

- More Standard ML: polymorphic types and type variables
- Expressions with let-bound variables
- Interpreters and one-stage execution; compilers and two-stage execution
- Dynamic semantics (run-time) versus static semantics (compile-time)
- Static checking of expressions: no unbound variables
- Postfix or reverse Polish notation
- Postscript, a stack-based language
- Abstract machines with stacks

TF-C

SML polymorphic types and type variables

```
fun len [] = 0
  | len (x::xr) = 1 + len xr;

> val 'a len = fn : 'a list -> int
```

The type says: for any type 'a, function len has type 'a list -> int.

The 'a is a *type variable*: it may be instantiated with any type.

That is, len works for int list and string list and (string*int) list and so on.

SML equality type variables

```
fun member x [] = false
  | member x (y::yr) = x=y orelse member x yr;

fun workday d = member d ["Mon", "Tue", "Wed", "Thu", "Fri"];

> val 'a member = fn : 'a -> 'a list -> bool
```

The 'a is an *equality type variable*: it may be instantiated only with types that admit equality (=) comparison.

For any type t that admits equality, function member has type t -> t list -> bool.

That is, member works only on data that can be compared for equality. Almost all data can, but functions cannot.

SML: polymorphic datatypes

A tree is a tree ... regardless of the type of node values:

```
datatype 'a tree =
  Lf
  | Br of 'a * 'a tree * 'a tree;

fun sumtree Lf = 0
  | sumtree (Br(v, t1, t2)) = v + sumtree t1 + sumtree t2;
```

Creating a list of tree nodes

```
fun preorder1 Lf = []
  | preorder1 (Br(v, t1, t2)) = v :: preorder1 t1 @ preorder1 t2

So preorder1 (Br(1, Br(2, Lf, Lf), Br(3, Lf, Lf))) is [1, 2, 3].
```

A more efficient version: avoid using the linear-time append (@) operator:

```
fun preco Lf acc = acc
  | preco (Br(v, t1, t2)) acc = v :: preco t1 (preco t2 acc)
fun preorder t = preco t []
```

The result of preco is accumulated in the *accumulating parameter* acc.

TF-C

Various kinds of type polymorphism

- Parametric polymorphism, as in Standard ML, Generic Java, and Generic C#:

The type variable 'a is a parameter that may range over arbitrary types.

A parametric polymorphic function works the same way regardless how the type variables are instantiated.

- Bounded parametric polymorphism, as in Standard ML equality types (and Generic Java, Generic C#):

The type variable 'a is a parameter that may range over all types with certain properties.

- Ad hoc polymorphism, or overloading:

The Java plus operator (+) may be used at certain types: int, double, String, ...

- Sometimes, 'polymorphism' is used to describe virtual method calls in object-oriented languages:

The method call o.m(...) may call different methods depending on the class of the object bound to o.

A parametric polymorphic type is a strong assertion about a function. Examples:

There is only one terminating pure function of type 'a -> 'a.

There is only one terminating pure function of type 'a * 'b -> 'b * 'a.

TF-C

TF-C

Object language expressions with variable bindings and nested scope

```
let z = 17 in z + z
let z = 17 in (let z = 22 in 100 * z end) + z
```

Abstract syntax

```
datatype expr =
  CStI of int
| Var of string * expr * expr
| Let of string * expr list
| Prim of string * expr list

val e1 = Let("z", CStI 17, Prim("+", [Var "z", Var "z"]));
val e2 = Let("z", CStI 17,
  Prim("+", [Let("z", CStI 22, Prim("*", [CStI 100, Var "z"])],
    Var "z"]));
```

IFC

Evaluation of expressions with variables and nested scope

The environment env binds variables to their values.

Let-binding *let x = e1 in ebody* evaluates *e1*s and binds *x* to the result during the evaluation of *ebody*:

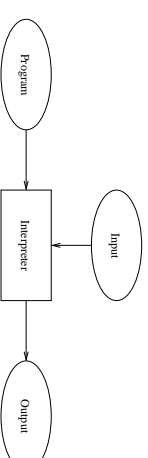
```
fun lookup []      x = raise Fail (x ^ " not found")
  | lookup ((y, v)::r) x = if x=y then v else lookup r x;

fun eval (e : expr) (env : (string * int) list) : int =
  case e of
  CStI i => i
  | Var x => lookup env x
  | Let(x, e1s, ebody) =>
      let val xv1 = eval e1s env
          in eval ebody env1 end
  | 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 _      => raise Fail "unknown primitive"

fun eval0 e = eval e []
```

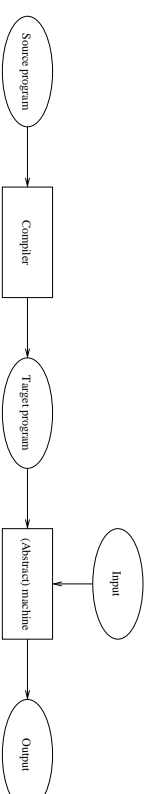
IFC

Interpretation and one-stage execution



An interpreter takes a program and possibly some input, and executes the program to produce some output. All the eval functions we have seen are simple interpreters.

Compilation and two-stage execution



A compiler takes a source program, checks that it is well-formed, and then generates a target program. The target program then may be executed by an (abstract) machine, to produce some output. Because the compiler has done the checks, the (abstract) machine need not do them, and execution is faster.

IFC

Interpreter and compiler: dynamic and static semantics

The *dynamic semantics* of a language determines the effect of a program when executed.

The *dynamic semantics* of a language is typically realized by an interpreter or (abstract) machine.

It may be described informally (the *Java Language Specification*) or formally (the *Definition of Standard ML*).

The *static semantics* of a language is the collection of well-formedness requirements on programs.

The static semantics of a language is typically realized by the checks performed by a compiler.

For instance, a compiler usually checks that all variables in a program are bound or declared.

Example: The static semantics of Java requires all variables to be declared, so a Java compiler must reject this:

```
class A {
  public static void main(String[] args) {
    System.out.println(7 + x);
  }
}
```

IFC

Checking that all every variable occurrence is bound (declared)

A variable occurrence bound by an enclosing let-binding is called *bound*; otherwise it is *free*.

In *let x = 1 + z in let y = 2 in x + y end*, variable *x* occurs bound, *z* free, and *y* both bound and free.

A simple check: is every variable occurrence bound by an enclosing let-binding?

If so, we say the expression is *closed*.

This is checked by e.g. Java and Pascal compilers. Classic C compilers check it for variables but not functions.

The call `closedin e env` returns true if every variable in *e* is bound or appears in the list *env* of variables:

```
fun closedin (e : expr) (env : string list) : bool =
  case e of
  CStl i => true
  | Var x => member x env
  | Let(x, eh1, ebody) =>
      let val env1 = x :: env
      in closedin eh1 env andalso closedin ebody env1 end
  | Prim(ope, [e1, e2]) => closedin e1 env andalso closedin e2 env;
```

An expression is closed if it is closed in the empty environment:

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

A closed expression can be evaluated without causing the interpreter to fail.

FC

Programming Languages, F2002

Page 2-9

Checking that all every variable occurrence is bound (declared)

A variable occurrence bound by an enclosing let-binding is called *bound*; otherwise it is *free*.

In *let x = 1 + z in let y = 2 in x + y end*, variable *x* occurs bound, *z* free, and *y* both bound and free.

A simple check: is every variable occurrence bound by an enclosing let-binding?

If so, we say the expression is *closed*.

This is checked by e.g. Java and Pascal compilers. Classic C compilers check it for variables but not functions.

The call `closedin e env` returns true if every variable in *e* is bound or appears in the list *env* of variables:

```
fun closedin (e : expr) (env : string list) : bool =
  case e of
  CStl i => true
  | Var x => member x env
  | Let(x, eh1, ebody) =>
      let val env1 = x :: env
      in closedin eh1 env andalso closedin ebody env1 end
  | Prim(ope, [e1, e2]) => closedin e1 env andalso closedin e2 env;
```

An expression is closed if it is closed in the empty environment:

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

A closed expression can be evaluated without causing the interpreter to fail.

FC

Programming Languages, F2002

Page 2-9

Computing sets of free variables

Alternatively, we may compute the set of free variables and check that it is empty.

We represent sets as lists. This is simple but inefficient; we could use binary trees or hashsets for efficiency.

`union(xs, ys)` is `xs ∪ ys`, the set of all elements in *xs* or *ys*, without duplicates:

```
fun union([], ys) = ys
  | union(x::xr, ys) = if member x ys then union(xr, ys)
  else x :: union(xr, ys)
```

`minus(xs, ys)` is `xs \ ys`, the set of all elements in *xs* but not in *ys*:

```
fun minus([], ys) = []
  | minus(x::xr, ys) = if member x ys then minus(xr, ys)
  else x :: minus(xr, ys)
```

FC

Programming Languages, F2002

Page 2-10

Find all variables that occur free in expression e

```
fun freevars e : string list =
  case e of
```

```
  CStl i => []
  | Var x => [x]
  | Let(x, eh1, ebody) =>
      union (freevars eh1, minus (freevars ebody, [x]))
  | Prim(ope, [e1, e2]) => union (freevars e1, freevars e2)
```

Alternative definition of the closedness check:

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

FC

Programming Languages, F2002

Page 2-11

Towards compilation of expressions

So far variables in expressions have been represented as names (strings).

In compiled programs, variables are usually represented as addresses or indexes or offsets (all integers).

To make expressions more machine-like, we replace symbolic variable names with numerical indexes.

```
datatype texpr =
  TConst of int
  | TVar of int
  | TLet of texpr * texpr
  | TPrim of string * texpr list
  (* t for target expressions *)
  (* index into runtime environment *)
  (* eh1 and ebody *)
```

Translating variable name to variable index at compile-time

A *compile-time environment* `cenv` is a list of variable names.

Function `getindex` determines the *run-time* index of a variable as its position (0,1,...) in `cenv`:

```
fun getindex (cenv : 'a list) (x : 'a) =
  let fun h [] = raise Fail "unknown variable"
      | h (y::yr) = if x=y then 0 else 1 + h yr
  in h cenv end
```

FC

Programming Languages, F2002

Page 2-12

Compiling from `expr` to `texpr`

Function `tcomp` is a compiler from the source language `expr` to the target language `texpr`.

```
fun tcomp (e : expr) (cenv : string list) : texpr =
  case e of
  | CSTI i => TCSTI i
  | Var x => TVar (getIndex cenv x)
  | Let(x, e1, e2) =>
    let val cenv1 = x :: cenv
        in TLet(tcomp e1s cenv1, tcomp e2 cenv1)
  | Prim(ope, [e1, e2]) => TPrim(ope, [tcomp e1 cenv, tcomp e2 cenv])
```

The order of variable names in `cenv` at compile-time must reflect the order of variable values at run-time.

The compiler works only for closed expressions — a free variable cannot be compiled to an index.

What is `Let("x", CSTI 17, Prim("++", [Var "x", CSTI 33]))` compiled to?

What about `Let("x", CSTI 17, Prim("++", [Var "x", Let("x", CSTI 34, Var "x")]))`?

FC

Expressions in postfix notation

Any arithmetic expression can be rewritten in *postfix notation*, without any parentheses:

```
3 + 4      is written as  3 4 +
(7 * 9) + 10  is written as  7 9 * 10 +
7 * (9 + 10) is written as  7 9 10 + *
10 + 7 * 9   is written as  10 7 9 * +
```

Also known from H-P calculators as *RPN* for *reverse Polish notation* after the logician Łukasiewicz (1878–1956).

Observation: An expression written in postfix is a sequence of instructions for a stack machine:

Expression in postfix	Stack
$7\ 9\ *\ 10\ +$	(empty)
$\Rightarrow 9\ *\ 10\ +$	7
$\Rightarrow *\ 10\ +$	9 7
$\Rightarrow 10\ +$	63
$\Rightarrow +$	10 63
\Rightarrow (empty)	73

The stack top is to the left. At the end of the computation, the result is on the stack top.

FC

Execution of target expressions

We define a new interpreter `teval` to evaluate target expressions with variable indexes.

The run-time environment `renv` is a list of variable values (integers for now).

```
fun teval e (renv : int list) : int =
  case e of
  | TCSTI i => i
  | TVar x => List.nth(renv, x)          (* get x'th element of renv *)
  | TLet(e1s, ebody) =>
    let val xval = teval e1s renv
        val renv1 = xval :: renv
        in teval ebody renv1 end
  | 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 _ => raise Fail "unknown primitive"
```

There are no variable names, only variable indexes, at run-time. The names were removed at compile-time.

Correctness on the `tcomp` compiler: `eval e []` equals `teval (tcomp e []) []`.

FC

Stack machines

A stack machine is an abstract machine with an evaluation stack for storing intermediate results.

A 'constant' instruction (such as 7) pushes the constant onto the stack top (shown to the left).

The '+' instruction pops the two top-most values, adds them, and puts the result back on the stack.

An SML datatype for representing stack machine instructions:

```
datatype rinstr = RCSTI of int | RAdd | RSub | RMul | RDup | RSwap
```

The effect of stack machine instructions, schematically:

Instruction	Stack before	Stack after
RCST <i>i</i>	<i>s</i> \Rightarrow	<i>i</i> :: <i>s</i>
RAdd	<i>i</i> ₂ :: <i>i</i> ₁ :: <i>s</i> \Rightarrow	(<i>i</i> ₁ + <i>i</i> ₂) :: <i>s</i>
RSub	<i>i</i> ₂ :: <i>i</i> ₁ :: <i>s</i> \Rightarrow	(<i>i</i> ₁ - <i>i</i> ₂) :: <i>s</i>
RMul	<i>i</i> ₂ :: <i>i</i> ₁ :: <i>s</i> \Rightarrow	(<i>i</i> ₁ * <i>i</i> ₂) :: <i>s</i>
RDup	<i>i</i> ₁ :: <i>s</i> \Rightarrow	<i>i</i> ₁ :: <i>i</i> ₁ :: <i>s</i>
RSwap	<i>i</i> ₂ :: <i>i</i> ₁ :: <i>s</i> \Rightarrow	<i>i</i> ₁ :: <i>i</i> ₂ :: <i>s</i>

FC

An implementation of a stack machine in SML

The stack is represented by a list, with the stack top at the head of the list.

```
fun reveal ([_] : rinstr list) (v :: _) = v
  | reveal ([_] : rinstr list) []      = raise Fail "reveal: no result"
  | reveal (inst :: rest)              stk =
    case (inst, stk) of
      (RCSTI i, _)      => reveal rest (i::stk)
    | (RAdd, i2 :: i1 :: stk) => reveal rest ((i1+i2)::stk)
    | (RSub, i2 :: i1 :: stk) => reveal rest ((i1-i2)::stk)
    | (RMul, i2 :: i1 :: stk) => reveal rest ((i1*i2)::stk)
    | (RDiv, i1 :: stk)      => reveal rest (i1 :: i1 :: stk)
    | (RSwap, i2 :: i1 :: stk) => reveal rest (i1 :: i2 :: stk)
```

When there are no more instructions, the value on the stack top is returned.

Evaluating the expression `10 + 17 * 17`:

```
val rpr1 = reveal [RCSTI 10, RCSTI 17, RDup, RMul, RAdd] [];
```

FIG

Stack machines in the real world: Postscript

Postscript (ca. 1984) is a stack-based programming language used to control high-end printers.

To compute $(4+5) * 8$ in Postscript, write:

```
4 5 add 8 mul
```

To bind `x` to 7 and then compute $x*x+9$ and print it (using the `'='` function):

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

Defining the factorial function (`!n`) and computing the factorial of 0, 1, ..., 12:

```
/fac { dup 0 eq { pop 1 } { dup 1 sub fac mul } ifelse } def
0 1 12 { fac = } for
```

The `gs` or Ghostscript interpreter can be used to experiment with Postscript programs. Under Linux:

```
gs -dNODISPLAY
```

Under MS Windows at FIG, this might work:

```
C:\AIaddin\gs6.01\bin\gswin32 -dNODISPLAY
```

The Java Virtual Machine and Microsoft Common Language Runtime (.NET) are also abstract stack machines.

FIG

Compilation of a variable-free expression to a sequence of stack machine instructions

This is the same as transforming the expression into postfix form.

```
fun rcomp e : rinstr list =
  case e of
    SCSTI i => [RCSTI i]
  | 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 "_"      => raise Fail "unknown primitive"
```

To compile `e1 + e2`, compile `e1` then `e2`, concatenate their instruction sequences, and put `'+'` at the end.

The expression `10 + 17 * 17` compiles to `[RCSTI 10, RCSTI 17, RCSTI 17, RMul, RAdd]`.

Net effect principle

Executing a compiled expression puts the expression's value on top of the stack, changing nothing below it.

Correctness of the `rcomp` compiler: `eval e []` equals `reveal (rcomp e) []`.

FIG

Storing intermediate results and variable bindings in the same stack

Often intermediate results and variables are stored in the same stack.

This is possible when the language has static nested scopes.

```
datatype sinstr =
  SCSTI of int
  | SVar of int
  | SAdd
  | SSub
  | SMul
  | SPop
  | SSwap

(* push integer *)
(* push variable from env *)
(* pop args, push sum *)
(* pop args, push diff. *)
(* pop args, push product *)
(* pop value/unbind var *)
(* exchange top and next *)
```

The expression `let z = 17 in z + z` would be compiled to these instructions:

```
SCSTI 17, SVar 0, SVar 1, SAdd, SSwap, SPop
```

The purpose of `SSwap`, `SPop` is to remove the variable binding (of `z`) below the intermediate result (34).

FIG

A unified-stack abstract machine

```

fun seval ([ ] : sinstr list) (v::_) = v
| seval ([ ] : sinstr list) _ = raise Fail "seval: no result??"
| seval (inst :: rest) stk =
  (case (inst, stk) of
   (SCSTI i, _) => seval rest (i :: stk)
 | (SVar i, _) => seval rest (List.nth(stk, i) :: stk)
 | (SAdd, i2::i1::stkr) => seval rest (i1+i2 :: stkr)
 | (SSub, i2::i1::stkr) => seval rest (i1-i2 :: stkr)
 | (SMul, i2::i1::stkr) => seval rest (i1*i2 :: stkr)
 | (SPOP, _ :: stkr) => seval rest stkr
 | (SSwap, i2::i1::stkr) => seval rest (i1::i2::stkr))

```

The evaluation of SCSTI 17, SVar 0, SVar 1, SAdd, SSwap, SPOP, that is, *let z = 17 in z + z*:

Expression in postfix	Stack
SCSTI 17, SVar 0, SVar 1, SAdd, SSwap, SPOP	(empty)
⇒ SVar 0, SVar 1, SAdd, SSwap, SPOP	17
⇒ SVar 1, SAdd, SSwap, SPOP	17 17
⇒ SAdd, SSwap, SPOP	17 17 17
⇒ SSwap, SPOP	34 17
⇒ SPOP	17 34
⇒ (empty)	34

Compiling for the unified-stack machine

The expression *let x = 17 in x + x* is compiled to:

```
SCst 17, SVar 0, SVar 1, SAdd, SSwap, SPOP
```

The expression *let x = 17 in let z = 22 in 100 * z end + z* is compiled to:

```
SCSTI 17, SCSTI 22, SCSTI 100, SVar 1, SMul, SSwap, SPOP,
SVar 1, SAdd, SSwap, SPOP
```

Bytecodes for stack machines

Internally, each instruction for a stack machine is often represented by a few small integers. For example:

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

Compiling to a machine with only one stack

The compile-time variable environment contains variable names, and dummies. Intrm for intermediate values.

The corresponding run-time environment positions will hold variable values resp. intermediate results.

```

datatype rvalue =
  Bound of string (* A bound variable *)
| Intrm (* An intermediate result *)

```

Compilation to a list of instructions for a unified-stack machine

```

fun scomp e (cenv : rvalue list) : sinstr list =
  case e of
   SCSTI i => [SCSTI i]
 | Var x => [SVar (getIndex cenv (Bound x))]
 | Let(x, e1s, ebody) =>
   scomp e1s cenv @ scomp ebody (Bound x :: cenv) @ [SSwap, SPOP]
 | Prim(ope, [e1, e2]) =>
   scomp e1 cenv
   @ scomp e2 (Intrm :: cenv)
   @ [case ope of
      "+" => SAdd
    | "-" => SSub
    | "*" => SMul
    | _ => raise Fail "Sprim2 operator"]

```

An abstract stack machine in Java (file `stack.java`)

```

class Stack {
  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 variable 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;
        ...
      }
    return stack[sp];
  }
}

```

SML: higher-order functions

```
fun map f [] = []
  | map f (x::xr) = f x :: map f xr;
```

SML: anonymous functions

```
fun double x = 2 * x;
map double [4, 5, 89];
map (fn x => 2 * x) [4, 5, 89];
```

ITC

SML: the tree iterator tfold replaces Br by f and Lf by e

```
fun tfold f e Lf = e
  | tfold f e (Br(v,t1,t2)) = f(v, tfold f e t1, tfold f e t2);
fun sumtree t = tfold (fn (v, r1, r2) => v + r1 + r2) 0 t;
```

ITC

SML: the list iterator foldr

An application foldr f e xs replaces '::' by f and [] by e in list xs:

```
fun foldr f e [] = e
  | foldr f e (x::xr) = f(x, foldr f e xr);
fun len xs = foldr (fn (_, res) => 1+res) 0 xs;
fun sum xs = foldr (fn (x, res) => x+res) 0 xs;
fun prod xs = foldr (fn (x, res) => x*res) 1 xs;
fun map g xs = foldr (fn (x, res) => g x :: res) [] xs;
```

Using foldr and map to compute Freevars for arbitrary primitives

```
fun freevars e : string list =
  case e of
    CstI i => []
  | Var x => [x]
  | Let(x, exs, eboddy) =>
    union (freevars exs, minus (freevars eboddy, [x]))
  | Prim(ope, es) => foldr union [] (map freevars es)
```

ITC