

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.

Originally developed on Windows, F# is available also on MacOS and Linux, for instance using the Mono implementation [4] of CLI/.NET.

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, start `fsharp` from the command line.

It allows you to enter declarations and evaluate expressions:

```
F# Interactive for F# 4.1
For help type #help;;
> let rec fac n = if n=0 then 1 else n * fac(n-1);;
val fac : n: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. One can also run F# Interactive inside Microsoft Visual Studio, but that may cause challenges 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
f e		Function application
!	'a ref -> 'a	Dereference
-	num -> num	Arithmetic negation
**	float * float -> float	Power
/	float * float -> float	Quotient
/	int * int -> int	Quotient, round toward 0
%	int * int -> int	Remainder of int quotient
*	num * num -> num	Product
+	string * string -> string	String concatenation
+	num * num -> num	Sum
-	num * num -> num	Difference
::	'a * 'a list -> 'a list	Cons onto list (right-assoc.)
+	string * string -> string	Concatenate
@	'a list * 'a list -> 'a list	List append (right-assoc.)
=	'a * 'a -> bool	Equal
<>	'a * 'a -> bool	Not equal
<	'a * 'a -> bool	Less than
>	'a * 'a -> bool	Greater than
<=	'a * 'a -> bool	Less than or equal
>=	'a * 'a -> bool	Greater than or equal
&&	bool * bool -> bool	Logical “and” (short-cut)
	bool * bool -> bool	Logical “or” (short-cut)
,	'a * 'b -> 'a * 'b	Tuple element separator
:=	'a ref * 'a -> unit	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 (@)

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 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.

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

Fig. A.2 Some monomorphic F# types

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 `System.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 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 $$$!";;

    "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.

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...
iter	('a -> unit) -> 'a list -> unit	Apply to all
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

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 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}tphone2 (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

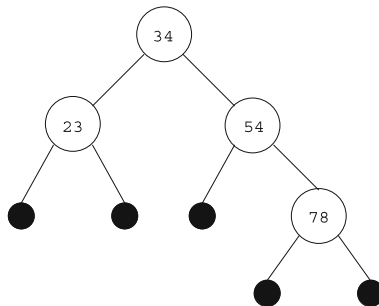
A datatype declaration may be recursive, which means that a value of the datatype `t` can have a component of type `t`. This can be used to represent trees and other data structures. For instance, a binary integer tree `inttree` may be defined to be either a leaf `Lf`, or a branching node `Br` that holds 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 : Br(34,Br(23,Lf,Lf),Br(54,Lf,Br(78,Lf,Lf))) : int tree
```

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 p. 308) 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 []          by e
replace (x :: xr) by f x xr
```

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
```

The .NET method `System.Environment.GetCommandLineArgs()` has type `string array` and holds the command line arguments of an F# when invoked from a command prompt. The element at index 0 is the name of the running executable, the element at index 1 is the first command line argument, and so on (as in C).

- 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 Section 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)
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. <http://www.mono-project.com/>
5. OCaml: Home page. <http://ocaml.org/>
6. Syme, D., Granicz, A., Cisternino, A.: Expert F#. Apress (2007)
7. Syme, D., et al.: The F# 4.0 language specification. Technical report, Microsoft (2016). <http://fsharp.org/specs/language-spec/>

Index

A

Abelson, Harold, 248
Abstract machine, 21, 161–178
Abstract syntax, 4
Abstract syntax tree, 2, 32
Abstraction principle, 9
Access expression, 128
Accumulating parameter, 210, 324
 and continuation, 214
Ackermann function, 107
Activation record, 147
Actual parameters, 122
ADD instruction, 145
Address operator, 184
add (x86 instruction), 287
‘a (F# type variable), 323
AH register (x86), 285
Aho, Alfred V., 54, 155
Algebraic datatype, 320
Algol 60 programming language, 155
Allocation in a heap, 185
AL register (x86), 285
Ambiguous grammar, 34, 45
and (F# keyword), 312, 320
and (x86 instruction), 287
Anonymous method, 85
Appel, Andrew, 155, 227
append (F# function), 317
Array assignment in Java and C#, 73
ASM Java bytecode engineering library, 165
Assembly code, 283–290
Association list, 3
AST, *see* abstract syntax tree
Augustsson, Lennart, 74
Autocode (early assembler), 303

B

Backtracking, 223–226
Backtracking continuation, 225
Backus, John, 53, 155
Backus-Naur Form, 53
Backwards code generation, 234
Backwards compiler, 234
Bacon, David F., 193
Barendregt, Henk, 91
Base pointer, 142
Bauer, Friedrich L., 26, 155, 303
BCEL, *see* Bytecode Engineering Library
BCPL programming language, 124
Bertelsen, Peter, 177
BESK computer, 302
Binding level, 100
Black block, 199
Block, list-C machine heap, 199
Blue block, 199
BNF, *see* Backus-Naur Form
Boehm-Demers-Weiser garbage collector,
 192, 193
Boehm, Hans, 192
Böhm, Corrado, 178
Börger, Egon, 173
Bottom-up parser, 32
Bound variable occurrence, 15
B programming language, 124
Bracha, Gilad, 108
Buffer overflow, 185
Burge, William H., 54
Bytecode
 CLI/.NET, 169–172
 JVM, 165
 symbolic, 145, 269
 verification, 162
Bytecode engineering library, 165

C

- & (C address operator), [124](#), [184](#)
- Call-by-reference, [122](#)
- Call-by-value, [122](#)
- Call-by-value-return, [122](#)
- call/cc, [223–224](#)
- Calling convention
 - C, [289](#)
 - micro-C for x86, [291](#)
- CALL instruction, [144](#)
- Call with current continuation, [223–224](#)
- call (x86 instruction), [287](#)
- Canonical representative, [106](#)
- Capture of variable, [18](#)
- Capture-avoiding substitution, [18](#)
- CAR instruction, [198](#)
- car (list-C function), [195](#)
- Carry flag CF (x86), [287](#)
- C calling convention, [289](#)
- cdecl program, [122](#), [140](#)
- * (C dereferencing operator), [124](#)
- cdq (x86 instruction), [287](#)
- CDR instruction, [197](#)
- cdr (list-C function), [195](#)
- Cheney, C. J., [202](#)
- Chomsky hierarchy, [53](#)
- Chomsky, Noam, [53](#)
- Church numerals, [90](#)
- Church, Alonzo, [88](#), [87](#)
- CIL, *see* Common Intermediate Language
- Class file, [165–167](#)
- CLI, *see also* Common Language Infrastructure
 - bytecode, [169–174](#)
 - example, [173](#)
 - instruction operand types, [170](#)
 - instructions, [166](#)
 - stack frame, [169](#)
- <clinit> pseudo-method (JVM), [167](#)
- Clojure programming language, [162](#)
- Closed expression, [16](#)
- Closure, [61](#)
 - in C#, [85](#)
 - in F#, [326](#)
- CLR, *see* Common Language Runtime
- cmp (x86 instruction), [287](#)
- CoCo/R parser generator, [34](#)
- Collection classes in Java and C#, [73](#)
- Collector, [186](#)
- Comment
 - delimited, [132](#)
 - end-line, [130](#)
- Common intermediate language, [169](#)

- Common language infrastructure, [162](#), [169–172](#)
- Common language runtime, [162](#)
- Common subexpression elimination, [246](#)
- Compacting garbage collection, [190](#)
- Compilation scheme, [149](#), [296](#)
- Compiler, [14](#)
- Composite pattern, [6](#)
- concat (F# function), [317](#)
- Concrete syntax, [4](#)
- cons instruction, [198](#)
- cons (list-C function), [195](#)
- Cons operator (::) in F#, [316](#)
- Conservative garbage collection, [192](#)
- Constant pool, [167](#)
- Constant propagation, [245](#)
- Context free grammar, [34](#)
- Continuation, [209–230](#)
 - and backtracking, [223–226](#)
 - and frame stack, [220](#)
 - call with current, [222–223](#)
 - failure, [225](#)
 - success, [225](#)
- Continuation-Passing Style (CPS), [212–219](#)
- Contra-variant generic type, [110](#)
 - in C#, [112](#)
 - in Java, [111](#)
- Correspondence principle, [9](#)
- Co-variant generic type, [110](#)
 - in C#, [112](#)
 - in Java, [111](#)
- CPL programming language, [124](#)
- C programming language, [124–127](#)
 - tail call, [244](#)
 - type declaration, [128–129](#)
- C# programming language, [184](#)
 - array assignment, [73](#)
 - collection classes, [73](#)
 - dynamic typing, [70](#), [71](#)
 - generic types, [108](#)
 - parser generator, [32](#)
- CPS, *see* continuation-passing style
- CPS transformation, [214](#), [227](#)
- CSTI instruction, [144](#)
- CUDA C programming language, [163](#)
- Curried function, [322](#)
- Curry, Haskell, [322](#)

D

- Damas, Luis, [111](#)
- Dangling pointer, [184](#)
- Danvy, Olivier, [227](#)

- DASK computer, 302
- Dead code elimination, 247
- Dean, Jeffrey, 86
- DeBruijn index, 20
- Declaration, 308
- Decompile, 175
- Delegate, 85
- delete, deallocation in C++, 185
- Delimited comment, 131
- Dereferencing operator, 124
- Detlefs, David, 193
- Diehl, Stephan, 176
- Dijkstra, Edsger W., 155, 202
- Disassembler
 - ildasm, 170
 - javap, 167
- Discriminated union, 320
- dispose, deallocation in Pascal, 185
- DIV instruction, 144
- Doligez, Damien, 74, 202
- Doubleword (x86 terminology), 284
- DUP instruction, 144
- Dyadic operator, 2
- Dynamic
 - scope, 67
 - semantics, 4
 - typing, 72
 - in Java and C#, 73–74

E

- Eager evaluation, 75, 87–88
- EAX register (x86), 286
- EBP base pointer register (x86), 286
- EBX register (x86), 286
- ECMAScript programming language, 73
- ECX register (x86), 286
- EDI register (x86), 286
- EDSAC computer, 302, 303
- EDX register (x86), 286
- EFLAGS status register (x86), 286
- EIP instruction pointer register (x86), 286
- End-line comment, 130
- Environment, 3
- Environment and store, 121–122
- EQ instruction, 144
- Error continuation, 216
- ESI register (x86), 286
- ESP stack pointer register (x86), 286
- Exception handling
 - in a stack machine, 220–221, 278
 - in an interpreter, 215–217
- .exe file, 170

- exists (F# function), 317
- Explicit types, 68
- Expression, 120
- Expression statement, 128

F

- Factorial (example), 128
- Failure continuation, 217, 225
- failwith (F# function), 319
- failwithf (F# function), 319
- := (F# assignment operator), 328
- ! (F# dereferencing operator), 328
- .f (F# component selector), 318
- Fibonacci function, 29, 303
- Filinski, Andrzej, 227
- filter (F# function), 317, 327
- Finalizer, 194
- Find operation (union-find), 106
- First-order functional language, 59
- Fischer, Michael J., 227
- :: (F# list constructor), 316
- float (F# function), 313
- floor (F# function), 313
- fold (F# function), 317
- foldBack (F# function), 317, 327
- forall (F# function), 317
- Ford, Brian, 55
- Formal parameter, 123
- Forth programming language, 28
- Fortran programming language, 120, 123, 155
- Forwarding pointer, 205
- Fparsec (parser combinators for F#), 55
- F# programming language, 1, 307–339
 - and keyword, 312, 320
 - anonymous function, 326
 - append function, 317
 - array, 329
 - ceil function, 313
 - concat function, 316
 - declaration, 308
 - dereferencing, 328
 - exception keyword, 319
 - exists function, 317
 - expression, 308
 - failwith function, 319
 - failwithf function, 319
 - filter function, 317
 - float function, 313
 - floor function, 313
 - fold function, 317
 - foldBack function, 327

- forall function, 317
 - fun keyword, 326, 312, 326
 - function keyword, 314, 326
 - if keyword, 309
 - incomplete match, 306
 - iter function, 317
 - label, 317
 - length function, 310, 317
 - let keyword, 308, 312, 313
 - let rec keyword, 312
 - list, 315
 - logical and, 309
 - logical expression, 309
 - logical negation, 309
 - logical or, 309
 - map function, 317, 325
 - match keyword, 313
 - module, 35
 - module keyword, 35
 - mutually recursive functions, 313
 - not function, 309
 - nth function, 317
 - open keyword, 308
 - operator, 309, 317
 - pair, 315
 - pattern matching, 313
 - pipe operator, 82
 - polymorphic datatype, 323
 - raise keyword, 319
 - record, 317
 - redundant match, 314
 - ref keyword, 328
 - reference, 317
 - rev function, 313
 - round function, 308
 - sqrt function, 314
 - square root, 308
 - string indexing, 310
 - try keyword, 319
 - tuple, 315
 - type
 - constraint, 313
 - predefined, 311
 - variable, 323
 - type keyword, 317, 320, 325
 - unit keyword, 315
 - wildcard pattern, 314
 - with keyword, 313, 319
 - WriteLine function, 315
 - Fragmentation, 187, 190
 - Frame in a stack, 147
 - Frame stack, 147
 - and continuation, 219
 - in JVM, 164
 - free, deallocation in C, 185
 - Free variable occurrence, 15
 - Freelist, 187
 - From-space, 187
 - fsharpc F# compiler, 45
 - fsharpfi F# interactive, 35, 36, 307
 - fslex lexer generator, 33, 36
 - + (F# string concatenation), 310
 - fsyacc parser generator, 36
 - Functional programming language, 59
 - Function closure, 65
 - function (F# keyword), 314, 326
 - fun (F# keyword), 312, 326
 - _ (F# wildcard pattern), 314
- G**
- Garbage collection, 183–207
 - conservative, 192
 - generational, 191
 - major, 191
 - mark-sweep, 188
 - minor, 191
 - precise, 192
 - slice, 189
 - two-space stop-and-copy, 190
 - Genealogy of programming languages, 8
 - Generalized LL (GLL) parsing, 54
 - Generational garbage collection, 190
 - Generic method, 108
 - Generic type, 108–110
 - variance
 - in C#, 112–113
 - in Java, 113
 - GETBP instruction, 144
 - GETSP instruction, 144
 - Ghemawat, Sanjay, 86
 - gnu.bytecode library, 165
 - Goal-directed computation, 224
 - Gordon, Michael, 74
 - GOTO instruction, 144
 - Grammar
 - ambiguous, 35, 48
 - class, 53
 - context free, 34
 - rule, 34
 - Grammar micro-sml, 253
 - Grammatical structure, 31
 - Grey block, 200
 - Groovy programming language, 162
 - Grune, Dick, 161
 - Gudeman, David, 199

H

Hallenberg, Niels, v, 253
 Hancock, Peter, 114
 Hankin, Chris, 75
 Harper, Robert, 74
 Hartel, Pieter, 176
 Haskell programming language, 88, 184, 313
 Header word, 200
 Heap, 163, 184
 allocation, 185
 and garbage collection, 183
 Hewitt, Carl, 202
 Higher-order function, 81
 Hindley, J. Roger, 114
 Hindley–Milner polymorphism, 114
 Hoare, Tony, 9, 126
 Hopcroft, J.E., 54
 HotSpot JVM implementation, 162
 Hudak, Paul, 75
 Hughes, John, 75
 Hutton, Graham, 54

I

IBM 701 computer, 303
 Icon programming language, 224
`idiv` (x86 instruction), 287
`if` (F# function), 309
`IFNZRO` instruction, 144
`IFZERO` instruction, 144
`ildasm` disassembler, 170
 Immutable list, 317
 Imperative programming language, 119
 naive, 119–121
 Implementation language, 13
`imul` (x86 instruction), 287
 Incremental garbage collection, 189
`INCSP` instruction, 144
 Initialized reference (C++), 135
`<init>` pseudo-method (JVM), 168
 Instantiation of type scheme, 99
 80386 Intel processor, 284
 Intel x86 processor family, 284
 Interpreter, 13
 exception handling, 215–217
 in continuation-passing style, 215–220
 Invariant generic type, 110
 IronPython implementation, 162
 IronRuby implementation, 162
 Itanium processor, 303
`iter` (F# function), 317
 Iterative, 211

J

JavaCC lexer and parser generator, 50
`javac` compiler, 165
 JavaCup parser generator, 32
`javap` disassembler, 168
 Java programming language, 184
 array assignment, 73
 collection classes, 74
 dynamic typing, 73, 74
 garbage collection, 192
 generic types, 108
 parser generator, 32, 50
 JavaScript programming language, 73
 Javassist library, 167
 Java Virtual Machine, 162–169
 Jensen, Jørn, 155
 JIT, *see* just-in-time compilation
`jmp` (x86 instruction), 287
`jnz` (x86 instruction), 287
 Johnson, Stephen C., 54
 Johnsson, Thomas, 75
 Johnstone, Adrian, 54
 Jones, Neil D., vii
 Jones, Richard, 202
 JRuby implementation, 162
 JScript programming language, 162
 Judgement, 64
 Just-in-time compilation, 176–177
 tracing, 178
 JVM, *see also* Java Virtual Machine
 bytecode, 165
 example, 171
 verification, 169
 class file, 165–168
 example, 167
 constant pool, 167
 frame stack, 164
 instruction operand types, 165
 instructions, 165
 Jython implementation, 162
`jz` (x86 instruction), 287

K

Kamin, Sam, 26
 Kennedy, Andrew, 108, 113
 Kernighan, Brian W., 137
 Kleene, Stephen C., 53
 Knuth, Donald E., 53, 303
 Kokholm, Niels, vii
 KVM JVM implementation, 162

L

- Lambda abstraction, 88
- Lambda calculus, 88–90
- Lam, Monica, 54, 156
- Landin, Peter, 9
- Layout-sensitive language, 312
- Lazy evaluation, 75, 87–89
- Lazy ML programming language, 75
- LDARGS instruction, 145
- LDI instruction, 145
- lea (x86 instruction), 287, 288
- Left-recursive grammar rules, 52
- Length (F# function), 310, 318
- Leroy, Xavier, 75, 108, 248
- let (F# keyword), 308, 312, 314
- let rec (F# keyword), 313
- Lexeme, 32
- Lexer, 32
 - generator, 32
 - specification, 32–34
 - examples, 40–41, 46–53
- Lexical scope, 67, 184
- Lexical structure, 31
- Lexical token, 32
- Lexing module (F#), 37
- Lieberman, Henry, 202
- Lifetime of a value, 184
- Lindholm, Tim, 177
- LinkedList (example), 75, 167, 185
- Lins, Rafael, 203
- Lisp programming language, 67, 73, 74, 184
- List-C language, 195–198
 - garbage collector, 200–202
 - stack machine, 198–199
- LL parser, 33
- LLVM compiler infrastructure, 163
- Local subroutine, 170
- longjmp function, 223
- Loop invariant computation, 247
- LR(0)-item, 41
- LR parser, 32
- LT instruction, 145
- Łukasiewicz, Jan, 26
- Lvalue, 122, 128

M

- MacQueen, David, 75
- Major garbage collection, 191
- malloc, allocation in C, 185
- Manchester Mark I computer, 303
- map (F# function), 325
- Mark-sweep garbage collection, 188

- McCarthy, John, 75, 202
- Memory leak, 192, 194
- Metadata in bytecode files, 175
- Meta language, 1
- Method, anonymous, 82
- Micro-C language, 127–134
 - abstract syntax, 129
 - backwards compilation, 234–245
 - calling convention for x86, 291
 - compilation for x86, 294–304
 - compilation to stack machine, 149–158
 - compiler functions, 149, 294
 - example programs, 131, 132
 - interpreter, 130–131
 - stack layout, 148
 - stack machine, 142–148
 - exception handling, 221
- Micro-ML language, 59
 - abstract syntax, 60, 256, 258, 259
 - explicitly typed, 68–70
 - first-order, 59–74
 - interpreter, 63–67
 - higher-order, 76–77
 - lexer and parser, 48
- Micro-SML language
 - compilation to stack machine, 280
 - compiler functions, 267
 - stack layout, 269
 - stack machine, 267
 - exception handling, 277
- Micro-SQL language
 - lexer and parser, 48
- Milner, Robin, 74, 114
- Minor garbage collection, 191
- Miranda programming language, 74
- ML programming language, 74
- MOD instruction, 144
- module (F# keyword), 36
- Mogensen, Torben, 31
- Monomorphic type rules, 98
- mov (x86 instruction), 287, 288
- MUL instruction, 144
- Mutator, 186

N

- Nair, Ravi, 177
- Naive imperative programming language, 120–122
- Naive store model, 121
- Nasm x86 assembler, 289–291
- Natural semantics, 64
- Naur, Peter, 53, 74, 172

Neshkov, Atanas, 175
 Net effect principle, 22, 148
 .NET (Microsoft), 179
 Netwide Assembler (Nasm, x86), 289–291
 new
 allocation in C++, 185
 allocation in Java, 185
 allocation in Pascal, 185
 New operation (union-find), 107
 Nielson, Flemming, 74
 Nielson, Hanne, 74
 NIL instruction, 195
 nil (list-C expression), 191
 Nonterminal symbol, 37
 Normal continuation, 216
 NOT instruction, 210
 not (F# function), 309
 nth (F# function), 317

O

Obfuscator, 175
 Object language, 1
 OCaml, 124
 Offset in run-time environment, 21
 Old generation, 191
 open (F# keyword), 308
 Operational semantics, 70
 Optimizing compiler
 for micro-C, 233–245
 Orphan block, 218
 Overflow flag OF (x86), 285

P

Packrat parser, 54
 Parallel Virtual Machine, 163
 Parameter, 122
 passing mechanisms, 122–123
 Parametric polymorphism, 97–107
 Parser, 32
 combinator, 54
 generator, 32
 specification, 32, 34–35
 examples, 37–38, 46–52
 Parsing expression grammar, 54
 Parsing module (F#), 36
 Pascal programming language, 9, 161
 Paulson, Larry C., 54
 P-code language, 161
 PDF, *see* Portable Document Format
 PEG, *see* Parsing expression grammar
 Perl programming language, 73, 184

Peyton Jones, Simon, 74
 Pierce, Benjamin, 113
 Plotkin, Gordon, 227
 Pointer, 125
 arithmetics, 125
 dangling, 184
 Polymorphic type, 97–113
 popad (x86 instruction), 287
 POPHDLR instruction, 278
 pop (x86 instruction), 287
 Portable Document Format, 23
 Postfix notation, 21
 Postscript programming language, 23, 73,
 121, 161
 Precise garbage collection, 192
 Preorder traversal, 324
 Principal type scheme, 103
 PRINTC instruction, 144
 Printezis, Tony, 193
 PRINTI instruction, 144
 Production (in grammar), 34
 Program counter, 142
 Prolog programming language, 114, 224
 Pure functional language, 59
 pushad (x86 instruction), 287
 PUSHHDLR instruction, 278
 PUSHLAB instruction, 270
 push (x86 instruction), 287
 PVM, *see* Parallel Virtual Machine
 Python programming language, 73, 189

Q

Quadword (x86 terminology), 284

R

Rabbit compiler, 227
 Rabin, Michael O., 53
 raise (F# keyword), 310
 Randell, Brian, 155
 Real machine code, 283–290
 Recursive descent, 48
 Reduce action, 43
 Reduce/reduce conflict, 46
 Reference counting, 187–188
 ref (F# function), 328
 Register allocation, 155
 Regular expression syntax, 33
 Rémy, Didier, 113
 Result sequence, 224
 Resumption, 225
 RET instruction, 144
 Return statement, compilation, 150, 295

ret (x86 instruction), 287
 rev (F# function), 317
 Reverse Polish notation, 21
 Reynolds, John, 227
 Richards, Martin, 124, 137
 Ritchie, Dennis M., 123, 137
 Robinson, Alan, 113
 Roeder, Lutz, 175
 Root set, 186
 round (F# function), 313
 RTCG, *see* run-time code generation
 Ruby programming language, 73
 Rvalue, 121, 127

S

sal (x86 instruction), 287
 Samelson, Klaus, 26, 155
 SASL programming language, 75
 Scala programming language, 162
 Scanner, *see* Lexer
 Scheme programming language, 73, 74, 184
 Schwartzbach, Michael, 114
 Scope, 14
 dynamic, 67
 lexical, 66
 nested, 15
 static, 14, 66, 184
 Scott, Dana, 53
 Scott, Elisabeth, 54
 Semantic action, 38
 Semantics, 5
 dynamic, 5
 operational, 64
 static, 5
 Semispace, 189
 SETCAR instruction, 198
 setcar (list-C function), 195
 SETCDR instruction, 198
 setcdr (list-C function), 195
 sete (x86 instruction), 287
 setg (x86 instruction), 287
 setge (x86 instruction), 287
 Sethi, Ravi, 54, 156
 setjmp function, 223
 setl (x86 instruction), 287
 setle (x86 instruction), 287
 setne (x86 instruction), 287
 S-expression, 49
 Shift action, 42
 Shift/reduce conflict, 45
 Side effect of expression, 121
 Sierpinski curve, 24

Sign flag SF (x86), 286
 Simula programming language, 178
 Single-threaded store, 122
 Slice of a garbage collection, 189
 Smalltalk programming language, 184
 Smith, James E., 177
 SML, *see* Standard ML programming language
 SML/NJ implementation, 223, 228
 Source language, 14
 Space leak, 185
 sqrt (F# function), 308
 Stack
 allocation, 183
 frame, 147
 machine, 21
 exception handling, 220–221, 278
 for expressions, 23
 for micro-C, 142–149
 pointer, 142
 Standard ML (SML) programming language, 74, 124, 184
 Start symbol, 34
 Statement, 120
 Static
 scope, 66, 184
 semantics, 5
 Steele, Guy L., 74, 228
 STI instruction, 144
 Stop-and-copy garbage collection, 189
 STOP instruction, 144
 Store and environment, 121–122
 Strachey, Christopher, 124, 134, 227
 Fundamental Concepts, 134–137
 String concatenation in F#, 310
 String indexing (F#), 310
 Structural operational semantics, 64
 SUB instruction, 144
 Substitution, 17–19
 sub (x86 instruction), 287, 288
 Success continuation, 216, 225
 Sussman, Gerald, 74, 248
 Sussman, Julie, 248
 SWAP instruction, 144
 Symbolic bytecode, 144, 270
 Syme, Don, 108
 Syntactic structure, 31
 Syntax, 5
 abstract, 5
 concrete, 5

T

Tail call, 211, 212

- and continuations, 221–222
- definition, 210
- in C programming language, 244
- optimization, 244–248
- Tail position, 212
- Tail-recursive, 211
- Target language, 14
- TCALL (x86 instruction), 144, 222
- Tennent, Robert, 9
- Terminal symbol, 34
- Thompson, Ken, 124
- THROW instruction, 278
- Tofte, Mads, vii, 75, 248
- Tokenizer, 39
- Top-down parser, 33
- Torvalds, Linus, 284
- To-space, 189
- try (F# keyword), 319
- Turing, Alan, 89, 178
- Turner, David A., 75
- Two-space garbage collection, 189
- Type
 - checking, 67–70
 - inference, 98–101
 - polymorphic, 97–114
 - scheme, 99
 - variable, 103
- type (F# function), 317, 320, 325
- Type scheme, 99
 - principal, 103

U

- Ullman, Jeffrey D., 54, 156
- Union operation (union-find), 107
- Union-find data structure, 106–107
- unit (F# keyword), 316
- Unmanaged CIL, 170
- Unreachable code, 243

V

- Variable capture, 19
- Variable occurrence

- bound, 15
- free, 15
- Variance
 - in C#, 112
 - in Java, 112
- VB.NET programming language, 162
- Verification of bytecode, 169
- Virtual machine, 161
- VM, *see* Virtual machine

W

- Wadler, Phil, 75
- Wadsworth, Christopher, 74, 227
- Wand, Mitchell, 228
- White block, 199
- Wildcard in generic type (Java), 111
- Wilson, Paul, 202
- Wirth, Niklaus, 9, 156
- with (F# keyword), 314, 320
- Word (x86 terminology), 285
- Write barrier, 191
- WriteLine (F# keyword), 316

X

- X86-64 instruction set, 284
- xor (x86 instruction), 287
- X86 processor family, 284–291
 - assembly code, 287–290
 - evolution, 284
 - instructions, 285
 - registers, 285
 - stack layout, 288
 - status flags, 286

Y

- Yellin, Frank, 177
- Young generation, 209

Z

- Zero flag ZF (x86), 286