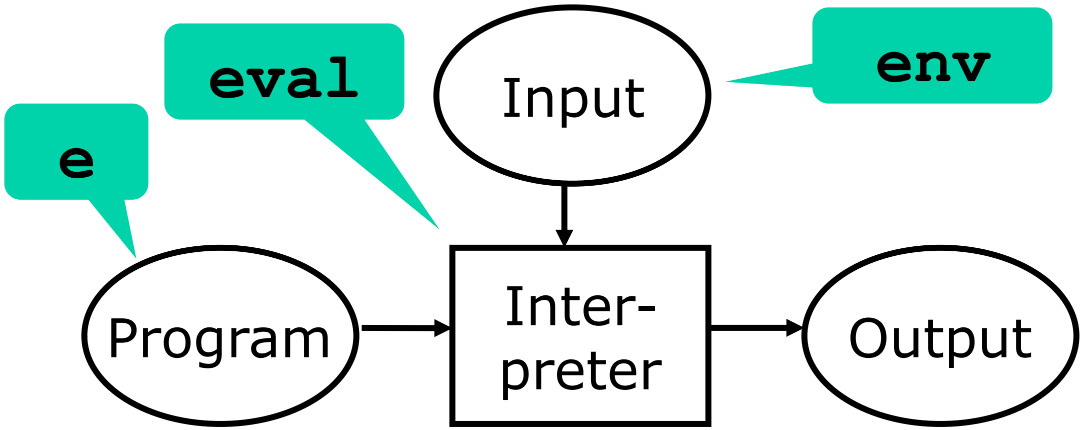
Programs as data
Interpretation vs compilation, stack machines

Peter Sestoft

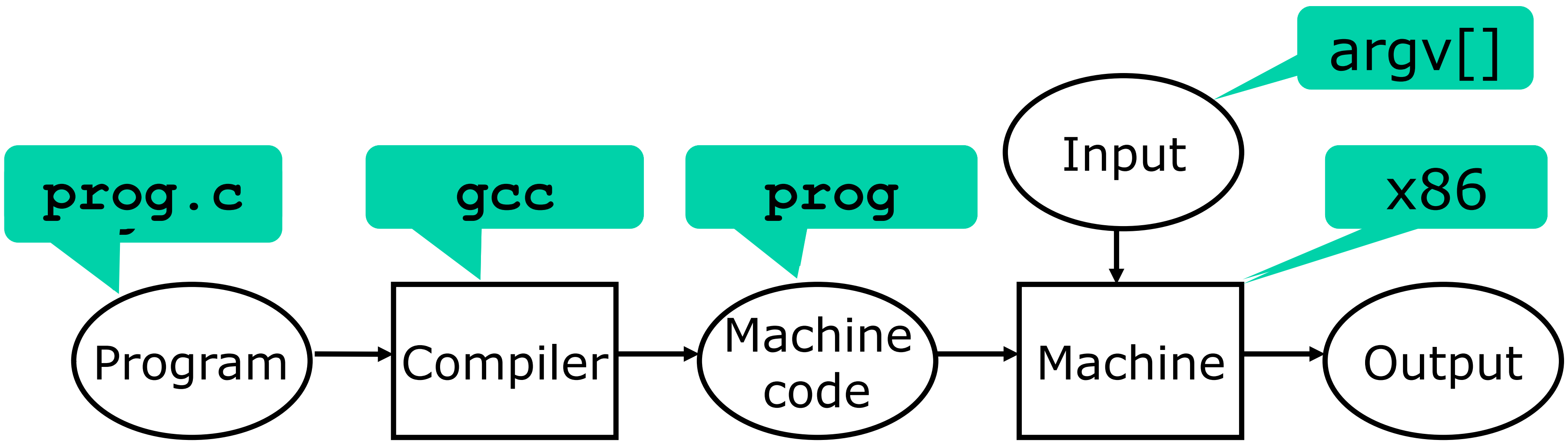
Monday 2012-09-03

Interpretation and compilation

- Interpretation = one-stage execution/evaluation:



- Compilation = two-stage execution/evaluation:



Why compilation?

- Better correctness and safety. The compiler can:
  - check that all names are defined: classes, methods, fields, variables, types, functions, …
  - check that the names have the correct type
  - check that it is legal to refer to them (not private etc)
  - improve the code, e.g. inline calls to private methods
- Better performance
  - The compiler checks are performed once, but the machine code gets executed again and again
- Why *not* compilation?
  - Compilation reduces flexibility by imposing static type checks and static name binding
  - Web programming often requires more flexibility
  - … hence PHP, Python, Ruby, JavaScript, VB.NET, …

Set operations in F#

- We represent a set as a list without duplicates; simple but inefficient for large sets
- The empty set $\varnothing$ is represented by []
- Set membership: $x \in vs$

```
let rec mem x vs =
    match vs with
    | []    -> false
    | v::vr -> x=v || mem x vr;;
```

```
> mem 42 [2; 5; 3];;
val it : bool = false
> mem 42 [];;
val it : bool = false
> mem 42 [2; 67; 42; 5];;
val it : bool = true
```

Set union and difference in F#

- Set union: A ∪ B

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

- Set difference: A \ B

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

Back to expressions: let-bindings

let z = 17 in z + z

body

rhs = right-hand side

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

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

- How represent

  let z=x in z+x

  let z=3 in let y=z+1 in z+y

  let z=(let x=4 in x+5) in z*2

Evaluation of expressions with let

```
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";;
```

- To evaluate "**`let x=erhs in ebody`**":
  - Evaluate **`erhs`** in given environment to get **`xval`**
  - Extend **`env`** with binding **`(x, xval)`** binding to get **`env1`**
  - Evaluate **`ebody`** in **`env1`**

Concepts: Free and bound variable occurrences

- A variable occurrence **`x`** is *bound* if it is in the **`ebody`** of a binding **`let x=erhs in ebody`**
- Otherwise it is *free*
- Which occurrences are bound and which free here:

  let z=x in z+x

  let z=3 in let y=z+1 in x+y

  let z=(let x=4 in x+5) in z*2

  let z=(let x=4 in x+5) + x in z*2

- A variable is *free* if it has some free occurrence
- Usually, a program must have no free variables...
- (… in C it may, but then must be bound by linking)

Finding the set of free variables

```
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)
```

- An expression is *closed* if it has no free variables

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

Substitution: replace free variables

- The substitution **`[(5-4)/z](y*z)`** replaces free **`z`** by expression **`(5-4)`** in expr. **`(y*z)`**
- The result is **`(y*(5-4))`**

- Think of **`[(5-4)/z]`** as an environment that maps **`z`** to **`(5-4)`**

  Like **`[("z", Prim("-", CstI 5, CstI 4))]`**
- A variable not mentioned maps to itself:

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

Substitution, continued

- Substitution affects only free occurrences of **`z`**
- So what is the expected result of
  `[(5-4)/z](let z=22 in y*z end)`   **??**

- And what is the expected result of
  `[(5-4)/z](z + let z=22 in y*z end)`   **??**

- Remove **`z`** from environment when processing **`body`** of **`let z = rhs in body end`**

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

Naive implementation of substitution

- Substitution recursively transforms expr. e:

```
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)
```

replace x (maybe with x)

replace in erhs

not in ebody

recursively in operands

- Apparently this works:

```
> 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))
```

- Also **`[(5-4)/z](let z=22 in y*z end)`** gives **`let z=22 in y*z end`** as it should

Problem: Capture of free variables

- But replacing **`y`** by **`z`**,
  as in **`[z/y](let z=22 in y*z end)`**
  gives **`let z=22 in z*z end`**
- The free variable z that was substituted in for variable y was *captured* by let z=...

- In a substitution [e/y]... free variables in e should remain free

Capture-avoiding substitution

- To avoid capture of new **free** variables, rename existing **bound** variables
- Easy: Invent fresh names, substitute for old

```
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)
```

rename bound variable x

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

make fresh variables

Replacing variable names with indices

- After compilation, there are no variable names, only indices (locations), at runtime
- Instead of symbolic names:

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

we shall use variable indexes:

```
Let(CstI 17, Prim("+", Var 0, Var 0))
```

No variable name

0 means closest variable binding

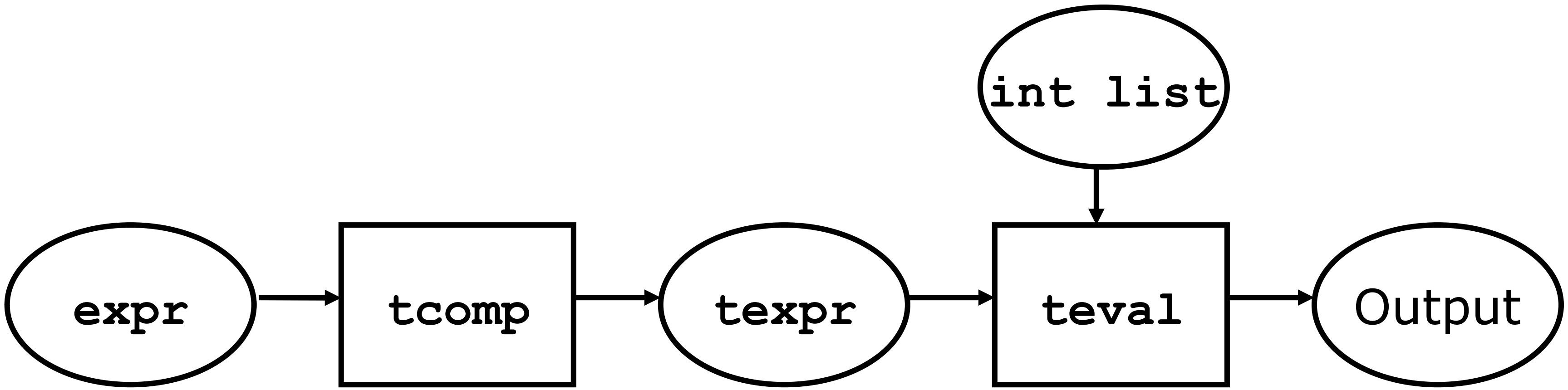
- Index = number of let-bindings to cross:

```
Let("z", CstI 17, Let("y", CstI 25,
                      Prim("+", Var "z", Var "y")))
```

Indexes instead of variable names

- We shall compile to this “target” language:

```
type texpr =                          (* target expressions *)
  | TCstI of int
  | TVar of int                       (* index at runtime   *)
  | TLet of texpr * texpr
  | TPrim of string * texpr * texpr
```



Evaluating texprs

- The runtime environment of a texpr is a list of values – not (name, value) pairs

```
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"
```

Replacing variable names with indices

```
let rec getindex vs x =
    match vs with
      | []     -> failwith "Variable not found"
      | y::yr -> if x=y then 0 else 1 + getindex yr x;;

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
     in TLet(tcomp erhs cenv, tcomp ebody cenv1)
   | Prim(ope, e1, e2) -> TPrim(ope, tcomp e1 cenv, tcomp e2 cenv)
```

let z=3 in let y=z+1 in z+y

`[]`   `["z"]`   `["y"; "z"]`

- What if the expression **`e`** is not closed?

Binding-times in the environment

- Run-time environment in expr interpreter:
  `[("y", 4); ("z", 3)]`
- Compile-time environment in expr compiler:
  `["y"; "z"]`
- Run-time environment of texpr “machine”:
  `[4; 3]`

- The interpreter runtime environment splits to
  - A compile-time environment in the compiler
  - A runtime environment in the “machine”
- We meet such “binding-time” separation again later…

Towards more machine-like code

- Consider expression `2 * 3 + 4 * 5`
- Write it in *postfix*: `2 3 * 4 5 * +`
- This is sequential code for a *stack machine*:

Instructions:	Stack contents:
`2 3 * 4 5 * +`	`2`
`3 * 4 5 * +`	`2 3`
`* 4 5 * +`	`6`
`4 5 * +`	`6 4`
`5 * +`	`6 4 5`
`* +`	`6 20`
`+`	`26`

10-minute exercises

- What is the postfix of

  2 * 3 + 4

  2 + 3 * 4

  2 * (3 + 4)

  2 – 3 – 4 – 5

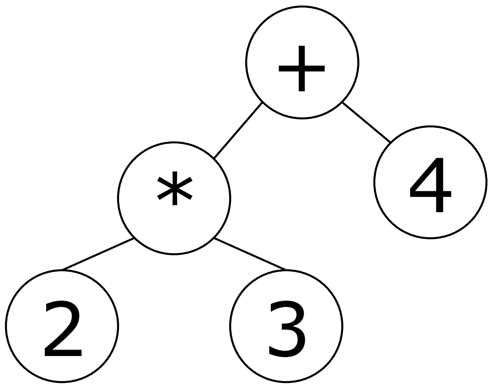
  2 – (3 – (4 – 5))

  2 + 3 * 4 / 5

- Evaluate the postfix versions using a stack

Expression stack machine without variables

Instruction	Stack before	Stack after	Effect
RCSTI n	s	s, n	Push const
RADD	s, n1, n2	s, n1+n2	Add
RSUB	s, n1, n2	s, n1-n2	Subtract
RMUL	s, n1, n2	s, n1*n2	Multiply
RDUP	s, v	s, v, v	Duplicate top elem
RSWAP	s, v1, v2	s, v2, v1	Swap



Compilation of expr to stack machine code

- A constant **`i`** compiles to code **`[RCst i]`**
- An operator application **`e1+e2`** compiles to:
  - code for operand **`e1`**
  - code for operand **`e2`**
  - code for the operator **+**

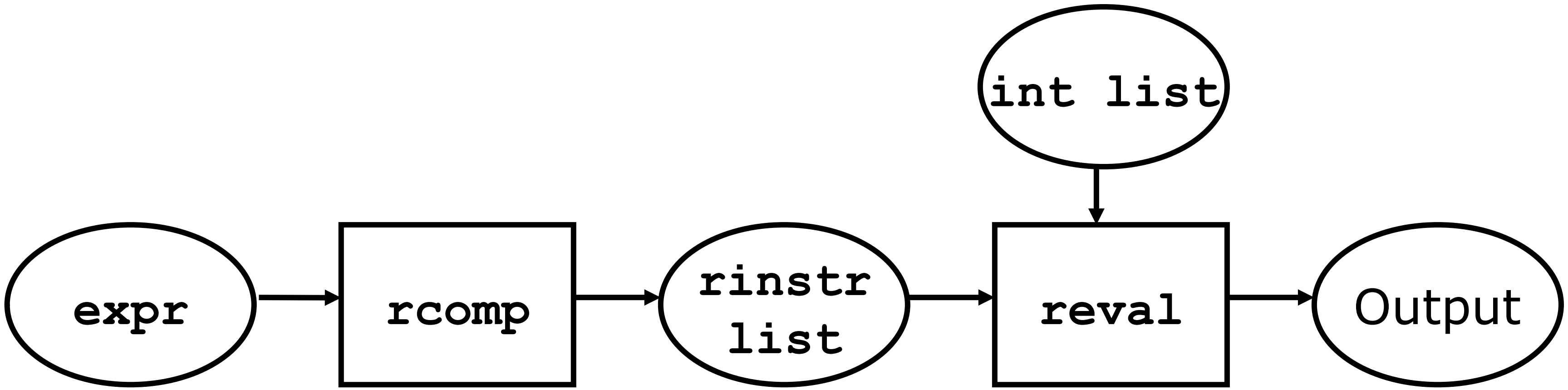
```
let rec rcomp (e : expr) : rinstr list =
    match e with
    | CstI i              -> [RCstI i]
    | Var _               -> failwith "rcomp cannot do Var"
    | Let _               -> failwith "rcomp cannot do Let"
    | Prim("+", e1, e2) -> rcomp e1 @ rcomp e2 @ [RAdd]
    | Prim("*", e1, e2) -> rcomp e1 @ rcomp e2 @ [RMul]
    | Prim("-", e1, e2) -> rcomp e1 @ rcomp e2 @ [RSub]
    | Prim _              -> failwith "unknown primitive";;
```

```
rcomp (Prim("+", Prim("*", CstI 2, CstI 3), CstI 4));;
val it : rinstr list = [RCstI 2; RCstI 3; RMul; RCstI 4; RAdd]
```

Stack machine (without variables)

- A direct implementation of state transitions:

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



Concepts

- An expression e is compiled to a sequence of instructions
- **Net effect principle**:
  - The *net effect* of executing the instructions is to leave the expression’s value on the stack
- *Compiler correctness* relative to interpreter
  - Executing the compiled code gives the same result as executing the original expression
  - That is:

    `reval (rcomp e []) []` equals **`eval e []`**

Stack machines everywhere

- Burroughs B5000 (1961) hardware
- Forth virtual machine (1970)
- P-code, UCSD Pascal (1977)
- Western Digital Pascal microEngine hardware
- Postscript (1984)
- Java Virtual Machine (1994)
- picoJava JVM core hardware
- .NET Common Language Runtime (1999)
- ARM Jazelle instructions (2005) hardware
- Intel cpu stack pointer prediction hardware
- … zillions of others

Postscript (.ps) is a postfix, stack-based language

- A Postscript printer is an interpreter:

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

(4 + 5) * 8

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

let x=7 in x*x+9

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

n!, factorial function

`gs -sNODISPLAY` on ssh.itu.dk

How store (let-bound) variables?

- Idea: Put them in the stack! Classic, 1960'es
- So stack contains mixture of
  - intermediate results (as before)
  - values of bound variables
- To get a variable's value, index off the stack top
- Example: **`2 * let x=3 in x+4 end`**
- Code: **`2 3 SVAR(0) 4 SADD SSWAP SPOP SMUL`**

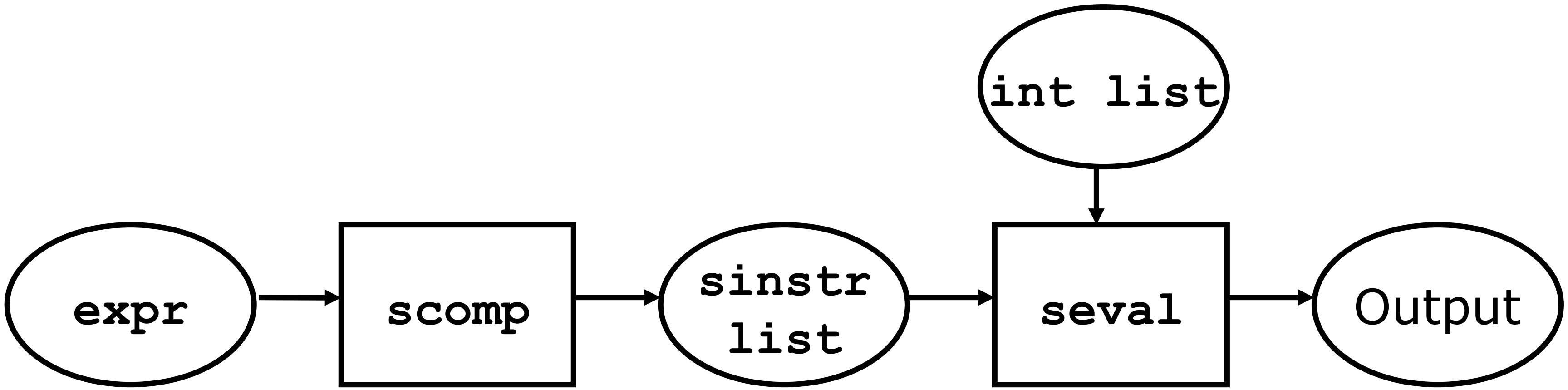
Instructions:	Stack:
`2 3 SVAR(0) 4 SADD SSWAP SPOP SMUL`	`2`
`3 SVAR(0) 4 SADD SSWAP SPOP SMUL`	`2 3`
`SVAR(0) 4 SADD SSWAP SPOP SMUL`	`2 3 3`
`4 SADD SSWAP SPOP SMUL`	`2 3 3 4`
`SADD SSWAP SPOP SMUL`	`2 3 7`
`SSWAP SPOP SMUL`	`2 7 3`
`SPOP SMUL`	`2 7`
`SMUL`	`14`

Value of let-rhs is put on stack top

Must be removed after let-body

Expression stack machine with variables

Instruction	Stack before	Stack after	Effect
SCSTI n	s	s, n	Push const
SVAR x	s	s, s[x]	Index into stack
SADD	s, n1, n2	s, n1+n2	Add
SSUB	s, n1, n2	s, n1-n2	Subtract
SMUL	s, n1, n2	s, n1*n2	Multiply
SPOP	s, v	s	Remove top elem
SSWAP	s, v1, v2	s, v2, v1	Swap



Stack machine (with vars) in F#

```
let rec seval (inss : sinstr list) (stack : int list) =
   match (inss, stack) with
    | ([], v :: _) -> v
    | ([], [])      -> failwith "seval: no result on stack"
    | (SCstI i :: insr,           stk) -> seval insr (i :: stk)
    | (SVar i  :: insr,           stk) -> seval insr (List.nth stk i :: stk)
    | (SAdd    :: insr, i2::i1::stkr) -> seval insr (i1+i2 :: stkr)
    | (SSub    :: insr, i2::i1::stkr) -> seval insr (i1-i2 :: stkr)
    | (SMul    :: insr, i2::i1::stkr) -> seval insr (i1*i2 :: stkr)
    | (SPop    :: insr,     _ :: stkr) -> seval insr stkr
    | (SSwap   :: insr, i2::i1::stkr) -> seval insr (i1::i2::stkr)
    | _ -> failwith "seval: too few operands on stack";;
```

```
type sinstr =
    | SCstI of int
    | SVar of int
    | SAdd
    | SSub
    | Smul
    | Spop
    | SSwap
```

This **`seval`** “machine” combines

- **`teval`**: variables as indices
- **`reval`**: stack machine code

Compiling to the seval “machine”

- The compile-time env. must distinguish between intermediate results and let-bound variables:

```
type stackvalue =
  | Value                                    (* A computed value *)
  | Bound of string;;                        (* A bound variable *)
```

```
let rec scomp (e:expr) (cenv : stackvalue list) : sinstr list =
  match e with
  | CstI i -> [SCstI i]
  | Var x  -> [SVar (getindex cenv (Bound x))]
  | Let(x, erhs, ebody) ->
    scomp erhs cenv @ scomp ebody (Bound x :: cenv)
                    @ [SSwap; SPop]
  | Prim("+", e1, e2) ->
    scomp e1 cenv @ scomp e2 (Value :: cenv) @ [SAdd]
  | Prim("-", e1, e2) ->
    scomp e1 cenv @ scomp e2 (Value :: cenv) @ [SSub]
  | Prim("*", e1, e2) ->
    scomp e1 cenv @ scomp e2 (Value :: cenv) @ [SMul]
  | Prim _ -> failwith "scomp: unknown operator";;
```

seval stack machine in Java (almost C)

```
while (pc < code.length)
  switch (instr = code[pc++]) {
  case SCST:
    stack[sp+1] = code[pc++]; sp++; break;
  case SVAR:
    stack[sp+1] = stack[sp-code[pc++]]; sp++; break;
  case SADD:
    stack[sp-1] = stack[sp-1] + stack[sp]; sp--; break;
  case SSUB:
    stack[sp-1] = stack[sp-1] - stack[sp]; sp--; break;
  case SMUL:
    stack[sp-1] = stack[sp-1] * stack[sp]; sp--; break;
  case SPOP:
    sp--; break;
  case SSWAP:
    { int tmp      = stack[sp];
      stack[sp]    = stack[sp-1];
      stack[sp-1] = tmp;
      break; }
  default:
    throw new RuntimeException("Illegal instruction");
}
```

code : int[]
pc = program counter, points into code
stack : int[]
sp = stack pointer, points into stack