

## Programming Languages, F2002

### Lecture 6, Wednesday 13 March 2002 \*

- A simple imperative language and a naive store model
- Variable declarations, expressions, and statements
- Assignable variables, assignment statements
- If-then-else statements, for-loops, while-loops
- A better model: values and values; environment and store
- Parameter passing mechanisms
- An imperative language (micro-C)
- C arrays, pointers, and pointer arithmetics
- The stack of activation records during function calls
- An interpreter for micro-C

IFC

Programming Languages, F2002

Page 6-1

### A simple imperative language: expressions and statements

#### Evaluation of an expression produces a value

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

#### Execution of a statement produces a modified store

```
datatype stmt =  
  Asgn of string * expr  
  | If of expr * stmt * stmt  
  | Seq of stmt list  
  | For of string * expr * expr * stmt  
  | While of expr * stmt  
  | Print of expr
```

IFC

Programming Languages, F2002

Page 6-3

### A simple imperative language: statements and expressions (file `imp/imp.sml`)

Example 1: Compute  $\text{sum} = 0 + 1 + \dots + 100$ :

```
sum = 0;  
for i = 0 to 100 do  
  sum = sum + i;  
print sum;
```

Example 2: Compute least  $i$  for which  $0 + 1 + 2 + \dots + i \geq 10000$ :

```
i = 1;  
sum = 0;  
while sum < 10000 do begin  
  print sum;  
  sum = sum + i;  
  i = 1 + i;  
end;  
print i;
```

IFC

Programming Languages, F2002

Page 6-2

### Naive machine model: the store maps names to values

A naive store `Naivestore.naivesto` is just a mapping from variable names to variable values:

```
type 'data naivesto (* A map from string to 'data *)  
  
val empty : 'data naivesto  
val get  : 'data naivesto -> string -> 'data  
val set  : 'data naivesto -> string * 'data -> 'data naivesto
```

IFC

Programming Languages, F2002

Page 6-4

**An expression is evaluated in a given store to produce a value:**

```
fun eval e (sto : (string, int) sto) : int =
  case e of
  Cst1 i => i
  | Var x => get sto x
  | Prim(ope, [e1, e2]) =>
    let val i1 = eval e1 sto
        val i2 = eval e2 sto
    in
      case ope of
      "*" => i1 * i2
      | "+" => i1 + i2
      | "-" => i1 - i2
      | "==" => bool2int (i1 = i2)
      | "<" => bool2int (i1 < i2)
      | _ => raise Fail "unknown primitive"
    end
  | Prim _ => raise Fail "unknown primitive"
```

So an expression cannot have side effects: it cannot change the store.

This is a limitation of the model.

In most real languages, expressions can have side effects:  $x++$ ,  $x = 67$ ,....

**A statement is executed in a given store to produce a new modified store:**

```
fun exec stmt (sto : (string, int) sto) : (string, int) sto =
  case stmt of
  Asgn(x, e) => set sto (x, eval e sto)
  | If(e1, stmt1, stmt2) => if int2bool(eval e1 sto) then
    exec stmt1 sto
  else
    exec stmt2 sto
  | Seq stmts =>
    let fun loop [] sto = sto
        | loop (s1::sr) sto = loop sr (exec s1 sto)
    in loop stmts sto end
  | For(x, estart, estop, stmt) =>
    let val start = eval estart sto
        val stop = eval estop sto
    in fun loop i sto1 =
        if i <= stop then
          loop (i+1) (exec stmt (set sto1 (x, i)))
        else
          sto1
        in loop start sto end
    while(e, stmt) =>
      let fun loop sto1 =
          if int2bool(eval e sto1) then
            loop (exec stmt sto1)
          else
            sto1
        in loop sto end
    print e =>
      in loop sto end
  | Print e =>
    (Print (Int.toString (eval e sto))); print "\n"; sto)
```

**A more realistic store model**

A store is a sequence of numbered cells. The cell numbers are called *addresses* or *locations*:

0	1	2	3	4	5	6	7	8	9	10	11	12
		16										

The value of a variable is stored in a given cell. Maybe variable  $i$  is in cell 2. Then executing

```
 $i = 1 + i;$ 
```

will produce this store:

0	1	2	3	4	5	6	7	8	9	10	11	12
		17										

**Lvalue and rvalue of a variable**

In the assignment, variable  $i$  appears in two distinct roles:

```
 $i = 1 + i;$ 
```

In the right-hand side ( $1 + i$ ), we use the contents of  $i$ , that is, 16.

In the left-hand side ( $i$ ), we use the address of  $i$ , that is, 2; we do not care about the 16.

These two aspects are called the variable's *rvalue* and its *lvalue*, for *right* and *left*.

**Some expressions have an lvalue, some do not**

A variable  $i$ , an array indexing  $a[i]$ , and a object field  $o.f$  all have an lvalue and an rvalue.

An arithmetic expression  $y+5$  has an rvalue but no lvalue.

**Some operators require an lvalue, some do not**

Arithmetic operators as in  $(x + 2)$  works on the rvalue of  $x$ .

Assignment  $x = \dots$  and increment/decrement operators  $x++$  work on the lvalue of  $x$ .

Compound assignment  $x += \dots$  works on the rvalue and lvalue of  $x$ .

**New roles for environment and store**

- the environment maps variable names to lvalues, that is, locations;
- the store maps locations to rvalues.

This model can describe e.g. C arrays, C pointer arithmetics, and C# parameter passing mechanisms.

### Parameter passing mechanisms

Consider a call `p(a, b)` to a procedure (or function or method) declared like this:

```
void p(int x, double y) { ... }
```

The argument `a` could be passed to the formal parameter `x` in several ways:

- Call-by-value: a copy of the value of `a` is made in a new store location, which is bound to `x`. Updates to `x` do not affect `a`.

This is used in C, Java, functional languages, ...

- Call-by-reference: the location (value) of `a` is passed to the procedure and bound to `x`.

Updates to `x` will immediately affect `a`.

This is possible in Pascal, C++, C#.

Note that `a` must have an value: `a` may be a variable, array element, object field, or structure field.

- Call-by-value-return: a copy of the value of `a` is made in a new location, which is bound to `x`. When the procedure returns, the value of `x` is copied to `a` if `a` has an value.

This is used only in Fortran (I think).

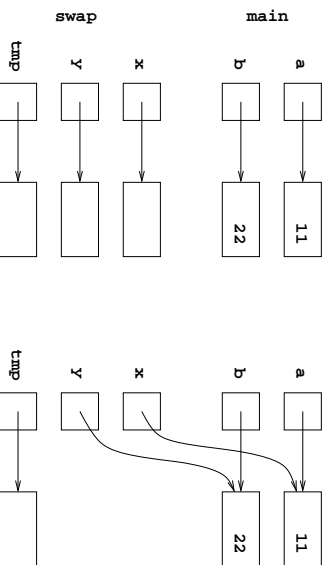
ITC

### Call-by-value and call-by-reference in C#

The C# programming language permits both call-by-value and call-by-reference:

```
void swapV(int x, int y) {
    int tmp = x; x = y; y = tmp;
}
a = 11; b = 22;
swapV(a, b);

void swapR(ref int x, ref int y) {
    int tmp = x; x = y; y = tmp;
}
a = 11; b = 22;
swapR(ref a, ref b);
```



ITC

### Micro-C: a small subset of the C programming language

- C-style expressions, statements, and statement blocks
- C-style declarations of `int` variables, arrays, and pointers
- C-style pointer arithmetics
- C-style function declarations and call-by-value parameter passing
- No type check
- No `return` statement — cumbersome to model abrupt termination in a direct-style interpreter

Our micro-C model is similar to B, an untyped predecessor of C:

CPL (early 1960s) → BCP (1967) → B (1971) → C (1972) → C++ (1984) → Java (1994) → C# (1999)

ITC

### Declarations in C: integers, pointers, arrays

Declaration	Meaning
<code>int i;</code>	<code>i</code> is an integer
<code>int ia[10];</code>	<code>ia</code> is an array of 10 integers
<code>int *p;</code>	<code>p</code> is a pointer to an integer
<code>int *ipa[10];</code>	<code>ipa</code> is an array of 10 pointers to integers
<code>int (*iap)[10];</code>	<code>iap</code> is a pointer to an array of 10 integers

An integer declaration `int i` reserves one memory cell for the integer.

An array declaration `int ia[10]` reserves 10 memory cells for the array's elements.

A pointer declaration `int *p` reserves one memory cell for the pointer.

An array-of-pointer declaration `int *ipa[10]` reserves 10 memory cells for the pointers.

A pointer-to-array declaration `int (*iap)[10]` reserves one memory cell for the pointer.

Array declarations look somewhat similar in Java, but mean something else altogether.

ITC

### Expressions in C: array indexing, pointer arithmetics, address-of

Expression	Lvalue	Rvalue
<code>i</code>	the cell occupied by <code>i</code>	the contents of that cell
<code>ia[4]</code>	cell 4 in array <code>a</code> <code>i</code>	the contents of that cell
<code>*p</code>	the cell pointed to by <code>p</code>	the contents of that cell
<code>*ipa[4]</code> ;	the cell pointed to by the contents of cell 4 in <code>ipa</code>	the contents of that cell
<code>(*iap)[4]</code> ;	cell 4 of the array pointed to by <code>iap</code>	the contents of that cell
<code>*ia</code>	first cell in array <code>a</code> <code>i</code>	the contents of that cell
<code>*(ia+4)</code>	cell 4 in array <code>a</code> <code>i</code>	the contents of that cell
<code>*(p+2)</code>	cell 2 after that pointed to by <code>p</code>	the contents of that cell
<code>&amp;i</code>		the address of the cell occupied by <code>i</code>

In C, adding an integer `i` to a pointer `p` gives a new pointer `(p+i)`.

In C, an array `ia` of integers is actually a pointer to the first array element `ia[0]`.

In C, an array access `a[i]` is actually `*(a+i)`.

### C pointers galore (file `imp/ex2.c`)

```
int *p; // pointer to int
int i; // int
int ia[10]; // array of 10 ints
int *ia2; // pointer to int
int *ipa[10]; // array of 10 pointers to int
int (**iap)[10]; // pointer to array of 10 ints
print i; // ~1
print p; // ~1
p = &i; // now p points to i
print p; // 1
ia2 = ia; // now ia2 points to ia[0]
print *ia2; // ~1
*p = 227; // now i is 227
print p; print i; // 1 227
*&i = 12; // now i is 12
print i; // 12
p = &*p; // no change
print *p; // 12
p = ia; // now p points to ia[0]
*ia = 14; // now ia[0] is 14
print ia[0]; // 14
*(ia+9) = 114; // now ia[9] is 114
print ia[9]; // 114
ipa[2] = p; // now ipa[2] points to i
print ipa[2]; // 1 (true)
print (*ipa[2] == *(ipa+2)); // 1 (true)
iap = &ia; // now iap points to ia
print (**iap)[2] == *((*iap)+2)); // 1 (true)
```

### More micro-C examples

Pass command line argument `n` to `main` and print the numbers `0, 1, ..., n - 1` (file `imp/ex3.c`):

```
void main(int n) {
    int i;
    i=0;
    while (i < n) {
        print i;
        i=i+1;
    }
}
```

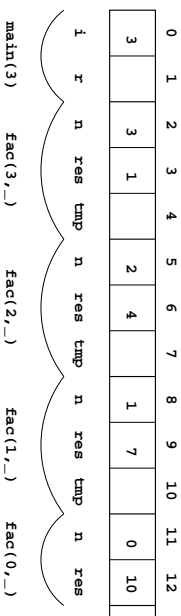
Using a pointer `rp` to simulate call-by-reference (file `ex5.c`)

```
void main(int n) {
    int r;
    square(n, &r);
    print r;
}
void square(int i, int *rp) {
    *rp = i * i;
}
```

### Micro-C example: Recursive function calls (file `imp/ex9.c`)

```
void main(int i) {
    int r;
    fac(i, &r);
    print r;
}
void fac(int n, int *res) {
    print &n;
    if (n == 0)
        *res = 1;
    else {
        int tmp;
        fac(n-1, &tmp);
        *res = tmp * n;
    }
}
```

The store is a stack of activation records:



### Micro-C abstract syntax (file `imp/AbSyn.sml1`)

```
datatype typ =
  | TypI
  | TypC
  | TYPA of typ * expr option
  | TYP of typ
and expr =
  | Access of access * expr
  | Assign of access * expr
  | Addr of access
  | Cst of constant
  | Priml of string * expr
  | ...
and access =
  | AccVar of string
  | AccDeref of expr
  | AccIndex of access * expr
and stmt =
  | Block of stmtordec list
  | ...
and stmtordec =
  | Dec of typ * string
  | Stmt of stmt
and topdec =
  | Fundec of typ option * string * paramdec list * stmt
  | Vardec of typ * string
```

```
(* Type int *)
(* Type char *)
(* Array type *)
(* Pointer type *)
(* Variable or element access *)
(* Assignment to var or arrayElem *)
(* Get address: &e *)
(* Constant *)
(* Strict primitive operator *)
(* Variable access: x *)
(* Pointer dereferencing: *p *)
(* Array indexing: acc[e1] *)
(* Block: grouping and scope *)
(* Declaration of local variable *)
(* A statement *)
```

IFC

### Environment and store in the micro-C interpreter

The environment maps names to locations (`int`) and remembers the next unused store location:

```
type envv = (string, int) env * int
```

The store maps locations to 'data, usually integers:

```
type 'data sto
val empty      : unit -> 'data sto
val getsto    : 'data sto -> int -> 'data
val setsto    : 'data sto -> int -> 'data -> 'data sto
val bindvar   : string -> 'data -> (string, int) Env.env * int
                -> 'data sto -> ((string, int) Env.env * int) * 'data sto
```

Function `bindvar` allocates a new variable `x` both in environment and store:

The call `bindvar x v (env, loc) sto` returns `(env1, sto1)` where

- `env1` maps `x` to `loc` and records that the next unused store location is `loc+1`;
- `sto1` maps `loc` to `v`.

IFC

### Evaluation of micro-C expressions (file `imp/c.sml1`)

Evaluation of an expression requires an environment and a store; it produces a result and an updated store:

```
and eval e env sto : int * sto =
  case e of
    Access acc => let val (loc, sto1) = access acc env sto
                  in (getsto sto1 loc, sto1) end
    Assign(acc, e) => let val (loc, sto1) = access acc env sto
                      val (res, sto2) = eval e env sto1
                      in (res, setsto sto2 loc res) end
    Cst (CstI i) => (i, sto)
    Cst CstN => (~1, sto)
    Addr acc => access acc env sto
    Priml(ope, e1) =>
      let val (i1, sto1) = eval e1 env sto
          val res =
            case ope of
              "!" => bool2int (not (int2bool i1))
              "printi" => (print (int.toString i1); print " "; i1)
              "printc" => (print (str (chr i1)); i1)
              _ => raise Fail "unknown primitive 1"
            in (res, sto1) end
          | Print2(ope, e1, e2) => ...
          | Call(f, es) => callfun f es env sto
```

IFC

### Interpretation of accesses

The interpretation of a variable or array access requires an environment and a store.

It produces an `ival` and a new store:

```
fun access (AccVar x) env sto = (lookup (#1 env) x, sto)
  | access (AccDeref e) env sto =
    let val (a, sto1) = eval e env sto
        in (a, sto1) end
  | access (AccIndex(acc, idx)) env sto =
    let val (a, sto1) = access acc env sto
        val aval = getsto sto1 a
        in (aval + i, sto2) = eval idx env sto1
           in (aval + i, sto2) end
```

A variable is just looked up in the environment to get its `ival`.

A dereferencing expression `*e` evaluates `e` to obtain an `ival`.

An array indexing `acc[e1]` finds the address of array `acc`; this is the `ival` `aval`.

Then it evaluates `e1` to obtain an integer `i`, and returns the `ival` `aval + i`.

The interpretation of an access `*e` or `acc[e1]` may modify the store because it evaluates `e` or `e1`.

IFC

### Execution of micro-C statements

Execution takes an environment and a store; it produces an extended environment and an updated store:

```
fun exec stmt (env : env) (sto : sto) : env * sto =
  case stmt of
  | If(e, stmt1, stmt2) =>
    let val (v, sto1) = eval e env sto
    in
      if !int2bool v then
        (env, #2 (exec stmt1 env sto1))
      else
        (env, #2 (exec stmt2 env sto1))
      end
    end
  | While(e, body) =>
    let fun loop sto1 =
        let val (v, sto2) = eval e env sto1
        in
          if !int2bool v then
            loop (#2 (exec body env sto2))
          else
            sto2
          end
        end
    in (env, loop sto) end
  | Expr e =>
    let val (v, sto1) = eval e env sto
    in (env, sto1) end
  | Block stmts =>
    let fun loop [] = (env, sto)
        | loop (s1::sr) (env, sto) = loop sr (stmtdrdec s1 env sto)
    in val (_, sto1) = loop stmts (env, sto)
    in (env, sto1) end
```