Programming Language Concepts for Software Developers

Peter Sestoft
IT University of Copenhagen, Denmark

Draft version 0.36 of 2009-08-23

Preface

This book takes an operational approach to presenting programming language concepts, studying those concepts in interpreters and compilers for a range of toy languages, and pointing out where those concepts are found in real-world programming languages.

What is covered Topics covered include abstract and concrete syntax; functional and imperative; interpretation, type checking, and compilation; continuations and peep-hole optimizations; abstract machines, automatic memory management and garbage collection; the Java Virtual Machine and Microsoft's (.NET) Common Language Runtime; and reflection and runtime code generation using these execution platforms.

Some effort is made throughout to put programming language concepts into their historical context, and to show how the concepts surface in languages that the students are assumed to know already; primarily Java or C#.

We do not cover regular expressions and parser construction in much detail. For this purpose, we have used compiler design lecture notes written by Torben Mogensen [**?**], University of Copenhagen.

Why virtual machines? We do not consider generation of machine code for 'real' microprocessors, nor classical compiler subjects such as register allocation. Instead the emphasis is on virtual stack machines and their intermediate languages, often known as bytecode.

Virtual machines are machine-like enough to make the central purpose and concepts of compilation and code generation clear, yet they are much simpler than present-day microprocessors such as Intel Pentium. Full understanding of performance issues in 'real' microprocessors, with deep pipelines, register renaming, out-of-order execution, branch prediction, translation lookaside buffers and so on, requires a very detailed study of their architecture, usually not conveyed by compiler text books anyway. Certainly, an understanding of the instruction set, such as x86, does not convey any information about

whether code is fast and or not.

The widely used object-oriented languages Java and C# are rather far removed from the 'real' hardware, and are most conveniently explained in terms of their virtual machines: the Java Virtual Machine and Microsoft's Common Language Runtime. Understanding the workings and implementation of these virtual machines sheds light on efficiency issues and design decisions in Java and C#. To understand memory organization of classic imperative languages, we also study a small subset of C with arrays, pointer arithmetics, and recursive functions.

Why F#? We use the functional language F# as presentation language throughout to illustrate programming language concepts by implementing interpreters and compilers for toy languages. The idea behind this is two-fold.

First, F# belongs to the ML family of languages and is ideal for implementing interpreters and compilers because it has datatypes and pattern matching and is strongly typed. This leads to a brevity and clarity of examples that cannot be matched by non-functional languages.

Secondly, the active use of a functional language is an attempt to add a new dimension to students' world view, to broaden their imagination. The prevalent single-inheritance class-based object-oriented programming languages (namely, Java and C#) are very useful and versatile languages. But they have come to dominate computer science education to a degree where students may become unable to imagine other programming tools, especially such that use a completely different paradigm. Our thesis is that knowledge of a functional language will make the student a better designer and programmer, whether in Java, C# or C, and will prepare him or her to adapt to the programming languages of the future.

For instance, so-called generic types and methods appeared in Java and C# in 2004 but has been part of other languages, most notably ML, since 1978. Similarly, garbage collection has been used in functional languages since Lisp in 1960, but entered mainstream use more than 30 years later, with Java.

Appendix A gives a brief introduction to those parts of F# we use in the rest of the book. The intention is that students learn enough of F# in the first third of this course, using a textbook such as Syme et al. [**?**].

Supporting material The book is accompanied by complete implementations in F# of lexer and parser specifications, abstract syntaxes, interpreters, compilers, and runtime systems (abstract machines, in Java and C) for a range of toy languages. Also, there are lecture slides in PDF, and practical exercises for every lecture (to be included in the book). This material is available separately from the author or from the course home page [**?**].

Acknowledgements This book originated as lecture notes for a sequence of courses held at the IT University of Copenhagen, Denmark. This version is updated and revised to use F# instead of Standard ML as meta-language. I would like to thank Andrzej Wasowski, Ken Friis Larsen, and past and present students, in particular Niels Kokholm and Mikkel Bundgaard, who pointed out mistakes and made suggestions on examples and presentation in earlier drafts. I also owe a big thanks to Neil D. Jones and Mads Tofte who influenced my own view of programming languages and the presentation of programming language concepts.

Warning This version of the lecture notes probably have a fair number of inconsistencies and errors. You are more than welcome to report them to me at sestoft@itu.dk — Thanks!

Contents

Chapter 1

Plan for the book

[This chapter requires more work; including variable lifetime vs scope]. This chapter introduces the approach taken and the plan followed in this book.

1.1 Meta language and object language

In linguistics and mathematics, an *object language* is a language we study (such as C++ or Latin) and the *meta language* is the language in which we conduct our discussions (such as Danish or English). Throughout this book we shall use the F# language as the meta language. We could use Java of C#, but that would be more cumbersome because of the lack of datatypes and pattern matching.

F# is a strict, strongly typed functional programming language in the ML family. Appendix A presents the basic concepts of F#: value, variable, binding, type, tuple, function, recursion, list, pattern matching, and datatype. Several books give a more detailed introduction, including Syme et al. [**?**].

It is convenient to run F# interactive sessions inside Microsoft Visual Studio (under MS Windows), or executing `fsi` interactive sessions using Mono (under Linux and MacOS X); see Appendix A.

1.2 A simple language of expressions

As an example object language we start by studying a simple language of expressions, with constants, variables (of integer type), let-bindings, (nested) scope, and operators; see file `sem1.fs`.

Thus in our example language, an abstract syntax tree (AST) represents an expression.

1.2.1 Expressions without variables

First, let us consider expressions consisting only of integer constants and two-argument (dyadic) operators such as (+) and (*). We model an expression as a term of an F# datatype `expr`, where integer constants are modelled by constructor `CstI`, and operator applications are modelled by constructor `Prim`:

```
type expr =
  | CstI of int
  | Prim of string * expr * expr
```

Here are some example expressions in this representation:

Expression	Representation in type `expr`
17	`CstI 17`
$3-4$	`Prim("-", CstI 3, CstI 4)`
$7\cdot 9+10$	`Prim("+", Prim("*", CstI 7, CstI 9), CstI 10)`

An expression in this representation can be evaluated to an integer by a function `eval : expr -> int` that uses pattern matching to distinguish the various forms of expression. Note that to evaluate e_1+e_2, it must evaluate e_1 and e_2 and to obtain two integers, and then add those, so the evaluation function must call itself recursively:

```
let rec eval (e : expr) : int =
    match e with
      | CstI i -> i
      | Prim("+", e1, e2) -> eval e1 + eval e2
      | Prim("*", e1, e2) -> eval e1 * eval e2
      | Prim("-", e1, e2) -> eval e1 - eval e2
      | Prim _            -> failwith "unknown primitive";;
```

The *eval* function is an *interpreter* for 'programs' in the expression language. It looks rather boring, as it maps the expression language constructs directly into F# constructs. However, we might change it to interpret the operator (-) as cut-off subtraction, whose result is never negative, then we get a 'language' with the same expressions but a very different meaning. For instance, $3-4$ now evaluates to zero:

```
let rec eval (e : expr) : int =
    match e with
```

```
| CstI i -> i
| Prim("+", e1, e2) -> eval e1 + eval e2
| Prim("*", e1, e2) -> eval e1 * eval e2
| Prim("-", e1, e2) ->
  let res = eval e1 - eval e2
  in if res < 0 then 0 else res
| Prim _            -> failwith "unknown primitive";;
```

1.2.2 Expressions with variables

Now, let us extend our expression language with variables. First, we add a new constructor `Var` to the syntax:

```
type expr =
  | CstI of int
  | Var of string
  | Prim of string * expr * expr
```

Here are some expressions and their representation in this syntax:

Expression	Representation in type `expr`
17	`CstI 17`
x	`, Var "x"`
$3+a$	`Prim("+", CstI 3, Var "a")`
$b \cdot 9+a$	`Prim("+", Prim("*", Var "b", CstI 9), Var "a")`

Next we need to extend the `eval` interpreter to give a meaning to such variables. To do this, we give `eval` an extra argument `env`, a so-called *environment*. The role of the environment is to associate a value (here, an integer) with a variable; that is, the environment is a map or dictionary, mapping a variable name to the variable's current value. A simple classical representation of such a map is an *association list*: a list of pairs of a variable name and the associated value:

```
let env = [("a", 3); ("c", 78); ("baf", 666); ("b", 111)];;
```

This environment maps `"a"` to 3, `"c"` to 78, and so on. The environment has type `(string * int) list`. An empty environment, which does not map any variable to anything, is represented by the empty association list

```
let emptyenv = [];;
```

To look up a variable in an environment, we define a function `lookup` of type `(string * int) list -> string -> int`. An attempt to look up variable `x` in

en empty environment fails; otherwise, if the environment first associates `y` with `v` and `x` equals `y`, then result is `v`; else the result is obtained by looking for `x` in the rest `r` of the environment:

```
let rec lookup env x =
    match env with
     | []        -> failwith (x ^ " not found")
     | (y, v)::r -> if x=y then v else lookup r x;;
```

As promised, our new `eval` function takes both an expression and an environment, and uses the environment and the `lookup` function to determine the value of a variable `Var x`. Otherwise the function is as before, except that `env` must be passed on in recursive calls:

```
let rec eval e (env : (string * int) list) : int =
    match e with
      | CstI i            -> i
      | Var x             -> lookup env x
      | Prim("+", e1, e2) -> eval e1 env + eval e2 env
      | Prim("*", e1, e2) -> eval e1 env * eval e2 env
      | Prim("-", e1, e2) -> eval e1 env - eval e2 env
      | Prim _            -> failwith "unknown primitive";;
```

Note that our `lookup` function returns the *first* value associated with a variable, so if `env` is `[("x", 11); ("x", 22)]`, then `lookup env "x"` is 11, not 22. This is useful when we consider nested scopes in Chapter 2.

1.3 Syntax and semantics

We have already mentioned syntax and semantics. *Syntax* deals with form: is this text a well-formed program? *Semantics* deals with meaning: what does this (well-formed) program mean, how does it behave – what happens when we execute it?

- Syntax – form: is this a well-formed program?
  - Abstract syntax – programs as trees, or values of an F# datatype such as `Prim("+", CstI 3, Var "a")`
  - Concrete syntax – programs as linear texts such as '$3+a$'.
- Semantics – meaning: what does this well-formed program mean?
  - Static semantics – is this well-formed program a legal one?

– Dynamic semantics – what does this program do when executed?

The distinction between syntax and static semantics is not clear-cut. Syntax can tell us that *x12* is a legal variable name (in Java), but it is impractical to use syntax to tells us that we cannot declare *x12* twice in the same scope (in Java). Hence this restriction is usually enforced by static semantics checks.

In the rest of the book we shall study a small example language, two small functional languages (a first-order and a higher-order one), a subset of the imperative language C, and a subset of the backtracking (or goal-directed) language Icon. In each case we take the following approach:

- We describe abstract syntax using F# datatypes.
- We describe concrete syntax using lexer and parser specifications (see Chapter 3), and implement lexers and parsers using `fslex` and `fsyacc`.
- We describe semantics using F# functions, both static semantics (checks) and dynamic semantics (execution). The dynamic semantics can be described in two ways: by direct interpretation using functions typically called `eval`, or by compilation to another language, such as stack machine code, using functions typically called `comp`.

In addition we study some abstract stack machines, both homegrown ones and two widely used so-called managed execution platforms: The Java Virtual Machine (JVM) and Microsoft's .Net Common Language Runtime.

1.4 Object-oriented representation of expressions

In this book we use a functional language to represent expressions and other program fragments. In particular, we use the F# algebraic datatype `expr` to represent expressions in the form of *abstract syntax*. We use the `eval` function to define their *dynamic semantics*, using pattern matching to distinguish the different forms of expressions: constants, variables, operators applications.

In this section we briefly consider an object-oriented modelling (in Java, say) of expression syntax and expression evaluation. In general, this would involve an abstract base class `Expr` of expressions (instead of the `expr` datatype), and a concrete subclass for each form of expression (instead of datatype constructor for each form of expression):

```
abstract class Expr { }
class CstI extends Expr {
  protected final int i;
```

```
  public CstI(int i) { this.i = i; }
}
class Var extends Expr {
  protected final String name;
  public Var(String name) { this.name = name; }
}
class Prim extends Expr {
  protected final String oper;
  protected final Expr e1, e2;
  public Prim(String oper, Expr e1, Expr e2) {
    this.oper = oper; this.e1 = e1; this.e2 = e2;
  }
}
```

Note that each `Expr` subclass has fields of exactly the same types as the arguments of the corresponding constructor in the `expr` datatype from Section 1.2.2. Also, in object-oriented terms `Prim` is a composite because it has fields whose type is its base type `Expr`; in functional programming terms one would say that the type declaration is recursive.

How can we define a function (method) that processes object structures that represent expressions; for instance, converts an expression to its String representation? We declare an abstract method on class `Expr`, override it in each subclass, and then rely on virtual method calls to invoke the correct override in the composite case:

```
abstract class Expr {
  abstract public String fmt();
}
class CstI extends Expr {
  protected final int i;
  ...
  public String fmt() { return i + ""; }
}
class Var extends Expr {
  protected final String name;
  ...
  public String fmt() { return name; }
}
class Prim extends Expr {
  protected final String oper;
  protected final Expr e1, e2;
  ...
  public String fmt() {
    return "(" + e1.fmt() + oper + e2.fmt() + ")";
  }
```

```
}
```

Most of the developments in this book could have been carried out in an object-oriented language, but the extra verbosity (of Java or C#) and the lack of nested pattern matching, would make the presentation considerable more involved in many cases.

1.5 The history of programming languages

Since 1956, thousands of programming languages have been proposed and implemented, but only a modest number of them, maybe a few hundred, have been widely used. Most new programming languages arise as a reaction to some language that the designer knows (and likes or dislikes) already, so one can propose a family tree or genealogy for programming languages, just as for living organisms. Figure 1.1 presents one such attempt.

In general, languages lower in the diagram (near the time axis) are closer to the real hardware than those higher in the diagram, which are more 'high-level' in some sense. In Fortran77 or C, it is fairly easy to predict what instructions and how many instructions will be executed at run-time for a given line of program. The mental machine model that the C or Fortran77 programmer must use to write efficient programs is very close to the real machine.

Conversely, the top-most languages (SASL, Haskell, Standard ML, F#) are functional languages, possibly with lazy evaluation, with dynamic or advanced static type systems and with automatic memory management, and it is in general difficult to predict how many machine instructions are required to evaluate any given expression. The mental machine model that the Haskell or Standard ML or F# programmer must use to write efficient programs is far from the details of a real machine, so he can think on a rather higher level. On the other hand, he loses control over detailed efficiency.

It is remarkable that the recent mainstream languages Java and C#, especially their post-2004 incarnations, have much more in common with the academic languages of the 1980's than with those languages that were used in the 'real world' during those years (C, Pascal, C++).

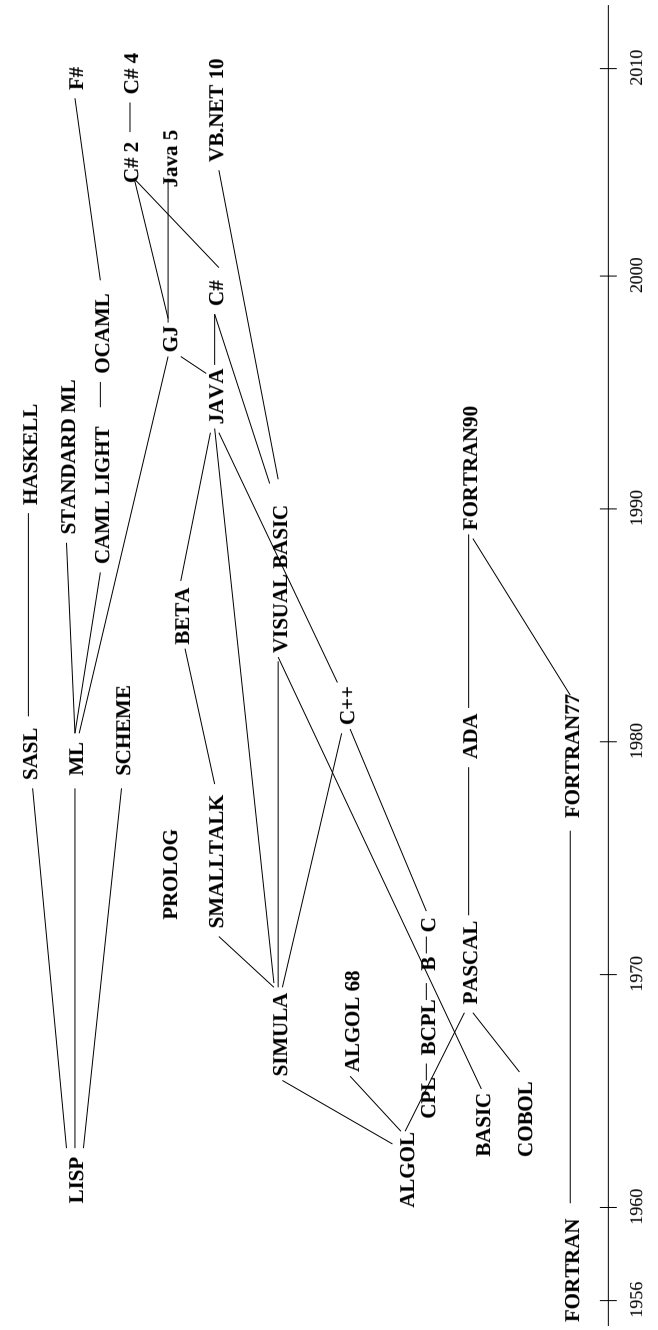

Figure 1.1: The genealogy of programming languages

Chapter 2

Interpreters and compilers

[This chapter is rather sketchy too.] This chapter introduces the distinction between interpreters and compilers, and demonstrates some concepts of compilation, using the simple expression language as an example. Some concepts of interpretation are illustrated also, using a stack machine as an example.

2.1 What files are provided for this chapter

File	Contents
`sem2.fs`	expression abstract syntax, evaluators, compilers
`prog.ps`	a simple Postscript program (see Section 2.6)
`sierpinski.eps`	an intricate Postscript program (see Section 2.6)
`Expr/Machine.java`	abstract machine in Java (see Section 2.8)

2.2 Interpreters and compilers

An *interpreter* executes a program on some input, producing an output or result; see Figure 2.1. An interpreter is usually itself a program, but one might also say that an Intel x86 processor (using in PC's) or an IBM PowerPC processor (used in Apple's computers) is an interpreter, implemented in silicon. For an interpreter program we must distinguish the interpreted language L (the language of the programs being executed, for instance our expression language expr) from the implementation language I (the language in which the interpreter is written, for instance F#). When program in the interpreted language L is a sequence of simple instructions, and thus looks like machine code, the interpreter is often called an abstract machine or virtual machine.

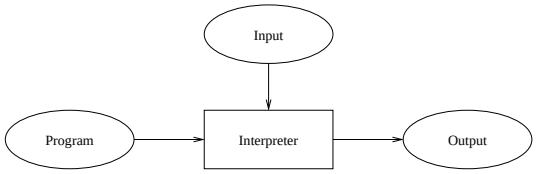

Figure 2.1: Interpretation in one stage

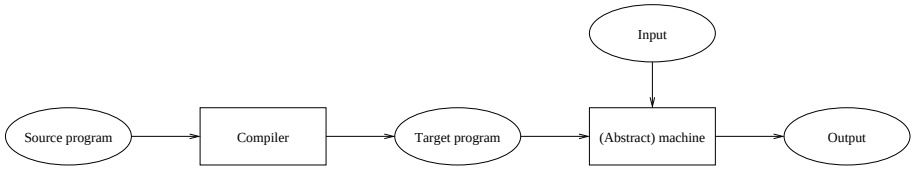

Figure 2.2: Compilation and execution in two stages

A *compiler* takes as input a source program and generates as output another (equivalent) program, called a target program, which can then be executed; see Figure 2.2. We must distinguish three languages: the source language S (eg. `expr`) of the input programs, the target language T (eg. `texpr`) of the output programs, and the implementation language I (for instance, F#) of the compiler itself.

The compiler does not execute the program; after the target program has been generated it must be executed by a machine or interpreter which can execute programs written in language T. Hence we can distinguish between compile-time (at which time the source program is compiled into a target program) and run-time (at which time the target program is executed on actual inputs to produce a result). At compile-time one usually also performs various so-called well-formedness checks of the source program: are all variables bound? do operands have the correct type in expressions? etc.

2.3 Variables: scope, and bound and free occurrences

The *scope* of a variable binding is that part of a program in which it is visible. For instance, the scope of the binding of `x` in this F# expression is the expression `x + 3`:

```
let x = 6 in x + 3
```

A language has *static scope* if the scopes of bindings follow the syntactic structure of the program. Most modern languages, such as C, C++, Pascal, Algol, Scheme, Java, C# and F# have static scope; but see Section 4.6 for some that do not.

A language has *nested scope* if an inner scope may create a 'hole' in an outer scope by declaring a new variable with the same name, as shown by this F# expression, where the second binding of `x` hides the first one in `x+2` but not in `x+3`:

```
let x = 6 in (let x = x + 2 in x * 2) + (x + 3)
```

Nested scope is known also from Standard ML, C, C++, Pascal, Algol; and from Java and C#, for instance when a parameter or local variable in a method hides a field from an enclosing class, or when a declaration in a Java anonymous inner class or a C# anonymous method hides a local variable already in scope.

It is useful to distinguish bound and free occurrences of a variable. A variable occurrence is *bound* if it occurs within the scope of a binding for that variable, and *free* otherwise. That is, `x` occurs bound in the body of this let-binding:

```
let x = 6 in x + 3
```

but `x` occurs free in this one:

```
let y = 6 in x + 3
```

and in this one

```
let y = x in y + 3
```

and it occurs free (the first time) as well as bound (the second time) in this expression

```
let x = x + 6 in x + 3
```

2.3.1 Expressions with let-bindings and static scope

Now let us extend the expression language from Section 1.2 with `let`-bindings of the form `let x = e1 in e2`, here represented by the `Let` constructor:

```
type expr =
  | CstI of int
  | Var of string
  | Let of string * expr * expr
  | Prim of string * expr * expr
```

Using the same environment representation and `lookup` function as in Section 1.2.2, we can interpret `let x = erhs in ebody` as follows. We evaluate the right-hand side `erhs` in the same environment as the entire let-expression, obtaining a value `xval` for `x`; then we create a new environment `env1` by adding the association (`x`, `xval`) and interpret the let-body `ebody` in that environment; finally we return the result as the result of the let-binding:

```
let rec eval e (env : (string * int) list) : int =
    match e with
      | CstI i            -> i
      | Var x             -> lookup env x
      | Let(x, erhs, ebody) ->
        let xval = eval erhs env
        let env1 = (x, xval) :: env
        in eval ebody env1
      | Prim("+", e1, e2) -> eval e1 env + eval e2 env
      | Prim("*", e1, e2) -> eval e1 env * eval e2 env
      | Prim("-", e1, e2) -> eval e1 env - eval e2 env
      | Prim _            -> failwith "unknown primitive";;
```

The new binding of `x` will hide any existing binding of `x`, thanks to the definition of `lookup`. Also, since the old environment `env` is not destructively modified — the new environment `env1` is just a temporary extension of it — further evaluation will continue on the old environment. Hence we obtain nested static scopes.

2.3.2 Closed expressions

An expression is *closed* if no variable occurs free in the expression. In most programming languages, programs must be closed: they cannot have unbound (undeclared) names. To efficiently test whether an expression is closed, we define a slightly more general concept, `closedin e vs`, of an expression `e` being closed in a list `vs` of bound variables:

```
let rec closedin (e : expr) (vs : string list) : bool =
    match e with
      | CstI i -> true
      | Var x  -> List.exists (fun y -> x=y) vs
      | Let(x, erhs, ebody) ->
        let vs1 = x :: vs
        in closedin erhs vs && closedin ebody vs1
      | Prim(ope, e1, e2) -> closedin e1 vs && closedin e2 vs;;
```

A constant is always closed. A variable occurrence `x` is closed in `vs` if `x` appears in `vs`. The expression `let x=erhs in ebody` is closed in `vs` if `erhs` is closed in `vs`

and ebody is closed in x :: vs. An operator application is closed in vs if both its operands are.

Now, an expression is closed if it is closed in the empty environment []:

```
let closed1 e = closedin e [];;
```

2.3.3 The set of free variables

Now let us compute the set of variables that occur free in an expression. First, if we represent a set of variables as a list without duplicates, then [] represents the empty set, and [x] represents the singleton set containing just x, and one can compute set union and set difference like this:

```
let rec union (xs, ys) =
    match xs with
      | []    -> ys
      | x::xr -> if mem x ys then union(xr, ys)
                 else x :: union(xr, ys);;
let rec minus (xs, ys) =
    match xs with
      | []    -> []
      | x::xr -> if mem x ys then minus(xr, ys)
                 else x :: minus (xr, ys);;
```

Now the set of free variables can be computed easily:

```
let rec freevars e : string list =
    match e with
      | CstI i -> []
      | Var x  -> [x]
      | Let(x, erhs, ebody) ->
            union (freevars erhs, minus (freevars ebody, [x]))
      | Prim(ope, e1, e2) -> union (freevars e1, freevars e2);;
```

The set of free variables in a constant is the empty set []. The set of free variables in a variable occurrence x is the singleton set [x]. The set of free variables in let x=erhs in ebody is the union of the free variables in erhs, with the free variables of ebody minus x. The set of free variables in an operator application is the union of the sets of free variables in its operands.

This gives a direct way to compute whether an expression is closed; simply check that the set of its free variables is empty:

```
let closed2 e = (freevars e = []);;
```

2.4 Integer addresses instead of names

For efficiency, symbolic variable names are replaced by variable addresses (integers) in real machine code and in most interpreters. To show how this may be done, we define an abstract syntax `texpr` for target expressions that uses (integer) variable indexes instead of symbolic variable names:

```
type texpr =                              (* target expressions *)
  | TCstI of int
  | TVar of int                           (* index into runtime environment *)
  | TLet of texpr * texpr                 (* erhs and ebody                 *)
  | TPrim of string * texpr * texpr
```

Then we can define a function

```
tcomp : expr -> string list -> texpr
```

to compile an `expr` to a `texpr` within a given compile-time environment. The compile-time environment maps the symbolic names to integer variable indexes. In the interpreter `teval` for `texpr`, a run-time environment maps integers (variable indexes) to variable values (accidentally also integers in this case).

In fact, the compile-time environment in `tcomp` is just a `string list`, a list of the bound variables. The position of a variable in the list is its binding depth (the number of other let-bindings between the variable occurrence and the binding of the variable). Correspondingly, the run-time environment in `teval` is an `int list` storing the values of the variables in the same order as their names in compile-time environment. Therefore we can simply use the binding depth of a variable to access the variable at run-time. The integer giving the position is called an offset by compiler writers, and a deBruijn index by theoreticians (in the lambda calculus): the number of binders between this occurrence of a variable, and its binding.

The type of `teval` is

```
teval : texpr -> int list -> int
```

Note that in one-stage interpretive execution (`eval`) the environment had type `(string * int) list` and contained both variable names and variable values. In the two-stage compiled execution, the compile-time environment (in `tcomp`) had type `string list` and contained variable names only, whereas the run-time environment (in `teval`) had type `int list` and contained variable values only.

Thus effectively the joint environment from interpretive execution has been split into a compile-time environment and a run-time environment. This is no

accident: the purpose of compiled execution is to perform some computations (such as variable lookup) early, at compile-time, and perform other computations (such as multiplications of variables' values) only later, at run-time.

The correctness requirement on a compiler can be stated using equivalences such as this one:

```
eval e [] equals teval (tcomp e []) []
```

which says that

- if `te = tcomp e []` is the result of compiling the closed expression `e` in the empty compile-time environment `[]`,
- then evaluation of the target expression `te` using the `teval` interpreter and empty run-time environment `[]` should produce the same result as evaluation of the source expression `e` using the `eval` interpreter and an empty environment `[]`,
- and vice versa.

2.5 Stack machines for expression evaluation

Expressions, and more generally, functional programs, are often evaluated by a *stack machine*. We shall study a simple stack machine (an interpreter which implements an abstract machine) for evaluation of expressions in *postfix* (or *reverse Polish*) form. Reverse Polish form is named after the Polish philosopher and mathematician Jan Łukasiewicz (1878–1956).

Stack machine instructions for an example language without variables (and hence without let-bindings) may be described using this F# type:

```
type rinstr =
  | RCstI of int
  | RAdd
  | RSub
  | RMul
  | RDup
  | RSwap
```

The state of the stack machine is a pair (c,s) of the control and the stack. The control c is the sequence of instructions yet to be evaluated. The stack s is a list of values (here integers), namely, intermediate results.

The stack machine can be understood as a transition system, described by the rules shown in Figure 2.3. Each rule says how the execution of one

Instruction	Stack before		Stack after	Effect
`RCst` i	s	$\Rightarrow$	s, i	Push constant
`RAdd`	s, i_1, i_2	$\Rightarrow$	$s, (i_1 + i_2)$	Addition
`RSub`	s, i_1, i_2	$\Rightarrow$	$s, (i_1 - i_2)$	Subtraction
`RMul`	s, i_1, i_2	$\Rightarrow$	$s, (i_1 * i_2)$	Multiplication
`RDup`	s, i	$\Rightarrow$	s, i, i	Duplicate stack top
`RSwap`	s, i_1, i_2	$\Rightarrow$	s, i_2, i_1	Swap top elements

Figure 2.3: Stack machine instructions for expression evaluation

instruction causes the machine may go from one state to another. The stack top is to the right.

For instance, the second rule says that if the two top-most stack elements are 5 and 7, so the stack has form $s, 7, 5$ for some s, then executing the `RAdd` instruction will cause the stack to change to $s, 12$.

The rules of the abstract machine are quite easily translated into an F# function (see file `sem2.fs`):

```
reval : rinstr list -> int list -> int
```

The machine terminates when there are no more instructions to execute (or we might invent an explicit `RStop` instruction, whose execution would cause the machine to ignore all subsequent instructions). The result of a computation is the value on top of the stack when the machine stops.

The *net effect principle* for stack-based evaluation says: regardless what is on the stack already, the net effect of the execution of an instruction sequence generated from an expression `e` is to push the value of `e` onto the evaluation stack, leaving the given contents of the stack unchanged.

Expressions in postfix or reverse Polish notation are used by scientific pocket calculators made by Hewlett-Packard, primarily popular with engineers and scientists. A significant advantage of postfix notation is that one can avoid the parentheses found on other calculators. The disadvantage is that the user must 'compile' expressions from their usual algebraic notation to stack machine notation, but that is surprisingly easy to learn.

2.6 Postscript, a stack-based language

Stack-based (interpreted) languages are widely used. The most notable among them is Postscript (ca 1984), which is implemented in almost all high-end laser-

printers. By contrast, Portable Document Format (PDF), also from Adobe Systems, is not a full-fledged programming language.

Forth (ca. 1968) is another stack-based language, which is an ancestor of Postscript. It is used in embedded systems to control scientific equipment, satellites etc.

In Postscript one can write

```
4 5 add 8 mul =
```

to compute $(4+5)*8$ and print the result, and

```
/x 7 def
x x mul 9 add =
```

to bind `x` to 7 and then compute `x*x+9` and print the result. The '=' function in Postscript pops a value from the stack and prints it. A name, such as `x`, that appears by itself causes its value to be pushed onto the stack. When defining the name (as opposed to using its value), it must be escaped with a slash as in `/x`.

The following defines the factorial function under the name `fac`:

```
/fac { dup 0 eq { pop 1 } { dup 1 sub fac mul } ifelse } def
```

This is equivalent to the F# function declaration

```
let rec fac n = if n=0 then 1 else n * fac (n-1)
```

Note that the `ifelse` conditional expression is postfix also, and expects to find three values on the stack: a boolean, a then-branch, and an else-branch. The then- and else-branches are written as code fragments, which in Postscript are enclosed in curly braces.

Similarly, a `for`-loop expects four values on the stack: a start value, a step value, and an end value for the loop index, and a loop body. It repeatedly pushes the loop index and executes the loop body. Thus one can compute and print factorial of $0, 1, \ldots, 12$ this way:

```
0 1 12 { fac = } for
```

One can use the `gs` (Ghostscript) interpreter to experiment with Postscript programs. Under Linux, use

```
gs -dNODISPLAY
```

and under Windows, use something like

```
gswin32 -dNODISPLAY
```

For more convenient interaction, run Ghostscript inside an Emacs shell (under Linux or MS Windows).

If `prog.ps` is a file containing Postscript definitions, `gs` will execute them on start-up if invoked with

```
gs -dNODISPLAY prog.ps
```

A function definition entered interactively in Ghostscript must fit on one line, but a function definition included from a file need not.

The example Postscript program below (file `prog.ps`)) prints some text in Times Roman and draws a rectangle. If you send this program to a Postscript printer, it will be executed by the printer's Postscript interpreter, and a sheet of printed paper will be produced:

```
/Times-Roman findfont 25 scalefont setfont
100 500 moveto
(Hello, Postscript!!) show
newpath
100 100 moveto
300 100 lineto 300 250 lineto
100 250 lineto 100 100 lineto stroke
showpage
```

Another short but much fancier Postscript example is found in file `sierpinski.eps`. It defines a recursive function that draws a *Sierpinski curve*, a recursively defined figure in which every part is similar to the whole. The core of the program is function `sierp`, which either draws a triangle (first branch of the `ifelse`) or calls itself recursively three times (second branch). The percent sign (%) starts and end-of-line comment in Postscript:

```
%!PS-Adobe-2.0 EPSF-2.0
%%Title: Sierpinski
%%Author: Morten Larsen (ml@dina.kvl.dk) LIFE, University of Copenhagen
%%CreationDate: Fri Sep 24 1999
%%BoundingBox: 0 0 444 386
% Draw a Sierpinski triangle

/sierp { % stack xtop ytop w h
dup 1 lt 2 index 1 lt or {
  % Triangle less than 1 point big - draw it
     4 2 roll moveto
     1 index -.5 mul exch -1 mul rlineto 0 rlineto closepath stroke
  } {
```

```
  % recurse
     .5 mul exch .5 mul exch
     4 copy sierp
     4 2 roll 2 index sub exch 3 index .5 mul 5 copy sub exch 4 2 roll sierp
     add exch 4 2 roll sierp
  } ifelse
} bind def

0 setgray
.1 setlinewidth
222 432 60 sin mul 6 add 432 1 index sierp
showpage
```

A complete web-server has been written in Postscript, see
`http://www.pugo.org/main/project_pshttpd/`

The Postscript Language Reference [**?**] can be downloaded from Adobe Corporation.

2.7 Compiling expressions to stack machine code

The datatype `sinstr` is the type of instructions for a stack machine with variables, where the variables are stored on the evaluation stack:

```
type sinstr =
  | SCstI of int                    (* push integer           *)
  | SVar of int                     (* push variable from env *)
  | SAdd                            (* pop args, push sum     *)
  | SSub                            (* pop args, push diff.   *)
  | SMul                            (* pop args, push product *)
  | SPop                            (* pop value/unbind var   *)
  | SSwap                           (* exchange top and next  *)
```

Since both `stk` in `reval` and `env` in `teval` behave as stacks, and because of lexical scoping, they could be replaced by a single stack, holding both variable bindings and intermediate results. The important property is that the binding of a let-bound variable can be removed once the entire let-expression has been evaluated.

Thus we define a stack machine `seval` that uses a unified stack both for storing intermediate results and bound variables. We write a new version `scomp` of `tcomp` to compile every use of a variable into an (integer) offset from the stack top. The offset depends not only on the variable declarations, but also the number of intermediate results currently on the stack. Hence the same variable may be referred to by different indexes at different occurrences. In the expression

```
Let("z", CstI 17, Prim("+", Var "z", Var "z"))
```

the two uses of `z` in the addition get compiled to two different offsets, like this:

```
[SCstI 17, SVar 0, SVar 1, SAdd, SSwap, SPop]
```

The expression `20 + let z = 17 in z + 2 end + 30` is compiled to

```
[SCstI 20, SCstI 17, SVar 0, SCst 2, SAdd, SSwap, SPop, SAdd,
 SCstI 30, SAdd]
```

Note that the let-binding `z = 17` is on the stack above the intermediate result 20, but once the evaluation of the let-expression is over, only the intermediate results 20 and 19 are on the stack, and can be added.

The correctness of the `scomp` compiler and the stack machine `seval` relative to the expression interpreter `eval` can be asserted as follows. For an expression `e` with no free variables,

`seval (scomp e []) []` equals `eval e [] eval e []`

More general functional languages may be compiled to stack machine code with stack offsets for variables. For instance, Moscow ML is implemented that way, with a single stack for temporary results, function parameter bindings, and let-bindings.

2.8 Implementing an abstract machine in Java

An abstract machine implemented in F# may not seem very machine-like. One can get a step closer to real hardware by implementing the abstract machine in Java. One technical problem is that the `sinstr` instructions must be represented as numbers, so that the Java program can read the instructions from a file. We can adopt a representation such as this one:

Instruction	Bytecode
`SCst i`	`0 i`
`SVar x`	`1 x`
`SAdd`	`2`
`SSub`	`3`
`SMul`	`4`
`SPop`	`5`
`SSwap`	`6`

Note that most `sinstr` instructions are represented by a single number ('byte') but that those that take an argumen (`SCst i` and `SVar x`) are represented by two numbers: the instruction code and the argument. For example, the `[SCstI 17, SVar 0, SVar 1, SAdd, SSwap, SPop]` instruction sequence will be represented by the number sequence `0 17 1 0 1 1 2 6 5`.

This form of numeric program code can be executed by the method `seval` shown in Figure 2.4.

```
class Machine {
  final static int
    CST = 0, VAR = 1, ADD = 2, SUB = 3, MUL = 4, POP = 5, SWAP = 6;
  static int seval(int[] code) {
    int[] stack = new int[1000];        // evaluation and env stack
    int sp = -1;                        // pointer to current stack top
    int pc = 0;                         // program counter
    int instr;                          // current instruction
    while (pc < code.length)
      switch (instr = code[pc++]) {
      case CST:
        stack[sp+1] = code[pc++]; sp++; break;
      case VAR:
        stack[sp+1] = stack[sp-code[pc++]]; sp++; break;
      case ADD:
        stack[sp-1] = stack[sp-1] + stack[sp]; sp--; break;
      case SUB:
        stack[sp-1] = stack[sp-1] - stack[sp]; sp--; break;
      case MUL:
        stack[sp-1] = stack[sp-1] * stack[sp]; sp--; break;
      case POP:
        sp--; break;
      case SWAP:
        { int tmp     = stack[sp];
          stack[sp]   = stack[sp-1];
          stack[sp-1] = tmp;
          break;
        }
      default: ... error: unknown instruction ...
    return stack[sp];
  }
}
```

Figure 2.4: Stack machine in Java for expression evaluation