APPENDIX

F# Brief Language Guide

This appendix describes the essential constructs of the F# language in a compact form. You can find a full guide to the F# language in the F# Language Specification on the F# website.

Comments and Attributes

Comments (Chapter 2)

```
// comment

(* comment *)

/// XML doc comment
let x = 1
```

Attaching Attributes (Chapter 16)

```
[<Obsolete("Deprecated at 1.2")>]
type Type =
    ...
[<Conditional("DEBUG")>]
let Function(x) =

[<assembly: Note("argument")>]
do ()
```

Basic Types and Literals

Basic Types and Literals (Chapter 3)

```
sbyte      = System.SByte     76y
byte       = System.Byte      76uy
int16      = System.Int16     76s
uint16     = System.UInt16    76us
int32      = System.Int32     76
uint32     = System.UInt32    76u
int64      = System.Int64     76L
uint64     = System.UInt64    76UL
string     = System.String    "abc", @"c:\etc"
single     = System.Single    3.14f
double     = System.Double    3.14, 3.2e5
char       = System.Char      '7'
nativeint  = System.IntPtr     76n
unativeint = System.UIntPtr     76un
bool       = System.Boolean     true, false
unit       = Microsoft.FSharp.Core.Unit  ()
```

Basic Type Abbreviations

```
int8    = sbyte
uint8   = byte
int     = int32
float32 = single
float   = double
```

Types

Types (Chapter 3 and 5)

```
ident                    Named type
ident<type,...,type>     Type instantiation
type * ... * type        Tuple type
type[]                   Array type
#type                    Flexible type (accepts any
subtype)
'ident                   Variable type
type -> type             Function type
```

Type instantiations can be postfix:`int list`

Patterns and Matching

Patterns (Chapter 3 and 9)

Matching (Chapter 3)

`_`	Wildcard pattern
`literal`	Constant pattern
`ident`	Variable pattern
`(pat, ..., pat)`	Tuple pattern
`[ pat; ...; pat ]`	List pattern
`[| pat; ...; pat |]`	Array pattern
`{ id=pat; ...; id=pat }`	Record pattern
`id(pat, ..., pat)`	Union case pattern
`id expr ... expr (pat, ..., pat)`	Active pattern
`pat | pat`	"Or" pattern
`pat & pat`	"Both" pattern
`pat as id`	Named pattern
`:? type`	Type test pattern
`:? type as id`	Type cast pattern
`null`	Null pattern

```
match expr with
| pat -> expr
...
| pat -> expr
```

Note: Rules of a match may use

```
| pat when expr -> expr
```

Active Patterns (Chapter 9)

```
let (|Tag1|Tag2|)   inp = ...
let (|Tag1|_|)      inp = ...
let (|Tag1|)        inp = ...
```

Functions, Composition, and Pipelining

Function values (Chapter 3)

`fun pat ... pat -> expr`	Function

`function`
`| pat -> expr`
`...`
`| pat -> expr`
	Match function

Application and Pipelining (Chapter 3)

`f x`	Application
`x |> g`	Forward pipe
`f >> g`	Function composition

Binding and Control Flow

Control Flow (Chapter 3 and 4)

`expr` `expr`	Sequencing

`do expr`
`expr`
	Sequencing

`for id = expr to expr do`
`    expr`
	Simple loop

`for pat in expr do`
`    expr`
	Sequence loop

`while expr do`
`    expr`
	While loop

Binding and Scoping (Chapter 3)

`let pat = expr`
`expr`
	Value binding

`let id args = expr`
`expr`
	Function binding

`let rec id args = expr`
`expr`
	Recursive binding

`use pat = expr`
`expr`
	Auto dispose binding

Syntax Forms Without Indentation

```
let pat = expr in expr
while expr do expr done
for pat in expr do expr done
expr ; expr
do expr in expr
```

Exceptions

Exception Handling

`try`
`    expr`
`with`
`    | pat -> expr`
`    | pat -> expr`
	Handling

`try`
`    expr`
`finally`
`    expr`
	Compensation

`use id = expr`	Automatic Dispose

Some Exceptions (Chapter 4)

```
Microsoft.FSharp.Core.FailureException
System.MatchFailureException
System.InvalidArgumentException
System.StackOverflowException
```

Raising Exceptions (Common Forms)

`raise expr`	Throw exception
`failwith expr`	Throw `FailureException`

Catch and Rethrow

```
try expr
with
  | :? ThreadAbortException ->
    printfn "thrown!"
    rethrow ()
```

Tuples, Arrays, Lists, and Collections

Tuples (Chapter 3)

`(expr, ..., expr)`	Tuple
`fst expr`	First of pair
`snd expr`	Second of pair

Arrays (Chapter 4)

`[| expr; ...; expr |]`	Array literal
`[| expr..expr |]`	Range array
`[| comp-expr |]`	Generated array
`Array.create size expr`	Array creation
`Array.init size expr`	Array init
`arr.[expr]`	Lookup
`arr.[expr] <- expr`	Assignment
`arr.[expr..expr]`	Slice
`arr.[expr..]`	Right slice
`arr.[..expr]`	Left slice

See Chapter 4 for multi-dimensional operators.

F# Lists (Chapter 3)

`[ expr; ...; expr ]`	List
`[ expr..expr ]`	Range list
`[ comp-expr ]`	Generated list
`expr :: expr`	List cons
`expr @ expr`	List append

F# Options (Chapter 3)

`None`	No value
`Some(expr)`	With value

Some Other Collection Types

```
System.Collections.Generic.Dictionary
System.Collections.Generic.List
System.Collections.Generic.SortedList
System.Collections.Generic.SortedDictionary
System.Collections.Generic.Stack
System.Collections.Generic.Queue
Microsoft.FSharp.Collections.Set
Microsoft.FSharp.Collections.Map
```

Operators

Overloaded Arithmetic (Chapter 3)

`x + y`	Addition
`x - y`	Subtraction
`x * y`	Multiplication
`x / y`	Division
`x % y`	Remainder/modulus
`-x`	Unary negation

Overloaded Math Operators

```
abs, acos, atan, atan2,
ceil, cos, cosh, exp,
floor, log, log10, pow,
pown, sqrt, sin, sinh,
tan, tanh
```

Overloaded Conversion Operators

```
byte, sbyte, int16, uint16,
int, int32, uint32, int64,
uint64, float32, float, single,
double, nativeint,
unativeint
```

Mutable Locals (Chapter 4)

`let mutable` *`var`* `=` *`expr`*	Declare
`var`	Read
`var` `<-` *`expr`*	Update

Mutable Reference Cells (Chapter 4)

`ref expr`	Allocate
`!expr`	Read
`expr.Value`	Read
`expr := expr`	Assign

Overloaded Bitwise Operators (Chapter 3)

`x >>> y`	Shift right
`x <<< y`	Shift left
`x &&& y`	Bitwise logical and
`x ||| y`	Bitwise logical or
`x ^^^ y`	Bitwise exclusive or
`~~~ x`	Bitwise logical not

Generic Comparison and Hashing

`hash x`	Generic hashing
`x = y`	Generic equality
`x <> y`	Generic inequality
`compare x y`	Generic comparison
`x >= y, x <= y,`	
`x > y, x < y,`	
`min x y, max x y`	

Note: Records, tuples, arrays and unions automatically implement structural equality and hashing (see Chapters 5 and 8).

Indexed Lookup (Chapter 4)

`expr``.[`*`idx`*`]`	Lookup
`expr``.[`*`idx`*`] <-` *`expr`*	Assignment
`expr``.[`*`idx`*`..`*`idx`*`]`	Slice
`expr``.[`*`idx`*`..]`	Right slice
`expr``.[..`*`idx`*`]`	Left slice

See Chapter 4 for multidimensional operators

Booleans

`not expr`	Boolean negation
`expr && expr`	Boolean "and"
`expr || expr`	Boolean "or"

Object-Related Operators and Types

`type obj = System.Object`

`box(x)`	Convert to type `obj`
`unbox<type>(x)`	Extract from type `obj`
`typeof<type>`	Extract `Sytem.Type`
`x :> type`	Static cast to supertype
`x :?> type`	Dynamic cast to subtype

Type Definitions and Objects

Union Types: Chapters 3 and 6

```
type UnionType =
  | TagA of type * ... * type
  | TagB of type * ... * type
```

Constructed Class Types: Chapter 6

```
type ObjectType(args) =
  let internalValue = expr
  let internalFunction args = expr
  let mutable internalState = expr
  member x.Prop1 = expr
  member x.Meth2 args = expr
```

Record Types: Chapters 3 and 6

```
type Record =
  { Field1: type
    Field2: type }
```

Object Expressions: Chapter 6

```
{ new IObject with
    member x.Prop1 = expr
    member x.Meth1 args = expr }

{ new Object() with
    member x.Prop1 = expr
    interface IObject with
      member x.Meth1 args = expr
    interface IWidget with
      member x.Meth1 args = expr }
```

Object Interface Types: Chapter 6

```
type IObject =
  interface ISimpleObject
  abstract Prop1 : type
  abstract Meth2 : type -> type
```

Some Special Members

Implementation Inheritance

```
type ObjectType(args) as x =
  inherit BaseType(expr) as base
```

```
member x.Prop             setter property
  with get() = expr
  and  set v = expr

member x.Item             indexer property
  with get idx = expr
  and  set idx v = expr

static member (+) (x,y) = expr operator
```

Named and Optional Arguments for Members

`member obj.Method(?optArgA)`	Declaring optional arg
`new Object(x=expr, y=expr)`	Call with named args
`obj.Method(optArgA=expr, PropB=expr)`	Call with optional args and properties

Namespaces and Modules

Namespaces: Chapter 7

```
namespace Org.Product.Feature

type TypeOne =
    ...

module ModuleTwo =
    ...
```

Files As Modules: Chapter 7

```
module Org.Product.Feature.Module

type TypeOne =
    ...

module ModuleTwo =
    ...
```

Sequence Expressions and Workflows

Sequence Expressions and Workflows: See Chapters 3 and 9

`[ comp-expr ]`	Generated list
`[| comp-expr |]`	Generated array
`seq { comp-expr }`	Generated sequence

`async { comp-expr }`	Asynchronous workflow
`ident { comp-expr }`	Arbitrary workflow

Syntax for Workflows

`let! pat = expr`
`comp-expr`
	Execute and bind computation

`let pat = expr`
`comp-expr`
	Execute and bind expression

`do! expr`
`comp-expr`
	Execute computation

`do expr`
`comp-expr`
	Execute expression

`if expr then comp-expr else comp-expr`	Conditional workflow
`if expr then comp-expr`	Conditional workflow
`while expr do comp-expr`	Repeated workflow
`for pat in expr do comp-expr`	Enumeration loop
`try comp-expr with pat -> expr`	Workflow with catch
`try comp-expr finally expr`	Workflow with compensation
`use pat = expr in comp-expr`	Workflow with auto dispose
`return expr`	Return expression
`return! expr`	Return computation
`yield expr`	Yield expression (for sequences only)
`yield! expr`	Yield sequence (for sequences only)

Queries and Quotations

F# Queries (Chapter 13)

`query { for x in expr do … }`	Query table
`query { … let v = expr in … }`	Query local
`query { … where expr … }`	Query filtering
`query { … select expr … }`	Query selection
`query { … averageBy/minBy/maxBy/sumBy expr }`	Query statistic
`query { … sortBy/sortByDescending expr }`	Query ordering
`query { … thenBy/thenByDescending expr }`	Query subsequent ordering

`query { … distinct }`	Query unique selection
`query { … count }`	Query count selection
`query { … first/last/exactlyOne }`	Query first/last/unique result
`query { … firstOrDefault/lastOrDefault/exactlyOneOrDefault }`	Query result or default
`query { … exists` *expr* `}`	Predicate satisfied at least once
`query { … all` *expr* `}`	Predicate always satisfied
`query { … skip` *expr* `}`	Query paging
`query { … take` *expr* `}`	Query paging
`query { … distinct }`	Query unique selection
`query { … groupBy` *expr* `}`	Query grouping
`query { … groupBy` *expr* `into` *id* `…}`	Query grouping
`query { … groupValBy` *expr expr* `}`	Query grouping value by key
`query { … groupValBy` *expr expr* `into` *id* `… }`	Query grouping value by key
`query { … join` *expr* `in` *expr* `on (`*expr* `=` *expr*`) }`	Query inner join
`query { … groupJoin` *expr* `in` *expr* `on (`*expr* `=` *expr*`) into` *id* `…}`	Query group join
`query { … leftOuterJoin` *expr* `in` *expr* `on (`*expr* `=` *expr*`) into` *id* `…}`	Query left outer join
+ nullable variations on statistics, sorting and joining operators	

Quotations (Chapter 16)

`<@` *expr* `@>`	Quotation expression
`<@@` *expr* `@@>`	Untyped quotation expression
`%`*expr*	Splice of typed quotation
`%%`*expr*	Splice of untyped quotation
`[<ReflectedDefinition>]`	Include quoted form of definition at runtime