Chapter 2
Interpreters and Compilers

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 Files for This Chapter

File	Contents
`Intcomp/Intcomp1.fs`	very simple expression interpreter and compilers
`Intcomp/Machine.java`	abstract machine in Java (see Sect. 2.8)
`Intcomp/prog.ps`	a simple Postscript program (see Sect. 2.6)
`Intcomp/sierpinski.eps`	an intricate Postscript program (see Sect. 2.6)

2.2 Interpreters and Compilers

An *interpreter* executes a program on some input, producing an output or result; see Fig. 2.1. An interpreter is usually itself a program, but one might also say that an Intel or AMD x86 processor (used in many portable, desktop and server computers) or an ARM processor (used in many mobile phones and tablet 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 the 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.

P. Sestoft, *Programming Language Concepts*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-1-4471-4156-3_2, © Springer-Verlag London 2012

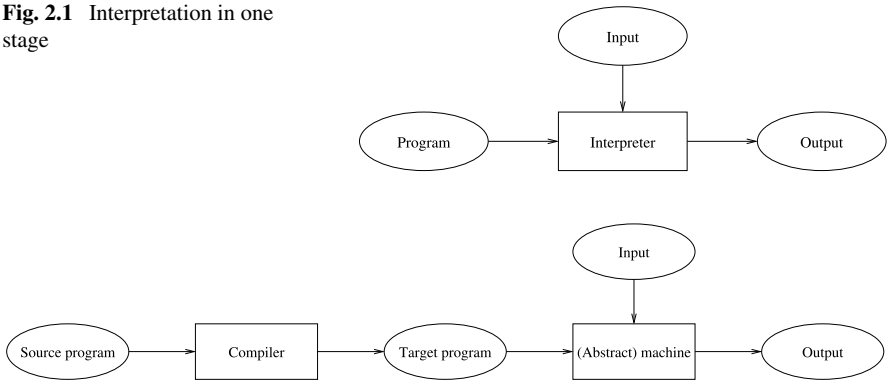

Fig. 2.1 Interpretation in one stage

Fig. 2.2 Compilation and execution in two stages

A *compiler* takes as input a source program and generates as output another program, called a target program, which can then be executed; see Fig. 2.2. We must distinguish three languages: the source language S (e.g. `expr`) of the input programs, the target language T (e.g. `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, and so on.

2.3 Scope and Bound and Free Variables

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

```
let f x = 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 Sect. 4.7 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# function definition, where the second binding of `x` hides the first one in `x*2` but not in `x+3`, so `f 1` evaluates to $(8 \cdot 2) + (1 + 3) = 20$:

```
let f x = (let x = 8 in x * 2) + (x + 3)
```

Nested scope is known also from Algol, Pascal, C, C++, and Standard ML; 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 Sect. 1.3 with let-bindings of the form `let x = e1 in e2 end`, 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 Sect. 1.3.2, we can evaluate `let x = erhs in ebody end` 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
      eval ebody env1
    | ...
```

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 in 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, a program must be closed: it 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
      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 end 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, we represent a set of variables as a list without duplicates, so [] 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 end` 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 another way to compute whether an expression is closed; simply check that the set of its free variables is empty:

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

2.3.4 Substitution: Replacing Variables by Expressions

In the preceding sections we have seen how to compute the free variables of an expression. In this section we show how to perform *substitution*, replacing free variables by expressions. The basic idea is to have an environment `env` that maps some variable names to expressions. For instance, this environment says that variable `z` should be replaced by the expression `5-4` and any other variable should be left as is:

```
[("z", Prim("-", CstI 5, CstI 4))]
```

Applying this substitution to `y*z` should produce `y*(5-4)` in which `y` is left as is. The above substitution environment may be written $[(5-4)/z]$, and the application of that substitution to `y*z` is written $[(5-4)/z](y*z)$.

To implement a substitution function in F#, we need a version of `lookup` that maps each variable present in the environment to the associated expression, and maps absent variables to themselves. We call this function `lookOrSelf`:

```
let rec lookOrSelf env x =
    match env with
    | []        -> Var x
    | (y, e)::r -> if x=y then e else lookOrSelf r x
```

Moreover, a substitution should only replace *free* variables. Applying the substitution $[(5-4)/z]$ to `let z=22 in y*z end` should *not* produce `let z=22`

`in y*(5-4) end` because the occurrence of `z` is bound, and should always evaluate to 22, not to −1. When we "go below" a let-binding of a variable `z`, then `z` should be removed from the substitution environment; hence this auxiliary function:

```
let rec remove env x =
    match env with
    | []        -> []
    | (y, e)::r -> if x=y then r else (y, e) :: remove r x
```

Finally we can present a substitution function (which, alas, turns out to be somewhat naive):

```
let rec nsubst (e : expr) (env : (string * expr) list) : expr =
    match e with
    | CstI i -> e
    | Var x  -> lookOrSelf env x
    | Let(x, erhs, ebody) ->
      let newenv = remove env x
      Let(x, nsubst erhs env, nsubst ebody newenv)
    | Prim(ope, e1, e2) ->
      Prim(ope, nsubst e1 env, nsubst e2 env)
```

Explanation: Substitution does not affect constants `CstI` so the given expression `e` is returned. Substitution replaces a variable `x` with whatever the substitution environment `env` says, or with itself if it is not bound in the environment. Substitution replaces a primitive operation with the same operation, after applying the substitution to both operand expressions `e1` and `e2`. Substitution replaces a let-binding `let x = erhs in ebody end` by another let-binding, after applying the substitution to `erhs`, and applying a slightly different substitution to `ebody` where the variable `x` has been removed from the substitution environment.

Apparently, this works fine, correctly replacing `z` but not `y` in `e6`, and correctly replacing none of them in `e9`:

```
> let e6 = Prim("+", Var "y", Var "z");;
> let e6s2 = nsubst e6 [("z", Prim("-", CstI 5, CstI 4))];;
val e6s2 : expr = Prim ("+",Var "y",Prim ("-",CstI 5,CstI 4))

> let e9 = Let("z",CstI 22,Prim("*",Var "y",Var "z"));;
> let e9s2 = nsubst e9 [("z", Prim("-", CstI 5, CstI 4))];;
val e9s2 : expr = Let ("z",CstI 22,Prim ("*",Var "y",Var "z"))
```

However, if we try a slightly fancier substitution $[z/y]$, where we want to replace `y` by `z`, then we discover a problem, namely, *variable capture*:

```
> let e9s1 = nsubst e9 [("y", Var "z")];;
val e9s1 = Let ("z",CstI 22,Prim ("*",Var "z",Var "z"))
```

In an attempt to replace a free variable `y` by another free variable `z`, the free `z` got "captured" under the let-binding and thereby turned into a bound variable `z`. There is a simple way around this problem, which is to systematically rename all the let-bound variables with fresh names that are used nowhere else. This can be

implemented elegantly by the substitution process itself, to obtain capture-avoiding substitution:

```
let rec subst (e : expr) (env : (string * expr) list) : expr =
    match e with
    | CstI i -> e
    | Var x  -> lookOrSelf env x
    | Let(x, erhs, ebody) ->
      let newx = newVar x
      let newenv = (x, Var newx) :: remove env x
      Let(newx, subst erhs env, subst ebody newenv)
    | Prim(ope, e1, e2) ->
      Prim(ope, subst e1 env, subst e2 env)
```

where a simple `newVar` auxiliary may be defined like this (provided no "normal" variable name ends with a number):

```
let newVar : string -> string =
    let n = ref 0
    let varMaker x = (n := 1 + !n; x + string (!n))
    varMaker
```

Now the problematic example above is handled correctly because the let-bound variable `z` gets renamed to `z4` (or similar), so the free variable `z` does not get captured:

```
> let e9s1a = subst e9 [("y", Var "z")];;
val e9s1a = Let ("z4",CstI 22,Prim ("*",Var "z",Var "z4"))
```

Note that we systematically rename *bound* variables to avoid capture of *free* variables. Capture-avoiding substitution plays an important role in program transformation and in programming language theory.

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 =
  | TCstI of int
  | TVar of int                          (* run-time index *)
  | TLet of texpr * texpr                (* erhs and ebody *)
  | TPrim of string * texpr * texpr
```

Then we can define `tcomp : expr -> string list -> texpr`, a compiler from `expr` to `texpr`, like this:

```
let rec tcomp (e : expr) (cenv : string list) : texpr =
    match e with
```

```
    | CstI i -> TCstI i
    | Var x  -> TVar (getindex cenv x)
    | Let(x, erhs, ebody) ->
      let cenv1 = x :: cenv
      TLet(tcomp erhs cenv, tcomp ebody cenv1)
    | Prim(ope, e1, e2) ->
      TPrim(ope, tcomp e1 cenv, tcomp e2 cenv);;
```

This compiler simply replaces symbolic variable names by numeric variable indexes; it therefore also drops the name of the bound variable from each `let`-binding.

Note that the compile-time environment `cenv` 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). Function `getindex` (not shown) uses `cenv` to map a symbolic name `x` to its integer variable index, simply by looking for the first occurrence of `x` in the `cenv` list.

Correspondingly, the run-time environment `renv` in the `teval` interpreter shown below is an `int list` storing the values of the variables in the same order as their names in the compile-time environment `cenv`. 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, Sect. 5.6): the number of binders between an occurrence of a variable and its binding.

The `teval: texpr -> int list -> int` evaluator for `texpr` can be defined like this:

```
let rec teval (e : texpr) (renv : int list) : int =
    match e with
    | TCstI i -> i
    | TVar n  -> List.nth renv n
    | TLet(erhs, ebody) ->
      let xval = teval erhs renv
      let renv1 = xval :: renv
      teval ebody renv1
    | TPrim("+", e1, e2) -> teval e1 renv + teval e2 renv
    | TPrim("*", e1, e2) -> teval e1 renv * teval e2 renv
    | TPrim("-", e1, e2) -> teval e1 renv - teval e2 renv
    | TPrim _            -> failwith "unknown primitive";;
```

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 vari-

able 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:

`teval (tcomp e []) []` equals `eval 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 `[]` produces the same result as evaluation of the source expression `e` using the `eval` interpreter and an empty environment `[]`.

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 that implements an abstract machine) for evaluation of expressions in *postfix* or *reverse Polish* form.

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 Fig. 2.3. Each rule says how the execution of one instruction causes the machine to 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 `reval`:

```
let rec reval (inss : rinstr list) (stack : int list) : int =
    match (inss, stack) with
    | ([], v :: _) -> v
    | ([], [])     -> failwith "reval: no result on stack!"
```

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

Fig. 2.3 Stack machine instructions for expression evaluation. The stack top is to the right

```
| (RCstI i :: insr,              stk)  ->
  reval insr (i::stk)
| (RAdd    :: insr, i2 :: i1 :: stkr) ->
  reval insr ((i1+i2)::stkr)
| (RSub    :: insr, i2 :: i1 :: stkr) ->
  reval insr ((i1-i2)::stkr)
| (RMul    :: insr, i2 :: i1 :: stkr) ->
  reval insr ((i1*i2)::stkr)
| (RDup    :: insr,       i1 :: stkr) ->
  reval insr (i1 :: i1 :: stkr)
| (RSwap   :: insr, i2 :: i1 :: stkr) ->
  reval insr (i1 :: i2 :: stkr)
| _ -> failwith "reval: too few operands on stack";;
```

The machine terminates when there are no more instructions to execute. The result of a computation is the value `v` 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 infix 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 printers. By contrast, Portable Document Format (PDF), also from Adobe Systems, is not a full-fledged programming language.

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 Postscript's `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 Postscript for-loop expects four values on the stack: a start value, a step value, 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 or MacOS, 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 much fancier Postscript example is found in file `sierpinski.eps`. It defines a recursive function that draws a *Sierpinski curve*, 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.

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 the run-time environments `renv` in `teval` (Sect. 2.4) and `stack` in `reval` (Sect. 2.5) 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 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 purpose of the `SSwap, SPop` instruction sequence is to discard the let-bound value stored below the stack top.

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 []`

More general functional languages are often compiled to stack machine code with stack offsets for variables, using 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 little 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 argument (`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 Java method `seval` shown in Fig. 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];        // eval and env stack
    int sp = -1;                        // stack top pointer
    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];
  }
}
```

Fig. 2.4 Stack machine in Java for expression evaluation (file `Machine.java`)

2.9 History and Literature

Reverse Polish form (Sect. 2.5) is named after the Polish philosopher and mathematician Łukasiewicz (1878–1956). The first description of compilation of arithmetic expressions to stack operations was given in 1960 by Bauer and Samelson [3].

The stack-based language Forth (ca. 1968) is an ancestor of Postscript. It is used in embedded systems to control scientific equipment, satellites etc. The Postscript Language Reference [1] can be downloaded from Adobe Corporation.

Kamin's 1990 textbook [2] presents a range of programming languages through interpreters.

2.10 Exercises

The goal of these exercises is to understand the compilation and evaluation of simple arithmetic expressions with variables and let-bindings.

Exercise 2.1 Extend the expression language `expr` from `Intcomp1.fs` with multiple *sequential* let-bindings, such as this (in concrete syntax):

```
let x1 = 5+7   x2 = x1*2 in x1+x2 end
```

To evaluate this, the right-hand side expression `5+7` must be evaluated and bound to `x1`, and then `x1*2` must be evaluated and bound to `x2`, after which the let-body `x1+x2` is evaluated.

The new abstract syntax for `expr` might be

```
type expr =
  | CstI of int
  | Var of string
  | Let of (string * expr) list * expr    (* CHANGED *)
  | Prim of string * expr * expr
```

so that the `Let` constructor takes a list of bindings, where a binding is a pair of a variable name and an expression. The example above would be represented as:

```
Let ([("x1", ...); ("x2", ...)], Prim("+", Var "x1", Var "x2"))
```

Revise the `eval` interpreter from `Intcomp1.fs` to work for the `expr` language extended with multiple sequential let-bindings.

Exercise 2.2 Revise the function `freevars : expr -> string list` to work for the language as extended in Exercise 2.1. Note that the example expression in the beginning of Exercise 2.1 has no free variables, but `let x1 = x1+7 in x1+8 end` has the free variable `x1`, because the variable `x1` is bound only in the body (`x1+8`), not in the right-hand side (`x1+7`), of its own binding. There *are* programming languages where a variable can be used in the right-hand side of its own binding, but ours is not such a language.

Exercise 2.3 Revise the `expr`-to-`texpr` compiler `tcomp : expr -> texpr` from `Intcomp1.fs` to work for the extended `expr` language. There is no need to modify the `texpr` language or the `teval` interpreter to accommodate multiple sequential let-bindings.

Exercise 2.4 Write a bytecode assembler (in F#) that translates a list of bytecode instructions for the simple stack machine in `Intcomp1.fs` into a list of integers. The integers should be the corresponding bytecodes for the interpreter in `Machine.java`. Thus you should write a function `assemble : sinstr list -> int list`.

Use this function together with `scomp` from `Intcomp1.fs` to make a compiler from the original expressions language `expr` to a list of bytecodes `int list`.

You may test the output of your compiler by typing in the numbers as an `int` array in the `Machine.java` interpreter. (Or you may solve Exercise 2.5 below to avoid this manual work).

Exercise 2.5 Modify the compiler from Exercise 2.4 to write the lists of integers to a file. An F# list `inss` of integers may be output to the file called `fname` using this function (found in `Intcomp1.fs`):

```
let intsToFile (inss : int list) (fname : string) =
    let text = String.concat " " (List.map string inss)
    System.IO.File.WriteAllText(fname, text)
```

Then modify the stack machine interpreter in `Machine.java` to read the sequence of integers from a text file, and execute it as a stack machine program. The name of the text file may be given as a command-line parameter to the Java program. Reading numbers from the text file may be done using the StringTokenizer class or StreamTokenizer class; see e.g. *Java Precisely* [4, Example 145].

It is essential that the compiler (in F#) and the interpreter (in Java) agree on the intermediate language: what integer represents what instruction.

Exercise 2.6 Now modify the interpretation of the language from Exercise 2.1 so that multiple let-bindings are *simultaneous* rather than sequential. For instance,

```
let x1 = 5+7   x2 = x1*2 in x1+x2 end
```

should still have the abstract syntax

```
Let ([("x1", ...); ("x2", ...)], Prim("+", Var "x1", Var "x2"))
```

but now the interpretation is that all right-hand sides must be evaluated before any left-hand side variable gets bound to its right-hand side value. That is, in the above expression, the occurrence of `x1` in the right-hand side of `x2` has nothing to do with the `x1` of the first binding; it is a free variable.

Revise the `eval` interpreter to work for this version of the `expr` language. The idea is that all the right-hand side expressions should be evaluated, after which all the variables are bound to those values simultaneously. Hence

```
let x = 11 in let x = 22  y = x+1 in x+y end end
```

should compute `12 + 22` because `x` in `x+1` is the outer `x` (and hence is 11), and `x` in `x+y` is the inner `x` (and hence is 22). In other words, in the let-binding

```
let x1 = e1  ...  xn = en in e end
```

the scope of the variables `x1 ... xn` should be `e`, not `e1 ... en`.

Exercise 2.7 Define a version of the (naive) Fibonacci function

```
let rec fib n = if n<2 then n else fib(n-1) + fib(n-2)
```

in Postscript. Compute Fibonacci of 0, 1, . . . , 25.

Exercise 2.8 Write a Postscript program to compute the sum $1 + 2 + \cdots + 1000$. It must really do the summation, not use the closed-form expression $\frac{n(n+1)}{2}$ with $n = 1000$. (Trickier: do this using only a for-loop, no function definition).

References

1. Adobe Systems: Postscript Language Reference, 3rd edn. Addison-Wesley, Reading (1999). At http://www.adobe.com/products/postscript/pdfs/PLRM.pdf
2. Kamin, S.: Programming Languages: An Interpreter-Based Approach. Addison-Wesley, Reading (1990)
3. Samelson, K., Bauer, F.L.: Sequential formula translation. Commun. ACM **3**(2), 76–83 (1960)
4. Sestoft, P.: Java Precisely, 2nd edn. MIT Press, Cambridge (2005)

http://www.springer.com/978-1-4471-4155-6

Programming Language Concepts
Sestoft, P.
2012, XIV, 278 p. 17 illus., Softcover
ISBN: 978-1-4471-4155-6