x * y`	Overloaded multiplication
`(/)`	`x / y`	Overloaded division
`(%)`	`x % y`	Overloaded modulus
`(~-)`	`-x`	Checked overloaded unary negation
`not`	`not x`	Boolean negation

17.2.2 Generic Equality and Comparison Operators

The following operators are defined in `Microsoft.FSharp.Core.Operators`:

Operator/Function Name	Expression Form	Short 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

17.2.3 Bitwise manipulation operators

The following operators are defined in `Microsoft.FSharp.Core.Operators`:

Operator/Function Name	Expression Form	Short Description
`(<<<)`	`x <<< y`	Overloaded bitwise shift-left
`(>>>)`	`x >>> y`	Overloaded bitwise arithmetic shift-right
`(^^^)`	`x ^^^ y`	Overloaded bitwise exclusive or
`(&&&)`	`x &&& y`	Overloaded bitwise and
`(|||)`	`x ||| y`	Overloaded bitwise or
`(~~~)`	`~~~x`	Overloaded bitwise negation

17.2.4 Math operators

The following operators are defined in `Microsoft.FSharp.Core.Operators`:

Operator/Function Name	Expression Form	Short 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
`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

17.2.5 Function Pipelining and Composition Operators

The following operators are defined in `Microsoft.FSharp.Core.Operators`:

Operator/Function Name	Expression Form	Short Description
(|>)	`x |> f`	Pipelining
(>>)	`f >> g`	Function composition
(<|)	`f <| x`	Backward pipelining
(<<)	`g << f`	Backward function composition
ignore	`ignore x`	Compute and discard a value

17.2.6 Object Transformation Operators

The following operators are defined in `Microsoft.FSharp.Core.Operators`:

Operator/Function Name	Expression Form	Short Description
box	`box x`	Convert to object representation
hash	`hash x`	Generic hashing operator
sizeof	`sizeof<type>`	Compute the size of a value of the given type
typeof	`typeof<type>`	Compute the `System.Type` representation of the given type
typedefof	`typedefof<type>`	Compute the `System.Type` representation of the given type and calls GetGenericTypeDefinition if this is a generic type.
unbox	`unbox x`	Convert form object representation
ref	`ref x`	Allocate a mutable reference cell
(!)	`!x`	Read a mutable reference cell

17.2.7 The typeof and typedefof Operators

The following operators are defined in `Microsoft.FSharp.Core.Operators`:

Operator/Function Name	Expression Form	Short Description
typeof	`typeof<type>`	Compute the `System.Type` representation of the given type
typedefof	`typedefof<type>`	Compute the `System.Type` representation of the given type and calls GetGenericTypeDefinition if this is a generic type.

17.2.8 Pair Operators

The following operators are defined in `Microsoft.FSharp.Core.Operators`:

Operator/Function Name	Expression Form	Short Description
`fst`	`fst p`	Take the first element of a pair
`snd`	`snd p`	Take the second element of a pair

17.2.9 Exception Operators

The following operators are defined in `Microsoft.FSharp.Core.Operators`:

Operator/Function Name	Expression Form	Short Description
`failwith`	`failwith x`	Raise a `FailureException` exception
`invalid_arg`	`invalid_arg x`	Raise an `ArgumentException` exception
`raise`	`raise x`	Raise an exception
`rethrow`	`rethrow()`	Special operator to raise an exception

17.2.10 Input/Output Handles

The following operators are defined in `Microsoft.FSharp.Core.Operators`:

Operator/Function Name	Expression Form	Short Description
`stdin`	`Stdin`	Computes System.Console.In
`stdout`	`Stdout`	Computes System.Console.Out
`stderr`	`Stderr`	Computes System.Console.Error

17.2.11 Overloaded Conversion Functions

The following operators are defined in `Microsoft.FSharp.Core.Operators`:

Operator/Function Name	Expression Form	Short 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

17.3 Checked Arithmetic Operators

The module `Microsoft.FSharp.Core.Operators.Checked` defines runtime-overflow-checked versions of the following operators:

Operator/Function Name	Expression Form	Short 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

17.4 List and Option Types

17.4.1 The List type

The definition of the F# type `Microsoft.FSharp.Collections.list` is shown below

```
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
```

17.4.2 The Option type

The definition of the F# type `Microsoft.FSharp.Core.option` is shown below

```
[<DefaultAugmentation(false)>]
[<CompilationRepresentation(CompilationRepresentationFlags.UseNullAsTrueValue)>]
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
```

17.5 Lazy Computations (Lazy)

Work in progress. See http://research.microsoft.com/fsharp/manual/namespaces.html

17.6 Asynchronous Computations (Async)

Work in progress. See http://research.microsoft.com/fsharp/manual/namespaces.html

17.7 Mailbox Processing (MailboxProcessor)

Work in progress. See http://research.microsoft.com/fsharp/manual/namespaces.html

17.8 Event Types

Work in progress. See http://research.microsoft.com/fsharp/manual/namespaces.html

17.9 Collection Types (Map,Set)

Work in progress. See http://research.microsoft.com/fsharp/manual/namespaces.html

17.10 Text Formatting (Printf)

Work in progress. See http://research.microsoft.com/fsharp/manual/namespaces.html

17.11 Reflection

Work in progress. See http://research.microsoft.com/fsharp/manual/namespaces.html

17.12 Quotations

Work in progress. See http://research.microsoft.com/fsharp/manual/namespaces.html

17.13 Additional Functions (printfn etc.)

Work in progress. See http://research.microsoft.com/fsharp/manual/namespaces.html