

Appendix A

Crash Course in F#

This chapter introduces parts of the F# programming language as used in this book; Hansen and Rischel [1] give a proper introduction to functional programming with F#. The F# programming language belongs to the ML family, which includes classical ML from 1978, Standard ML [3] from 1986, CAML from 1985, Caml Light [2] from 1990, and OCaml [5] from 1996, where F# resembles the latter most. All of these languages are strict, mostly functional, and statically typed with parametric polymorphic type inference.

In Microsoft Visual Studio 2010, F# is included by default. You can also run F# on MacOS and Linux, using the Mono implementation [4] of CLI/.NET: see “Using F# with Mono” in the README file of the F# distribution.

A.1 Files for This Chapter

File	Contents
Intro/Appendix.fs	All examples shown in this chapter

A.2 Getting Started

To get the F# interactive prompt, open a Visual Studio Command Prompt, then type `fsi` for F# Interactive. `fsi` It allows you to enter declarations and evaluate expressions:

```
Microsoft (R) F# 2.0 Interactive build 4.0.30319.1
Copyright (c) Microsoft Corporation. All Rights Reserved.
For help type #help;;
> let rec fac n = if n=0 then 1 else n * fac(n-1);;
val fac : int -> int
> fac 10;;
```

```
val it : int = 3628800
> #q;;
```

Text starting with an angle symbol (>) is entered by the user; the other lines show the F# system's response. You can also run the same F# Interactive inside Visual Studio 2010 by choosing View > Other Windows > F# Interactive, but that is likely to cause confusion when we start using the F# lexer and parser tools.

A.3 Expressions, Declarations and Types

F# is a mostly-functional language: a computation is performed by evaluating an *expression* such as `3+4`. If you enter an expression in the interactive system, followed by a double semicolon (`;;`) and a newline, it will be evaluated:

```
> 3+4;;
val it : int = 7
```

The system responds with the value (7) as well as the type (`int`) of the expression.

A *declaration* `let v = e` introduces a variable `v` whose value is the result of evaluating `e`. For instance, this declaration introduces variable `res`:

```
> let res = 3+4;;
val res : int = 7
```

After the declaration one may use `res` in expressions:

```
> res * 2;;
val it : int = 14
```

A.3.1 Arithmetic and Logical Expressions

Expressions are built from constants such as `2` and `2.0`, variables such as `res`, and operators such as multiplication (`*`). Figure A.1 summarizes predefined F# operators.

Expressions may involve functions, such as the predefined function `sqrt`. Function `sqrt` computes the square root of a floating-point number, which has type `float`, a 64-bit floating-point number. We can compute the square root of `2.0` like this:

```
> let y = sqrt 2.0;;
val y : float = 1.414213562
```

Floating-point constants must be written with a decimal point (`2.0`) or in scientific notation (`2E0`) to distinguish them from integer constants.

To get help on F#, consult <http://msdn.microsoft.com/fsharp/> where you may find the library documentation and the language specification [7].

Operator	Type	Meaning
<code>f e</code>		Function application
<code>!</code>	<code>'a ref -> 'a</code>	Dereference
<code>-</code>	<code>num -> num</code>	Arithmetic negation
<code>**</code>	<code>float * float -> float</code>	Power (right-assoc.)
<code>/</code>	<code>float * float -> float</code>	Quotient
<code>/</code>	<code>int * int -> int</code>	Quotient, round toward 0
<code>%</code>	<code>int * int -> int</code>	Remainder of int quotient
<code>*</code>	<code>num * num -> num</code>	Product
<code>+</code>	<code>string * string -> string</code>	String concatenation
<code>+</code>	<code>num * num -> num</code>	Sum
<code>-</code>	<code>num * num -> num</code>	Difference
<code>::</code>	<code>'a * 'a list -> 'a list</code>	Cons onto list (right-assoc.)
<code>+</code>	<code>string * string -> string</code>	Concatenate
<code>@</code>	<code>'a list * 'a list -> 'a list</code>	List append (right-assoc.)
<code>=</code>	<code>'a * 'a -> bool</code>	Equal
<code><></code>	<code>'a * 'a -> bool</code>	Not equal
<code><</code>	<code>'a * 'a -> bool</code>	Less than
<code>></code>	<code>'a * 'a -> bool</code>	Greater than
<code><=</code>	<code>'a * 'a -> bool</code>	Less than or equal
<code>>=</code>	<code>'a * 'a -> bool</code>	Greater than or equal
<code>&&</code>	<code>bool * bool -> bool</code>	Logical “and” (short-cut)
<code> </code>	<code>bool * bool -> bool</code>	Logical “or” (short-cut)
<code>,</code>	<code>'a * 'b -> 'a * 'b</code>	Tuple element separator
<code>:=</code>	<code>'a ref * 'a -> unit</code>	Reference assignment

Fig. A.1 Some F# operators grouped according to precedence. Operators at the top have high precedence (bind strongly). For overloaded operators, `num` means `int`, `float` or another numeric type. All operators are left-associative, except `(**)` and `(::)` and `(@)`

You can also use (static) methods from the .NET class libraries, after opening the relevant namespaces:

```
> open System;;
> let y = Math.Sqrt 2.0;;
val y : float = 1.414213562
```

Logical expressions have type `bool`:

```
> let large = 10 < res;;
val large : bool = false
```

Logical expressions can be combined using logical “and” (conjunction), written `&&`, and logical “or” (disjunction), written `||`. Like the similar operators of C, C++, Java and C#, these use short-cut evaluation, so that `&&` will evaluate its right operand only if the left operand is `true` (and dually for `||`):

```
> y > 0.0 && 1.0/y > 7.0;;
```

```
val it : bool = false
```

Logical negation is written `not e`:

```
> not false ;;  
val it : bool = true
```

The `(!)` operator is used for another purpose, as described in Sect. A.12.

Logical expressions are typically used in *conditional expressions*, written `if e1 then e2 else e3`, which correspond to `(e1 ? e2 : e3)` in C or C++ or Java or C#:

```
> if 3 < 4 then 117 else 118;;  
val it : int = 117
```

A.3.2 String Values and Operators

A text string has type `string`. A string constant is written within double quotes `"`). The string concatenation operator `(+)` constructs a new string by concatenating two strings:

```
> let title = "Prof.";;  
val title : string = "Prof."  
> let name = "Lauesen";;  
val name : string = "Lauesen"  
> let junk = "Dear " + title + " " + name + ", You won $$$!";;  
val junk : string = "Dear Prof. Lauesen, You won $$$!"
```

The instance property `Length` on a string returns its length (number of characters):

```
> junk.Length;;  
val it : int = 32
```

and the string index operation `s.[i]` gets the *i*'th character of string *s*, counting from 0.

A.3.3 Types and Type Errors

Every expression has a type, and the compiler checks that operators and functions are applied only to expressions of the correct type. There are no implicit type conversions. For instance, `sqrt` expects an argument of type `float` and thus cannot be applied to the argument expression `2`, which has type `int`. Some F# types are summarized in Fig. A.2; see also Sect. A.10. The compiler complains in case of type

Type	Meaning	Examples
Primitive types		
int	Integer number (32 bit)	0, 12, ~12
float	Floating-point number (64 bit)	0.0, 12.0, ~12.1, 3E~6
bool	Logical	true, false
string	String	"A", "", "den Haag"
char	Character	'A', ' ', ' '
Exception	Exception	Overflow, Fail "index"
Functions (Sects. A.3.4 – A.3.6 , A.9.3 , A.11)		
float -> float	Function from float to float	sqrt
float -> bool	Function from float to bool	isLarge
int * int -> int	Function taking int pair	addp
int -> int -> int	Function taking two ints	addc
Pairs and tuples (Sect. A.5)		
unit	Empty tuple	()
int * int	Pair of integers	(2, 3)
int * bool	Pair of int and bool	(2100, false)
int * bool * float	Three-tuple	(2, true, 2.1)
Lists (Sect. A.6)		
int list	List of integers	[7; 9; 13]
bool list	List of Booleans	[false; true; true]
string list	List of strings	["foo"; "bar"]
Records (Sect. A.7)		
{x : int; y : int}	Record of two ints	{x=2; y=3}
{y:int; leap:bool}	Record of int and bool	{y=2100; leap=false}
References (Sect. A.12)		
int ref	Reference to an integer	ref 42
int list ref	Reference to a list of integers	ref [7; 9; 13]

Fig. A.2 Some monomorphic F# types

errors, and refuses to compile the expression:

```
> sqrt 2;;
    sqrt 2;;
    -----^
stdin(51,6): error FS0001: The type 'int' does not support
any operators named 'Sqrt'
```

The error message points to the argument expression 2 as the culprit and explains that it has type `int` which does not support any function `Sqrt`. It is up to the reader to infer that the solution is to write `2.0` to get a constant of type `float`.

Some arithmetic operators and comparison operators are *overloaded*, as indicated in Fig. A.1. For instance, the plus operator (+) can be used to add two expressions of type `int` or two expressions of type `float`, but not to add an `int` and a `float`. Overloaded operators default to `int` when there are no `float` or `string` or `char` arguments.

A.3.4 Function Declarations

A *function declaration* begins with the keyword `let`. The example below defines a function `circleArea` that takes one argument `r` and returns the value of `Math.pi * r * r`. The function can be applied (called) simply by writing the function name before an argument expression:

```
> let circleArea r = System.Math.PI * r * r;;
val circleArea : float -> float
> let a = circleArea 10.0;;
val a : float = 314.1592654
```

The system infers that the type of the function is `float -> float`. That is, the function takes a floating-point number as argument and returns a floating-point number as result. This is because the .NET library constant `PI` is a floating-point number.

Similarly, this declaration defines a function `mul2` from `float` to `float`:

```
> let mul2 x = 2.0 * x;;
val mul2 : float -> float
> mul2 3.5;;
val it : float = 7.0
```

A function may take any type of argument and produce any type of result. The function `makejunk` below takes two arguments of type `string` and produces a result of type `string`:

```
> let makejunk title name =
    "Dear " + title + " " + name + ", You won $$$!";;
val makejunk : string -> string -> string
> makejunk "Vice Chancellor" "Tofte";;
val it : string = "Dear Vice Chancellor Tofte, You won $$$!"
```

Note that F# is layout-sensitive (like a few other programming languages, such as Haskell and Python). If the second line of the `makejunk` function declaration had no indentation at all, then we would get an error message (but strangely, in this particular case the declaration would still be accepted):

```
> let makejunk title name =
    "Dear " + title + " " + name + ", You won $$$!";;
    ^^^^^^^
stdin(16,1): warning FS0058: Possible incorrect indentation:
val makejunk : string -> string -> string
```

A.3.5 Recursive Function Declarations

A function may call any function, including itself; but then its declaration must start with `let rec` instead of `let`, where `rec` stands for *recursive*:

```
> let rec fac n = if n=0 then 1 else n * fac(n-1);;
val fac : int -> int
> fac 7;;
val it : int = 5040
```

If two functions need to call each other by so-called *mutual recursion*, they must be declared in one declaration beginning with `let rec` and connecting the two declarations by `and`:

```
> let rec even n = if n=0 then true else odd (n-1)
    and odd n = if n=0 then false else even (n-1);;
val even : int -> bool
val odd : int -> bool
```

A.3.6 Type Constraints

As you can see from the examples, the compiler automatically infers the type of a declared variable or function. Sometimes it is good to use an explicit *type constraint* for documentation. For instance, we may explicitly require that the function's argument `x` has type `float`, and that the function's result has type `bool`:

```
> let isLarge (x : float) : bool = 10.0 < x;;
val isLarge : float -> bool
> isLarge 89.0;;
val it : bool = true
```

If the type constraint is wrong, the compiler refuses to compile the declaration. A type constraint cannot be used to convert a value from one type to another as in C. Thus to convert an `int` to a `float`, you must use function `float : int -> float`. Similarly, to convert a `float` to an `int`, use a function such as `floor`, `round` or `ceil`, all of which have type `float -> int`.

A.3.7 The Scope of a Binding

The *scope* of a variable binding is that part of the program in which it is visible. In a local `let`-expression such as `let x = ...` below, the scope of variable `x` is the *body* expression `x * x`. The indentation shows that expression `x * x` belongs to the inner `let`-expression:

```
let r = let x = 9 + 16
        x * x
```

The value of the inner `let`-expression is $(9 + 16) \cdot (9 + 16) = 625$ but the sum is computed only once. The introduction of local variable `x` does not disturb any existing variables, not even variables with the same name. For instance:

```
> let x = 42;;
val x : int = 42                                (* outer x is 42 *)
> let r = let x = 9 + 16
-         x * x;;                               (* inner x is 25 *)
val r : int = 625
> x;;
val it : int = 42                               (* outer x unchanged *)
```

A.4 Pattern Matching

Like all languages in the ML family, but unlike most other programming languages, F# supports *pattern matching*. Pattern matching is performed by an expression of the form `match e with ...`, which consists of the expression `e` whose value must be matched, and a list (...) of match branches. For instance, the factorial function can be defined by pattern matching on the argument `n`, like this:

```
> let rec facm n =
    match n with
    | 0 -> 1
    | _ -> n * facm(n-1);;
val facm : int -> int
```

The patterns in a `match` are tried in order from top to bottom, and the right-hand side corresponding to the first matching pattern is evaluated. For instance, calling `facm 7` will find that 7 does not match the pattern 0, but it does match the *wildcard pattern* (`_`) which matches any value, so the right-hand side `n * facm(n-1)` gets evaluated.

A slightly more compact notation for one-argument function definitions uses the `function` keyword, which combines parameter binding with pattern matching:

```
> let rec faca =
    function
    | 0 -> 1
    | n -> n * faca(n-1);;
val faca : int -> int
```

Pattern matching in the ML languages is similar to, but much more powerful, than `switch`-statements in C/C++/Java/C#, because matches can involve also tuple patterns (Sect. A.5) and algebraic datatype constructor patterns (Sect. A.9) and any combination of these. This makes the ML-style languages particularly useful for

writing programs that process other programs, such as interpreters, compilers, program analysers, and program transformers.

Moreover, ML-style languages, including F#, usually require the compiler to detect both *incomplete* matches and *redundant* matches; that is, matches that either leave some cases uncovered, or that have some branches that are not usable:

```
> let bad1 n =
    match n with
    | 0 -> 1
    | 1 -> 2;;
warning FS0025: Incomplete pattern matches on this expression.
For example, the value '2' may not be covered by the patterns.
> let bad2 n =
    match n with
    | _ -> 1
    | 1 -> 2;;

    | 1 -> 2;;
    -----^
warning FS0026: This rule will never be matched
```

A.5 Pairs and Tuples

A *tuple* has a fixed number of components, which may be of different types. A *pair* is a tuple with two components. For instance, a pair of integers is written simply $(2, 3)$, and its type is `int * int`:

```
> let p = (2, 3);;
val p : int * int = (2, 3)
> let w = (2, true, 3.4, "blah");;
val w : int * bool * float * string = (2, true, 3.4, "blah")
```

A function may take a pair as an argument, by performing pattern matching on the pair pattern (x, y) :

```
> let add (x, y) = x + y;;
val add : int * int -> int
> add (2, 3);;
val it : int = 5
```

In principle, function `add` takes only one argument, but that argument is a pair of type `int * int`. Pairs are useful for representing values that belong together; for instance, the time of day can be represented as a pair of hours and minutes:

```
> let noon = (12, 0);;
val noon : int * int = (12, 0)
> let talk = (15, 15);;
val talk : int * int = (15, 15)
```

Pairs can be nested to any depth. For instance, a function can take a pair of pairs of values as argument:

```
> let earlier ((h1, m1), (h2, m2)) = h1 < h2 || (h1 = h2 && m1 < m2);;
```

The empty tuple is written `()` and has type `unit`. This seemingly useless value is returned by functions that are called for their side effect only, such as `WriteLine` from the .NET class library:

```
> System.Console.WriteLine "Hello!";;
Hello!
val it : unit = ()
```

Thus the `unit` type serves much the same purpose as the `void` return type in C/C++/Java/C#.

A.6 Lists

A *list* contains zero or more elements, all of the same type. For instance, a list may hold three integers; then it has type `int list`:

```
> let x1 = [7; 9; 13];;
val x1 : int list = [7; 9; 13]
```

The empty list is written `[]`, and the operator `(::)` called “cons” prepends an element to an existing list. Hence this is equivalent to the above declaration:

```
> let x2 = 7 :: 9 :: 13 :: [];;
val x2 : int list = [7; 9; 13]
> let equal = (x1 = x2);;
val equal : bool = true
```

The cons operator `(::)` is right associative, so `7 :: 9 :: 13 :: []` reads `7 :: (9 :: (13 :: []))`, which is the same as `[7; 9; 13]`.

A list `ss` of strings can be created just as easily as a list of integers; note that the type of `ss` is `string list`:

```
> let ss = ["Dear"; title; name; "you won $$$!"];;
val ss : string list = ["Dear"; "Prof."; "Lauesen";
                        "you won $$$!"]
```

The elements of a list of strings can be concatenated to a single string using the `String.concat` function:

```
> let junk2 = String.concat " " ss;;
val junk2 : string = "Dear Prof. Lauesen you won $$$!"
```

Functions on lists are conveniently defined using pattern matching and recursion. The `sum` function computes the sum of an integer list:

```
> let rec sum xs =
    match xs with
    | [] -> 0
    | x::xr -> x + sum xr;;
val sum : int list -> int
> let x2sum = sum x2;;
val x2sum : int = 29
```

The `sum` function definition says: The sum of an empty list is zero. The sum of a list whose first element is `x` and whose tail is `xr`, is `x` plus the sum of `xr`.

Many other functions on lists follow the same paradigm:

```
> let rec prod xs =
    match xs with
    | [] -> 1
    | x::xr -> x * prod xr;;
val prod : int list -> int
> let x2prod = prod x2;;
val x2prod : int = 819
> let rec len xs =
    match xs with
    | [] -> 0
    | x::xr -> 1 + len xr;;
val len : 'a list -> int
> let x2len = len x2;;
val x2len : int = 3
> let sslen = len ss;;
val sslen : int = 4
```

Note the type of `len`. Since the `len` function does not use the list elements, it works on all lists regardless of the element type; see Sect. A.10.

The append operator (`@`) creates a new list by concatenating two given lists:

```
> let x3 = [47; 11];;
val x3 : int list = [47; 11]
> let x1x3 = x1 @ x3;;
val x1x3 : int list = [7; 9; 13; 47; 11]
```

The append operator does not copy the list elements, only the “spine” of the left-hand operand `x1`, and it does not copy its right-hand operand at all. In the computer’s memory, the tail of `x1x3` is shared with the list `x3`. This works as expected because lists are *immutable*: One cannot destructively change an element in list `x3` and thereby inadvertently change something in `x1x3`, or vice versa.

Some commonly used F# list functions are shown in Fig. A.3.

A.7 Records and Labels

A *record* is basically a tuple whose components are labelled. Instead of writing a pair (`"Kasper"`, `5170`) of a name and the associated phone number, one can

Function	Type	Meaning
append	'a list -> 'a list -> 'a list	Append lists
exists	('a -> bool) -> 'a list -> bool	Does any satisfy...
filter	('a -> bool) -> 'a list -> 'a list	Those that satisfy...
fold	('r -> 'a -> 'r) -> 'r -> 'a list -> 'r	Fold (left) over list
foldBack	('a -> 'r -> 'r) -> 'r -> 'a list -> 'r	Fold (right) over list
forall	('a -> bool) -> 'a list -> bool	Do all satisfy...
length	'a list -> int	Number of elements
map	('a -> 'b) -> 'a list -> 'b list	Transform elements
nth	'a list -> int -> 'a	Get <i>n</i> 'th element
rev	'a list -> 'a list	Reverse list

Fig. A.3 Some F# list functions, from the List module. The function name must be qualified by List, as in List.append [1; 2] [3; 4]. Some of the functions are polymorphic (Sect. A.10) or higher-order (Sect. A.11.2). For the list operators cons (::) and append (@), see Fig. A.1

use a record. This is particularly useful when there are many components. Before one can create a record value, one must create a record type, like this:

```
> type phonerec = { name : string; phone : int };;
type phonerec =
    {name: string;
     phone: int;}
> let x = { name = "Kasper"; phone = 5170 };;
val x : phonerec = {name = "Kasper";
                    phone = 5170;}
```

Note how the type of a record is written, with colon (:) instead of equals (=) as used in record expressions and values. One can extract the components of a record using a *record component selector*, very similar to field access in Java and C#:

```
> x.name;;
val it : string = "Kasper"
> x.phone;;
val it : int = 5170
```

A.8 Raising and Catching Exceptions

Exceptions can be declared, raised and caught as in C++/Java/C#. In fact, the exception concept of those languages is inspired by Standard ML. An exception declaration declares an exception constructor, of type Exception. A raise expression throws an exception:

```
> exception IllegalHour;;
exception IllegalHour
> let mins h =
    if h < 0 || h > 23 then raise IllegalHour
```

```

        else h * 60;;
val mins : int -> int
> mins 25;;
> [...] Exception of type 'IllegalHourException' was thrown.
   at FSI_0152.mins(Int32 h)
   at <StartupCode$FSI_0153>.$FSI_0153.main@()
stopped due to error

```

A `try-with-expression` (`try e1 with exn -> e2`) evaluates `e1` and returns its value, but if `e1` throws exception `exn`, it evaluates `e2` instead. This serves the same purpose as `try-catch` in C++/Java/C#:

```

> try (mins 25) with IllegalHour -> -1;;
val it : int = -1

```

As a convenient shorthand, one can use the function `failwith` to throw the standard `Failure` exception, which takes a string message as argument. The variant `failwithf` takes as argument a `printf`-like format string and a sequence of arguments, to construct a string argument for the `Failure` exception:

```

> let mins h =
    if h < 0 || h > 23 then failwith "Illegal hour"
    else h * 60;;
val mins : int -> int
> mins 25;;
Microsoft.FSharp.Core.FailureException: Illegal hour
> let mins h =
    if h < 0 || h > 23 then failwithf "Illegal hour, h=%d" h
    else h * 60;;
val mins : int -> int
> mins 25;;
Microsoft.FSharp.Core.FailureException: Illegal hour, h=25

```

A.9 Datatypes

A *datatype*, sometimes called an *algebraic datatype* or *discriminated union*, is useful when data of the same type may have different numbers and types of components. For instance, a person may either be a `Student` who has only a name, or a `Teacher` who has both a name and a phone number. Defining a person datatype means that we can have a list of person values, regardless of whether they are `Students` or `Teachers`. Recall that all elements of a list must have the same type, so without the common person type, we could not mix students and teachers in the same list.

```

> type person =
    | Student of string                                (* name *)
    | Teacher of string * int;;                        (* name and phone no *)
type person =

```

```

    | Student of string
    | Teacher of string * int
> let people = [Student "Niels"; Teacher("Peter", 5083)];;
val people : person list = [Student "Niels";
                             Teacher ("Peter",5083)]
> let getphone1 person =
    match person with
    | Teacher(name, phone) -> phone
    | Student name         -> failwith "no phone";;
val getphone1 : person -> int
> getphone1 (Student "Niels");;
Microsoft.FSharp.Core.FailureException: no phone

```

Multiple type declarations that depend on each other can be connected with the keyword `and`.

A.9.1 The *option* Datatype

A frequently used datatype is the *option* datatype, used to represent the presence or absence of a value.

```

> type intopt =
    | Some of int
    | None;;
type intopt =
    | Some of int
    | None
> let getphone2 person =
    match person with
    | Teacher(name, phone) -> Some phone
    | Student name         -> None;;
val getphone2 : person -> intopt
> ge\texttt{t}phone2 (Student "Niels");;
val it : intopt = None

```

In Java and C#, some methods return `null` to indicate the absence of a result, but that is a poor substitute for an *option* type, both in the case where the method should never return `null`, and in the case where `null` is a legitimate result from the method. The type inferred for function `getphone2` clearly says that we cannot expect it to always return an integer, only an `intopt`, which may or may not hold an integer.

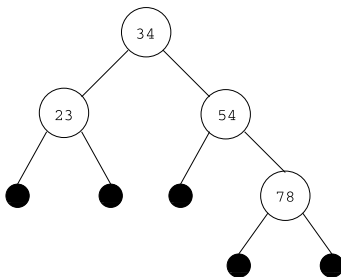
In F#, there is a predefined polymorphic datatype `'a option` with constructors `Some` and `None`; using those instead of `intopt` above, function `getphone2` would have type `person -> int option`.

A.9.2 Binary Trees Represented by Recursive Datatypes

A datatype declaration may be recursive, which means that a value of the datatype t can have a component of type t . This can be used to represent trees and other data structures. For instance, a binary integer tree `inttree` may be defined to be either a leaf `Lf`, or a branching node `Br` that holds an integer and a left subtree and a right subtree:

```
> type inttree =
    | Lf
    | Br of int * inttree * inttree;;
type inttree =
    | Lf
    | Br of int * inttree * inttree
> let t1 = Br(34, Br(23,Lf,Lf), Br(54,Lf,Br(78,Lf,Lf)));;
val t1 : inttree = Br(34,Br(23,Lf,Lf),Br(54,Lf,Br(78,Lf,Lf)))
```

The tree represented by `t1` has 34 at the root node, 23 at the root of the left subtree, and so on, like this, where a solid dot represents an `Lf` value:



Functions on trees and other datatypes are conveniently defined using pattern matching and recursion. This function computes the sum of the nodes of an integer tree:

```
> let rec sumtree t =
    match t with
    | Lf -> 0
    | Br(v, t1, t2) -> v + sumtree t1 + sumtree t2;;
val sumtree : inttree -> int
> let t1sum = sumtree t1;;
val t1sum : int = 189
```

The definition of `sumtree` reads: The sum of a leaf node `Lf` is zero. The sum of a branch node `Br(v, t1, t2)` is v plus the sum of `t1` plus the sum of `t2`.

A.9.3 Curried Functions

A function of type `int * int -> int` that takes a pair of arguments is closely related to a function of type `int -> int -> int` that takes two arguments. The latter is called a *curried* version of the former; this is a pun on the name of logician Haskell B. Curry, who proposed this idea. For instance, function `addc` below is a curried version of function `addp`. Note the types of `addp` and `addc` and how the functions are applied to arguments:

```
> let addp (x, y) = x + y;;
val addp : int * int -> int
> let addc x y = x + y;;
val addc : int -> int -> int
> let res1 = addp(17, 25);;
val res1 : int = 42
> let res2 = addc 17 25;;
val res2 : int = 42
```

A major advantage of curried functions is that they can be partially applied. Applying `addc` to only one argument, 17, we obtain a new function of type `int -> int`. This new function adds 17 to its argument and can be used on as many different arguments as we like:

```
> let addSeventeen = addc 17;;
val addSeventeen : (int -> int)
> let res3 = addSeventeen 25;;
val res3 : int = 42
> let res4 = addSeventeen 100;;
val res4 : int = 117
```

A.10 Type Variables and Polymorphic Functions

We saw in Sect. A.6 that the type of the `len` function was `'a list -> int`:

```
> let rec len xs =
    match xs with
    | [] -> 0
    | x::xr -> 1 + len xr;;
val len : 'a list -> int
```

The `'a` is a *type variable*. Note that the prefixed prime (`'`) is part of the type variable name `'a`. In a call to the `len` function, the type variable `'a` may be instantiated to any type whatsoever, and it may be instantiated to different types at different uses. Here `'a` gets instantiated first to `int` and then to `string`, in two different applications of `len`:

```
> len [7; 9; 13];;
val it : int = 3
```



```
> len ["Oslo"; "Aarhus"; "Gothenburg"; "Copenhagen"];;
val it : int = 4
```

A.10.1 Polymorphic Datatypes

Some data structures, such as a binary trees, have the same shape regardless of the element type. Fortunately, we can define polymorphic datatypes to represent such data structures. For instance, we can define the type of binary trees whose leaves can hold a value of type `'a` like this:

```
> type 'a tree =
    | Lf
    | Br of 'a * 'a tree * 'a tree;;
type 'a tree =
    | Lf
    | Br of 'a * 'a tree * 'a tree
```

Compare this with the monomorphic integer tree type in Sect. A.9.2. Values of this type can be defined exactly as before, but the type is slightly different:

```
> let t1 = Br(34, Br(23,Lf,Lf), Br(54,Lf,Br(78,Lf,Lf)));;
val t1 : int tree = Br(34, Br(23, Lf, Lf), Br(54, Lf, Br(78, Lf, Lf)))
```

The type of `t1` is `int tree`, where the type variable `'a` has been instantiated to `int`.

Likewise, functions on such trees can be defined as before:

```
> let rec sumtree t =
    match t with
    | Lf -> 0
    | Br(v, t1, t2) -> v + sumtree t1 + sumtree t2;;
val sumtree : int tree -> int
> let rec count t =
    match t with
    | Lf -> 0
    | Br(v, t1, t2) -> 1 + count t1 + count t2;;
val count : 'a tree -> int
```

The argument type of `sumtree` is `int tree` because the function adds the node values, which must be of type `int`.

The argument type of `count` is `'a tree` because the function ignores the node values `v`, and therefore works on an `'a tree` regardless of the node type `'a`.

Function `preorder1 : 'a tree -> 'a list` returns a list of the node values in a tree, in *preorder*, that is, the root node comes before the left subtree which comes before the right subtree:

```
> let rec preorder1 t =
    match t with
```

```

    | Lf          -> []
    | Br(v, t1, t2) -> v :: preorder1 t1 @ preorder1 t2;;
val preorder1 : 'a tree -> 'a list
> preorder1 t1;;
val it : int list = [34; 23; 54; 78]

```

A side remark on efficiency: When the left subtree `t1` is large, then the call `preorder1 t1` will produce a long list of node values, and the list append operator (`@`) will be slow. Moreover, this happens recursively for all left subtrees.

Function `preorder2` does the same job in a more efficient, but slightly more obscure way. It uses an auxiliary function `preo` that has an *accumulating parameter* `acc` that gradually collects the result without ever performing an append (`@`) operation:

```

> let rec preo t acc =
    match t with
    | Lf          -> acc
    | Br(v, t1, t2) -> v :: preo t1 (preo t2 acc);;
val preo : 'a tree -> 'a list -> 'a list
> let preorder2 t = preo t [];;
val preorder2 : 'a tree -> 'a list
> preorder2 t1;;
val it : int list = [34; 23; 54; 78]

```

The following relation holds for all `t` and `xs`:

```
preo t xs = preorder1 t @ xs
```

It follows that

```
preorder2 t = preo t [] = preorder1 t @ [] = preorder1 t
```

A.10.2 Type Abbreviations

When a type, such as `(string * int) list`, is used frequently, it is convenient to abbreviate it using a name such as `intenv`:

```

> type intenv = (string * int) list;;
type intenv = (string * int) list
> let bind1 (env : intenv) (x : string, v : int) : intenv =
-   (x, v) :: env;;
val bind1 : intenv -> string * int -> intenv

```

The type declaration defines a *type abbreviation*, not a new type, as can be seen from the compiler's response. This also means that the function can be applied to a perfectly ordinary list of `string * int` pairs:

```

> bind1 [("age", 47)] ("phone", 5083);;
val it : intenv = [("phone", 5083); ("age", 47)]

```

A.11 Higher-Order Functions

A *higher-order function* is one that takes another function as an argument. For instance, function `map` below takes as argument a function `f` and a list, and applies `f` to all elements of the list:

```
> let rec map f xs =
    match xs with
    | []      -> []
    | x::xr  -> f x :: map f xr;;
val map : ('a -> 'b) -> 'a list -> 'b list
```

The type of `map` says that it takes as arguments a function from type `'a` to type `'b`, and a list whose elements have type `'a`, and produces a list whose elements have type `'b`. The type variables `'a` and `'b` may be independently instantiated to any types. For instance, we can define a function `mul2` of type `float -> float` and use `map` to apply that function to all elements of a list:

```
> let mul2 x = 2.0 * x;;
val mul2 : float -> float
> map mul2 [4.0; 5.0; 89.0];;
val it : float list = [8.0; 10.0; 178.0]
```

Or we may apply a function `isLarge` of type `float -> bool` (defined on page 251) to all elements of a float list:

```
> map isLarge [4.0; 5.0; 89.0];;
val it : bool list = [false; false; true]
```

Function `map` is so useful that it is predefined in F#'s `List` module; see Fig. A.3.

A.11.1 Anonymous Functions

Sometimes it is inconvenient to introduce named auxiliary functions. In this case, one can write an anonymous *function expression* using `fun` instead of a named *function declaration* using `let`:

```
> fun x -> 2.0 * x;;
val it : float -> float = <fun:clo@0-1>
```

The expression `(fun x -> ...)` evaluates to a closure, or function value, which can be passed around exactly like any other F# value. This is particularly useful in connection with higher-order functions such as `map`:

```
> map (fun x -> 2.0 * x) [4.0; 5.0; 89.0];;
val it : float list = [8.0; 10.0; 178.0]
> map (fun x -> 10.0 < x) [4.0; 5.0; 89.0];;
val it : bool list = [false; false; true]
```

The function `tw` defined below takes a function closure `g` and an argument `x` and applies `g` twice; that is, it computes `g (g x)`. Using `tw` one can define a function `quad` that applies `mul2` twice, thus multiplying its argument by 4.0:

```
> let tw g x = g (g x);;
val tw : ('a -> 'a) -> 'a -> 'a
> let quad = tw mul2;;
val quad : (float -> float)
> quad 7.0;;
val it : float = 28.0
```

An anonymous function created with `fun` may take any number of arguments. A function that takes two arguments is similar to one that takes the first argument and then returns a new anonymous function that takes the second argument:

```
> fun x y -> x+y;;
val it : int -> int -> int = <fun:clo@0-2>
> fun x -> fun y -> x+y;;
val it : int -> int -> int = <fun:clo@0-3>
```

The difference between `fun` and `function` is that a `fun` can take more than one parameter but can have only one match case, whereas a `function` can take only one parameter but can have multiple match cases. For instance, two-argument `increaseBoth` is most conveniently defined using `fun` and one-argument `isZeroFirst` is most conveniently defined using `function`:

```
> let increaseBoth = fun i (x, y) -> (x+i, y+i);;
val increaseBoth : int -> int * int -> int * int
> let isZeroFirst = function | (0::_) -> true | _ -> false;;
val isZeroFirst : int list -> bool
```

A.11.2 Higher-Order Functions on Lists

Higher-order functions are particularly useful in connection with polymorphic datatypes. For instance, one can define a function `filter` that takes as argument a predicate (a function of type `'a -> bool`) and a list, and returns a list containing only those elements for which the predicate is `true`. This may be used to extract the even elements (those divisible by 2) in a list:

```
> let rec filter p xs =
    match xs with
    | [] -> []
    | x::xr -> if p x then x :: filter p xr
               else filter p xr;;
val filter : ('a -> bool) -> 'a list -> 'a list
> let onlyEven =
    filter (fun i -> i%2 = 0) [4; 6; 5; 2; 54; 89];;
val onlyEven : int list = [4; 6; 2; 54]
```

Note that the filter function is polymorphic in the argument list type. The filter function is predefined in F#'s `List` module; see Fig. A.3. Another very general predefined polymorphic higher-order list function is `foldr`, for *fold right*, which exists in F# under the name `List.foldBack`:

```
> let rec foldr f xs e =
    match xs with
    | []      -> e
    | x::xr   -> f x (foldr f xr e);;
val foldr : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

One way to understand `foldr f xs e` is to realize that it systematically and recursively replaces the list constructors by other operators as follows:

replace	<code>[]</code>	by	<code>e</code>
replace	<code>(x :: xr)</code>	by	<code>f x xr</code>

The `foldr` function presents a general procedure for processing a list, and is closely related to the visitor pattern in object-oriented programming, although this may not appear very obvious.

Many other functions on lists can be defined in terms of `foldr`:

```
> let len xs = foldr (fun _ res -> 1+res) xs 0;;
val len : 'a list -> int
> let sum xs = foldr (fun x res -> x+res) xs 0;;
val sum : int list -> int
> let prod xs = foldr (fun x res -> x*res) xs 1;;
val prod : int list -> int
> let map g xs = foldr (fun x res -> g x :: res) xs [];;
val map : ('a -> 'b) -> 'a list -> 'b list
> let listconcat xss = foldr (fun xs res -> xs @ res) xss [];;
val listconcat : 'a list list -> 'a list
> let stringconcat xss = foldr (fun xs res -> xs+res) xss "";;
val stringconcat : string list -> string
> let filter p xs =
    foldr (fun x r -> if p x then r else x :: r) xs [];;
val filter : ('a -> bool) -> 'a list -> 'a list
```

The functions `map`, `filter`, `fold`, `foldBack` and many others are predefined in the F# `List` module; see Fig. A.3.

A.12 F# Mutable References

A *reference* is a handle to a *memory cell*. A reference in F# is similar to a reference in Java/C# or a pointer in C/C++, but the reference cannot be `null` or uninitialized. Moreover, the memory cell cannot be uninitialized and cannot be accidentally changed by other memory write operations, only through the reference.

A new unique memory cell and a reference to it is created by applying the `ref` constructor to a value. Applying the dereferencing operator (`!`) to a reference returns

the value in the corresponding memory cell. The value in the memory cell can be changed by applying the assignment (`:=`) operator to the reference and a new value:

```
> let r = ref 177;;
val r : int ref = {contents = 177;}
> let v = !r;;
val v : int = 177
> r := 288;;
val it : unit = ()
> !r;;
val it : int = 288
```

A typical use of references and memory cells is to create a sequence of distinct names or labels:

```
> let nextlab = ref -1;;
val nextlab : int ref = {contents = -1;}
> let newLabel () =
-     (nextlab := 1 + !nextlab; "L" + string (!nextlab));;
val newLabel : unit -> string
> newLabel();;
val it : string = "L0"
> newLabel();;
val it : string = "L1"
> newLabel();;
val it : string = "L2"
```

References are used also to implement efficient algorithms with destructive update, such as graph algorithms.

A.13 F# Arrays

An F# array is a zero-based indexable fixed-size collection of mutable elements of a particular type, just like a .NET array, but it uses a different syntax for array creation, indexing and update. The F# array type is generic in its element type:

```
> let arr = [| 2; 5; 7 |];;
val arr : int array = [|2; 5; 7|]
> arr.[1];;
val it : int = 5
> arr.[1] <- 11;;
val it : unit = ()
> arr;;
val it : int array = [|2; 11; 7|]
> arr.Length;;
val it : int = 3
```


Exercise A.1 Define the following functions in F#:

- A function `max2 : int * int -> int` that returns the largest of its two integer arguments. For instance, `max(99, 3)` should give 99.
- A function `max3 : int * int * int -> int` that returns the largest of its three integer arguments.
- A function `isPositive : int list -> bool` so that `isPositive xs` returns true if all elements of `xs` are greater than 0, and false otherwise.
- A function `isSorted : int list -> bool` so that `isSorted xs` returns true if the elements of `xs` appear sorted in non-decreasing order, and false otherwise. For instance, the list `[11; 12; 12]` is sorted, but `[12; 11; 12]` is not. Note that the empty list `[]` and all one-element lists such as `[23]` are sorted.
- A function `count : inttree -> int` that counts the number of internal nodes (Br constructors) in an `inttree`, where the type `inttree` is defined in Sect. A.9.2. That is, `count (Br(37, Br(117, Lf, Lf), Br(42, Lf, Lf)))` should give 3, and `count Lf` should give 0.
- A function `depth : inttree -> int` that measures the depth of a tree, that is, the maximal number of internal nodes (Br constructors) on a path from the root to a leaf. For instance, `depth (Br(37, Br(117, Lf, Lf), Br(42, Lf, Lf)))` should give 2, and `depth Lf` should give 0.

Exercise A.2 Define an F# function `linear : int -> int tree` so that `linear n` produces a right-linear tree with `n` nodes. For instance, `linear 0` should produce `Lf`, and `linear 2` should produce `Br(2, Lf, Br(1, Lf, Lf))`.

Exercise A.3 Sect. A.10.1 presents an F# function `preorder1 : 'a tree -> 'a list` that returns a list of the node values in a tree, in *preorder* (root before left subtree before right subtree).

Now define a function `inorder` that returns the node values in *inorder* (left subtree before root before right subtree) and a function `postorder` that returns the node values in *postorder* (left subtree before right subtree before root):

```
inorder   : 'a tree -> 'a list
postorder : 'a tree -> 'a list
```

Thus if `t` is `Br(1, Br(2, Lf, Lf), Br(3, Lf, Lf))`, then `inorder t` is `[2; 1; 3]` and `postorder t` is `[2; 3; 1]`.

It should hold that `inorder (linear n)` is `[n; n-1; ...; 2; 1]` and `postorder (linear n)` is `[1; 2; ...; n-1; n]`, where `linear n` produces a right-linear tree as in Exercise A.2.

Note that the postfix (or reverse Polish) representation of an expression is just a *postorder list of the nodes in the expression's abstract syntax tree*.

Finally, define a more efficient version of `inorder` that uses an auxiliary function `ino : 'a tree -> 'a list -> 'a list` with an accumulating parameter; and similarly for `postorder`.

References

1. Hansen, M.R., Rischel, H.: Functional Programming Using F#. Cambridge University Press, Cambridge (2013, to appear)
2. Leroy, X.: The Zinc experiment: an economical implementation of the ML language. Rapport Technique 117, INRIA Rocquencourt, France (1990)
3. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised). MIT Press, Cambridge (1997)
4. Mono project: At <http://www.mono-project.com/>
5. OCaml: Home page. At <http://caml.inria.fr/>
6. Syme, D., Granicz, A., Cisternino, A.: Expert F#. Apress (2007)
7. Syme, D., et al.: The F# 2.0 language specification. Tech. rep., Microsoft (2010). At <http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec.html>

Index

Symbols

! (F# dereferencing operator), 266
& (C address operator), 121, 176
'a (F# type variable), 260
* (C dereferencing operator), 120
+ (F# string concatenation), 248
.f (F# component selector), 256
:: (F# list constructor), 254
:= (F# assignment operator), 266
_ (F# wildcard pattern), 252

A

Abelson, Harold, 239
Abstract machine, 21, 155–171
Abstract syntax, 4
Abstract syntax tree, 2, 32
Abstraction principle, 9
Access expression, 124
Accumulating parameter, 202, 262
 and continuation, 205
Ackermann function, 103
Activation record, 143
Actual parameters, 118
ADD instruction, 140
Address operator, 176
Aho, Alfred V., 53, 151
Algebraic datatype, 257
Algol 60 programming language, 150
Allocation in a heap, 177
Ambiguous grammar, 34, 45
and (F# keyword), 251, 258
Anonymous method, 80
Appel, Andrew, 151, 219
append (F# function), 256
Array assignment in Java and C#, 71
ASM Java bytecode engineering library, 159
Association list, 3

AST, *see* abstract syntax tree
Augustsson, Lennart, 72

B

B programming language, 120
Backtracking, 215–218
Backtracking continuation, 216
Backus, John, 52, 150
Backus-Naur Form, 52
Backwards code generation, 226
Backwards compiler, 226
Bacon, David F., 185
Barendregt, Henk, 86
Base pointer, 138
Bauer, Friedrich L., 26, 150
BCEL, *see* Bytecode Engineering Library
BCPL programming language, 120
Bertelsen, Peter, 171
Binding level, 96
Black block, 191
Block, list-C machine heap, 191
Blue block, 191
BNF, *see* Backus-Naur Form
Boehm, Hans, 184
Boehm-Demers-Weiser garbage collector, 184, 185
Börger, Egon, 171
Bottom-up parser, 32
Bound variable occurrence, 15
Bracha, Gilad, 104
Buffer overflow, 177
Burge, William H., 53
Bytecode
 CLI/.NET, 163–165
 JVM, 159
 symbolic, 140
 verification, 162, 163

Bytecode Engineering Library, 159

C

C programming language, 120–123
 tail call, 236
 type declaration, 122, 123
 C# programming language, 176
 array assignment, 71
 collection classes, 71
 dynamic typing, 71
 generic types, 104
 parser generator, 32
 CALL instruction, 140
 Call with current continuation, 214, 215
 Call-by-reference, 119
 Call-by-value, 119
 Call-by-value-return, 119
 call/cc, 214, 215
 Canonical representative, 102
 Capture of variable, 18
 Capture-avoiding substitution, 19
 car (list-C function), 187
 CAR instruction, 190
 cdecl program, 122, 136
 cdr (list-C function), 187
 CDR instruction, 190
 ceil (F# function), 251
 Cheney, C. J., 193
 Chomsky, Noam, 52
 Chomsky hierarchy, 52
 Church, Alonzo, 84, 86
 Church numerals, 85
 CIL, *see* Common Intermediate Language
 Class file, 159–162
 CLI, *see also* Common Language
 Infrastructure
 bytecode, 163–165
 example, 166
 instruction operand types, 164
 instructions, 160
 stack frame, 163
 <clinit> pseudo-method (JVM), 161
 Clojure programming language, 156
 Closed expression, 16
 Closure, 59
 in C#, 80
 in F#, 263
 in Java, 80
 CLR, *see* Common Language Runtime
 Co-variance
 in C#, 108
 in Java, 107
 Co-variant generic type, 106
 CoCo/R parser generator, 32

Cohen, R.M., 171
 Collection classes in Java and C#, 71
 Collector, 178
 Comment
 delimited, 127
 end-line, 126
 Common Intermediate Language, 163
 Common Language Infrastructure, 156,
 163–165
 Common Language Runtime, 156
 Common subexpression elimination, 238
 Compacting garbage collection, 182
 Compilation scheme, 144
 Compiler, 14
 Composite pattern, 6
 concat (F# function), 254
 Concrete syntax, 4
 cons (list-C function), 187
 CONS instruction, 190
 Cons operator (: :) in F#, 254
 Conservative garbage collection, 183, 184
 Constant pool, 159
 Constant propagation, 237
 Context free grammar, 34
 Continuation, 201–222
 and backtracking, 215–218
 and frame stack, 211
 call with current, 214, 215
 failure, 216
 success, 216
 Continuation-passing style (CPS), 204–211
 Contra-variance
 in C#, 108
 in Java, 107
 Contra-variant generic type, 107
 Correspondence principle, 9
 CPL programming language, 120
 CPS, *see* continuation-passing style
 CPS transformation, 206, 219
 CSTI instruction, 140
 CUDA C programming language, 157
 Curried function, 260
 Curry, Haskell B., 260

D

Damas, Luis, 109
 Dangling pointer, 176
 Danvy, Olivier, 219
 Db4objects database system, 80
 Dead code elimination, 238
 Dean, Jeffrey, 81
 DeBruijn index, 20
 Declaration, 246
 Decompile, 168

- Delegate, 80
- delete, deallocation in C++, 177
- Delimited comment, 127
- Dereferencing operator, 120
- Detlefs, David, 184
- Diehl, Stephan, 171
- Dijkstra, Edsger W., 150, 193
- Disassembler
 - ildasm, 163
 - javap, 162
- Discriminated union, 257
- dispose, deallocation in Pascal, 177
- DIV instruction, 140
- Doligez, Damien, 72, 193
- DUP instruction, 140
- Dyadic operator, 2
- Dynamic
 - scope, 64
 - semantics, 4
 - typing, 70
 - in Java and C#, 71, 72
- E**
- Eager evaluation, 72, 82, 83
- ECMAScript programming language, 71
- End-line comment, 126
- Environment, 3
- Environment and store, 117, 118
- EQ instruction, 140
- Error continuation, 208
- exception (F# keyword), 256
- Exception handling
 - in a stack machine, 211–213
 - in an interpreter, 207, 208
- .exe file, 164
- exists (F# function), 256
- Explicit types, 65
- Expression, 116
- Expression statement, 124
- F**
- F# programming language, 1, 245–267
 - and keyword, 251, 258
 - anonymous function, 263
 - append function, 256
 - array, 266
 - ceil function, 251
 - concat function, 254
 - declaration, 246
 - dereferencing, 265
 - exception keyword, 256
 - exists function, 256
 - expression, 246
 - failwith function, 257
 - failwithf function, 257
 - filter function, 256, 264, 265
 - float function, 251
 - floor function, 251
 - fold function, 256
 - foldBack function, 256, 265
 - forall function, 256
 - fun keyword, 250, 263, 264
 - function keyword, 252, 264
 - if keyword, 248
 - incomplete match, 253
 - label, 255
 - Length function, 248
 - length function, 256
 - let keyword, 246, 250, 252
 - let rec keyword, 251
 - list, 254
 - logical and, 247
 - logical expression, 247
 - logical negation, 248
 - logical or, 247
 - map function, 256, 263
 - match keyword, 252
 - module, 35
 - module keyword, 35
 - mutually recursive functions, 251
 - not function, 248
 - nth function, 256
 - open keyword, 247
 - operator, 247, 256
 - pair, 253
 - pattern matching, 252
 - pipe operator, 78
 - polymorphic datatype, 261
 - raise keyword, 256
 - record, 255
 - redundant match, 253
 - ref keyword, 265
 - reference, 265
 - rev function, 256
 - round function, 251
 - sqrt function, 246
 - square root, 246
 - string indexing, 248
 - try keyword, 257
 - tuple, 253
 - type
 - constraint, 251
 - predefined, 249
 - variable, 260
 - type keyword, 255, 257, 262
 - unit keyword, 254
 - wildcard pattern, 252
 - with keyword, 252, 257

WriteLine function, 254
 Factorial (example), 123
 Failure continuation, 208, 216
 failwith (F# function), 257
 failwithf (F# function), 257
 Filinski, Andrzej, 219
 filter (F# function), 256, 264, 265
 Finalizer, 186
 Find operation (union-find), 102
 First-order functional language, 57
 Fischer, Michael J., 219
 float (F# function), 251
 floor (F# function), 251
 fold (F# function), 256
 foldBack (F# function), 256, 265
 forall (F# function), 256
 Ford, Brian, 53
 Formal parameter, 118
 Forth programming language, 26
 Fortran programming language, 116, 119, 150
 Forwarding pointer, 197
 Fpascal (parser combinators for F#), 53
 Fragmentation, 179, 182
 Frame in a stack, 143
 Frame stack, 143

- and continuation, 211
- in JVM, 158

 free, deallocation in C, 177
 Free variable occurrence, 15
 Freelist, 178
 From-space, 181
 fsc F# compiler, 35
 fsi F# interactive, 35, 245
 fslex lexer generator, 33, 36
 fsyacc parser generator, 36
 fun (F# keyword), 250, 263, 264
 function (F# keyword), 252, 264
 Function closure, 59
 Functional programming language, 57

G

Gafter, Neal, 80
 Garbage collection, 175–198

- conservative, 183, 184
- generational, 182
- major, 183
- mark-sweep, 180
- minor, 183
- precise, 183, 184
- slice, 181
- two-space stop-and-copy, 181

 Genealogy of programming languages, 8
 Generalized LL (GLL) parsing, 53
 Generational garbage collection, 182

Generic method, 104
 Generic type, 104–109

- variance
 - in C#, 108
 - in Java, 107

 GETBP instruction, 140
 GETSP instruction, 140
 Ghemawat, Sanjay, 81
 GLL parsing, 53
 gnu.bytecode library, 159
 Goal-directed computation, 215
 Gordon, Michael, 72
 GOTO instruction, 140
 Grammar

- ambiguous, 34, 45
- class, 52
- context free, 34
- rule, 34

 Grammatical structure, 31
 Grey block, 191
 Groovy programming language, 156
 Grune, Dick, 151
 Gudeman, David, 190

H

Hancock, Peter, 109
 Hankin, Chris, 72
 Harper, Robert, 72
 Hartel, Pieter, 171
 Haskell programming language, 72, 83, 176, 250
 Header word, 191
 Heap, 157, 176

- allocation, 177
- and garbage collection, 175

 Hewitt, Carl, 193
 Higher-order function, 77, 263
 Hindley, J. Roger, 109
 Hindley-Milner polymorphism, 109
 Hoare, Tony, 9, 121
 Hopcroft, J.E., 53
 HotSpot JVM implementation, 156
 Hudak, Paul, 72
 Hughes, John, 72
 Hutton, Graham, 53

I

Icon programming language, 215
 if (F# keyword), 248
 IFNZRO instruction, 140
 IFZERO instruction, 140
 illdasm disassembler, 163
 Immutable list, 255
 Imperative programming language, 115

- naive, 115–117
- Implementation language, 14
- Incremental garbage collection, 181
- INCSP instruction, 140
- `<init>` pseudo-method (JVM), 161
- Initialized reference (C++), 131
- Instantiation of type scheme, 95
- Interpreter, 13, 14
 - exception handling, 207, 208
 - in continuation-passing style, 206–211
- Invariant generic type, 106
- IronPython implementation, 156
- IronRuby implementation, 156
- Iterative, 203

J

- Java programming language, 176
 - array assignment, 71
 - collection classes, 71
 - dynamic typing, 71
 - garbage collection, 184
 - generic types, 104
 - parser generator, 32, 50
- Java Virtual Machine, 156–163
- `javac` compiler, 159
- JavaCC lexer and parser generator, 50
- JavaCup parser generator, 32
- `javap` disassembler, 162
- JavaScript programming language, 71
- Javassist library, 159
- Jensen, Jørn, 150
- JIT, *see* just-in-time compilation
- JMangler library, 159
- Johnson, Stephen C., 53
- Johnsson, Thomas, 72
- Johnstone, Adrian, 53
- Jones, Richard, 194
- IRuby implementation, 156
- JScript programming language, 156
- Judgement, 62
- Just-in-time compilation, 169–171
 - tracing, 171
- JVM, *see also* Java Virtual Machine
 - bytecode, 159
 - example, 166
 - verification, 163
 - class file, 159–162
 - example, 161
 - constant pool, 159
 - frame stack, 158
 - instruction operand types, 159
 - instructions, 160
- Jython implementation, 156

K

- Kamin, Sam, 26
- Kennedy, Andrew, 104, 109
- Kernighan, Brian W., 120, 133
- Kleene, Stephen C., 52
- Knuth, Donald E., 53
- KVM JVM implementation, 156

L

- Lam, Monica, 53, 151
- Lambda abstraction, 84
- Lambda calculus, 83–86
- Landin, Peter, 9
- Layout-sensitive language, 250
- Lazy evaluation, 72, 82, 83
- Lazy ML programming language, 72
- LDARGS instruction, 140
- LDI instruction, 140
- Left-recursive grammar rules, 51
- `length` (F# function), 248, 256
- Leroy, Xavier, 72, 104, 239
- `let` (F# keyword), 246, 250, 252
- `let rec` (F# keyword), 251
- Lexeme, 32
- Lexer, 32
 - generator, 32
 - specification, 32–34
 - examples, 38–40, 46–52
- Lexical scope, 64, 176
- Lexical structure, 31
- Lexical token, 32
- Lexing module (F#), 36
- Lieberman, Henry, 193
- Lifetime of a value, 176
- Lindholm, Tim, 171
- LinkedList (example), 71, 105, 106, 161, 177
- Lins, Rafael, 194
- Lisp programming language, 65, 71, 72, 176
- List-C language, 186–189
 - garbage collector, 191–193
 - stack machine, 189, 190
- LL parser, 32
- LLVM compiler infrastructure, 157
- Local subroutine, 164
- `longjmp` function, 214
- Loop invariant computation, 238
- LR parser, 32
- LR(0)-item, 41
- LT instruction, 140
- Łukasiewicz, Jan, 26
- Lvalue, 117, 124

M

- MacQueen, David, 72

- Major garbage collection, 183
- `malloc`, allocation in C, 176
- `map` (F# function), 256, 263
- Mark-sweep garbage collection, 180
- `match` (F# keyword), 252
- McCarthy, John, 72, 193
- Memory leak, 184, 185
- Meta language, 1
- Metadata in bytecode files, 168
- Method, anonymous, 80
- Micro-C language, 123–130
 - abstract syntax, 124
 - backwards compilation, 226–237
 - compilation to stack machine, 144–153
 - compiler functions, 145
 - example programs, 125, 127
 - interpreter, 125, 126
 - stack layout, 143
 - stack machine, 138–144
 - exception handling, 212
- Micro-ML language, 57
 - abstract syntax, 58
 - explicitly typed, 65–68
 - first-order, 57–71
 - interpreter, 60–64
 - higher-order, 81, 82
 - lexer and parser, 46, 47
- Micro-SQL language
 - lexer and parser, 47, 48
- Milner, Robin, 72, 109
- Minor garbage collection, 183
- Miranda programming language, 72
- ML programming language, 72
- `MOD` instruction, 140
- `module` (F# keyword), 35
- Mogensen, Torben, v, 31
- Monomorphic type rules, 94
- `MUL` instruction, 140
- Mutator, 178
- N**
- Nair, Ravi, 171
- Naive imperative programming language, 115–117
- Naive store model, 116
- Natural semantics, 62
- Naur, Peter, 52, 72, 150
- Neshkov, Atanas, 168
- .NET (Microsoft), 156
- Net effect principle, 22, 145
- `new`
 - allocation in C++, 176
 - allocation in Java, 177
 - allocation in Pascal, 176
- New operation (union-find), 102
- Nielson, Flemming, 72
- Nielson, Hanne, 72
- `nil` (list-C expression), 187
- `NIL` instruction, 190
- Nonterminal symbol, 34
- Normal continuation, 207
- `not` (F# function), 248
- `NOT` instruction, 140
- `nth` (F# function), 256
- O**
- Obfuscator, 168
- Object language, 1
- Offset
 - in run-time environment, 20
- Old generation, 183
- `open` (F# keyword), 247
- Operational semantics, 62
- Optimizing compiler
 - for micro-C, 225–237
- Orphan block, 192
- P**
- P-code language, 155
- Packrat parser, 53
- Parallel Virtual Machine, 157
- Parameter, 118
 - passing mechanisms, 118–120
- Parametric polymorphism, 93–103
- Parser, 32
 - combinator, 53
 - generator, 32
 - specification, 32, 34, 35
 - examples, 37, 38, 46–52
- Parsing expression grammar, 53
- Parsing module (F#), 36
- Pascal programming language, 9, 155
- Paulson, Larry C., 53
- PDF, *see* Portable Document Format
- PEG, *see* parsing expression grammar
- Perl programming language, 71, 176
- Peyton Jones, Simon, 72
- Pierce, Benjamin, 109
- Plotkin, Gordon, 219
- Pointer, 120
 - arithmetics, 121
 - dangling, 176
- Polymorphic type, 93–109
- Portable Document Format, 22
- Postfix notation, 21
- Postscript programming language, 22, 71, 117, 155
- Precise garbage collection, 183, 184

Preorder traversal, 261
 Principal type scheme, 99
 PRINTC instruction, 140
 Printezis, Tony, 184
 PRINTI instruction, 140
 Production (in grammar), 34
 Program counter, 138
 Prolog programming language, 109, 215
 Pure functional language, 57
 PVM, *see* Parallel Virtual Machine
 Python programming language, 71, 176, 250

R

Rabbit compiler, 219
 Rabin, Michael O., 52
 raise (F# keyword), 256
 Randell, Brian, 150
 Recursive descent, 48
 Reduce action, 42
 Reduce/reduce conflict, 45
 ref (F# keyword), 265
 Reference counting, 179, 180
 Register allocation, 150
 Regular expression syntax, 33
 Reinhold, Mark, 80
 Rémy, Didier, 109
 Result sequence, 215
 Resumption, 216
 RET instruction, 140
 Return statement, compilation, 148
 rev (F# function), 256
 Reverse Polish notation, 21
 Reynolds, John, 219
 Richards, Martin, 120, 133
 Ritchie, Dennis M., 120, 133
 Robinson, Alan, 109
 Roeder, Lutz, 168
 Root set, 178
 round (F# function), 251
 RTCG, *see* run-time code generation
 Ruby programming language, 71
 Rvalue, 118, 124

S

S-expression, 48
 Samelson, Klaus, 26, 150
 SASL programming language, 72
 Scala programming language, 156
 Scanner, *see* lexer
 Scheme programming language, 71, 72, 176
 Schwartzbach, Michael, 109
 Scope, 14
 dynamic, 64
 lexical, 64

 nested, 14
 static, 14, 64, 176
 Scott, Dana, 52
 Scott, Elisabeth, 53
 Semantic action, 38
 Semantics, 4
 dynamic, 4
 operational, 62
 static, 4
 Semispace, 181
 setcar (list-C function), 187
 SETCAR instruction, 190
 setcdr (list-C function), 187
 SETCDR instruction, 190
 Sethi, Ravi, 53, 151
 setjmp function, 214
 Shift action, 41
 Shift/reduce conflict, 45
 Side effect of expression, 116
 Sierpinski curve, 24
 Simula programming language, 176
 Single-threaded store, 118
 Slice of a garbage collection, 181
 Smalltalk programming language, 176
 Smith, James E., 171
 SML, *see* Standard ML programming language
 SML/NJ implementation, 214, 219
 Source language, 14
 Space leak, 177
 sqrt (F# function), 246
 Stack
 allocation, 175
 frame, 143
 machine, 21
 exception handling, 211–213
 for expressions, 22
 for micro-C, 138–144
 pointer, 138
 Standard ML (SML) programming language,
 72, 176
 Start symbol, 34
 Statement, 116
 Static
 scope, 64, 176
 semantics, 4
 Steele, Guy L., 72, 219
 STI instruction, 140
 STOP instruction, 140
 Stop-and-copy garbage collection, 181
 Store and environment, 117, 118
 Strachey, Christopher, 120, 130, 218
 Fundamental Concepts, 130–133
 String concatenation in F#, 248
 String indexing (F#), 248

Structural operational semantics, 62
 SUB instruction, 140
 Substitution, 17–19
 Success continuation, 207, 216
 Sussman, Gerald, 72, 239
 Sussman, Julie, 239
 SWAP instruction, 140
 Symbolic bytecode, 140
 Syme, Don, 104
 Syntactic structure, 31
 Syntax, 4
 abstract, 4
 concrete, 4

T

Tail call, 203, 204
 and continuations, 213, 214
 definition, 202
 in C programming language, 236
 optimization, 234–237
 Tail position, 203, 204
 Tail-recursive, 203
 Target language, 14
 TCALL instruction, 140, 214
 Tennent, Robert, 9
 Terminal symbol, 34
 Thompson, Ken, 120
 To-space, 181
 Tofte, Mads, vii, 72, 239
 Tokenizer, 39
 Top-down parser, 32
 try (F# keyword), 257
 Turing, Alan, 84
 Turner, David A., 72
 Two-space garbage collection, 181
 Type
 checking, 65–68
 inference, 94–97
 polymorphic, 93–109
 scheme, 95
 variable, 99, 260

type (F# keyword), 255, 257, 262
 Type scheme, 95
 principal, 99

U

Ullman, Jeffrey D., 53, 151
 Union operation (union-find), 102
 Union-find data structure, 102, 103
 unit (F# keyword), 254
 Unmanaged CIL, 164
 Unreachable code, 234

V

Variable capture, 18
 Variable occurrence
 bound, 15
 free, 15
 Variance
 in C#, 108
 in Java, 107
 VB.NET programming language, 156
 Verification of bytecode, 162, 163
 Virtual machine, 155
 VM, *see* virtual machine

W

Wadler, Phil, 72
 Wadsworth, Christopher, 72, 218
 Wand, Mitchell, 219
 White block, 191
 Wildcard in generic type (Java), 107
 Wilson, Paul, 194
 Wirth, Niklaus, 9, 150
 with (F# keyword), 252, 257
 Write barrier, 183
 WriteLine (F# function), 254

Y

Yellin, Frank, 171
 Young generation, 183