

Programming Languages, F2003

Lecture 6, Wednesday 12 March 2003 *

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

Read: Programming Language Concepts, sections 6.1-6.6; Kernighan and Ritchie (1988) pages 93-114.

IT-C

Programming Languages, F2003

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
```

IT-C

Programming Languages, F2003

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;
```

IT-C

Programming Languages, F2003

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
```

IT-C

Programming Languages, F2003

Page 6-4

An expression is evaluated in a given store to produce a value (file `imp/imp.sml1`)

```
fun eval e (sto : int nativesto) : int =
  case e of
  case e of
  | Cst 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
      "==" => if i1 = i2 then 1 else 0
      "<" => if i1 < i2 then 1 else 0
      _ => 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++`, `....`

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*.

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

```
fun exec stmt (sto : int nativesto) : int nativesto =
  case stmt of
  | Asgn(x, e) =>
    set sto (x, eval e sto)
  | If(e1, stmt1, stmt2) =>
    if eval e1 sto <> 0 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 eval e sto1 <> 0 then
            loop (exec stmt sto1)
          else
            sto1
        in loop start sto end
    | Print e =>
      in loop sto end
    | Print (Int.toString (eval e sto)); print "\n"; sto
```

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)` work only on the rvalue of `x`.

Assignment `x = ...` works only on the lvalue of `x`.

The increment operator `x++` and compound assignment `x += ...` work on both 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).

IT-C

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

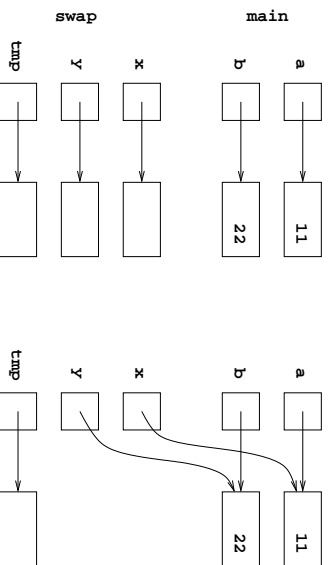
The C# programming language permits both call-by-value and call-by-reference:

```
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);
```



IT-C

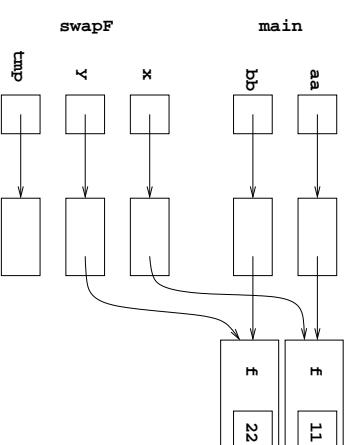
Java has call-by-value (only)

But parameter passing never copies an object (or array), only an object reference.

```
static void swapF(C x, C y) {
    int tmp = x.f; x.f = y.f; y.f = tmp;
}

C aa = new C(11), bb = new C(22);
swapF(aa, bb);

class C {
    public int f;
    public C(int f) {
        this.f = f;
    }
}
```



IT-C

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)

IT-C

Declarations in C: integers, pointers, arrays

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

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>ia</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>ia</code>	the contents of that cell
<code>*(ia+4)</code>	cell 4 in array <code>ia</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>&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)`.

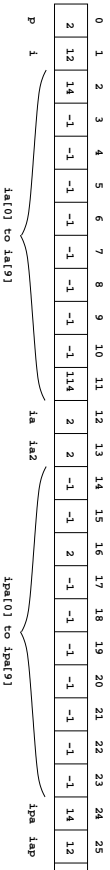
In C, an array `ia` of integers is actually a pointer to the first array element `ia[0]`.

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 ia[0]
print ipa[2]; // 2
print (*ipa[2] == *(ipa+2)); // 1 (true)
iap = &ia; // now iap points to ia
print ((*iap)[2] == *((*iap)+2)); // 1 (true)
```

The store at the end of example `imp/ex2.c`



More micro-C examples

Pass command line argument n to `main` and print the numbers $0, 1, \dots, n-1$ (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 `imp/ex5.c`)

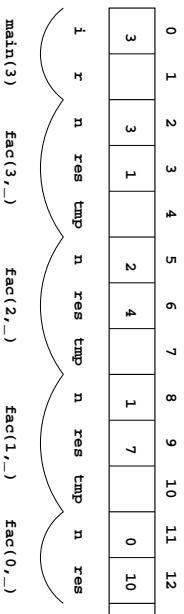
```
void main(int n) {
    int r;
    square(n, &r);
    print r;
}
void square(int i, int *rp) {
    *rp = i * i;
}
```

F-C

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 (or stack frames):



F-C

Micro-C abstract syntax (file `imp/AbSynr.sm1`)

```
datatype typ =
  | TypI
  | TypC
  | TypA of typ * int option
  | TypP of typ
and expr =
  | Access of access
  | Assign of access * expr
  | Addr of access
  | Cst of constant
  | ...
  | Call of string * expr list
and access =
  | AddrVar of string
  | AddrRef of expr
  | AddrIndex of access * expr
and stmt =
  | If of expr * stmt * stmt
  | While of expr * stmt
  | Expr of expr
  | Return of expr option
  | Block of stmtorderc list
and stmtorderc =
  | Dec of typ * string
  | Stmt of stmt
and lopdec =
  | Fundec of typ option * string * (typ * string) list * stmt
  | Vardec of typ * string
(* Type int *)
(* Type char *)
(* Array type *)
(* Pointer type *)
(* x or *p or alej *)
(* x=e or *p=e or alej=e *)
(* &x or &*p or &alej *)
(* Constant *)
(* Function call f(...) *)
(* Variable access *)
(* Dereferencing of a pointer *p *)
(* Array indexing *)
(* Conditional *)
(* While loop *)
(* Expression (as in C or Java) *)
(* Return from method *)
(* Block: grouping and scope *)
(* Declaration of local variable *)
(* A statement *)
```

F-C

Environment and store in the micro-C interpreter

The *environment* maps names to locations (integers) 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`.

F-C

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, stc1) = access acc env sto
                    in (getsto stc1 loc, stc1) end
  Assign(acc, e) => let val (loc, stc1) = access acc env sto
                    val (res, stc2) = eval e env stc1
                    in (res, setsto stc2 loc res) end
  Cst (CstI i)   => (i, stc)
  Cst CstN       => (~1, stc)
  Addr acc       => access acc env sto
  Prim1(ope, e1) =>
    let val (i1, stc1) = eval e1 env sto
    val res =
      case ope of
      "!" => if i1=0 then 1 else 0
      | "print!" => (print (Int.toString i1); print " "; i1)
      | "printc" => (print (str (chr i1)); i1)
      | _ => raise Fail "unknown primitive 1"
    in (res, stc1) end
  Prim2(ope, e1, e2) => ...
  Call(f, es) => callfun f es env sto
```

Interpretation of accesses

The interpretation of an access (to variable, pointer or array element) requires an environment and a store.

It returns an lvalue and a new store:

```
fun access (AccVar x)   env sto = (lookup (#1 env) x, stc)
  | access (AccDeref e) env sto =
    let val (a, stc1) = eval e env sto
    in (a, stc1) end
  | access (AccIndex(acc, idx)) env sto =
    let val (a, stc1) = access acc env sto
        val aval = getsto stc1 a
        val (i, stc2) = eval idx env stc1
    in (aval + i, stc2) end
```

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

A dereferencing expression *e evaluates e to obtain an lvalue.

An array indexing acc[i] finds the address of array acc; this is the lvalue aval.

Then it evaluates e_i to obtain an integer i, and returns the lvalue aval + i.

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

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 : venv) (sto : sto) : venv * sto =
  case stmt of
  IF(e, stmc1, stmc2) =>
    let val (v, stc1) = eval e env sto
    in
      if v<>0 then
        (env, #2 (exec stmc1 env stc1))
      else
        (env, #2 (exec stmc2 env stc1))
    end
  While(e, body) =>
    let fun loop stc1 =
          let val (v, stc2) = eval e env stc1
              in
                if v<>0 then
                  loop (#2 (exec body env stc2))
                else
                  stc2
              end
        in (env, loop stc) end
    Expr e =>
      let val (v, stc1) = eval e env sto
      in (env, stc1) end
  Block stmts =>
    let fun loop [] = (env, sto) = (env, sto)
        | loop (s1::sr) (env, sto) = loop sr (stmcorderc s1 env sto)
    val (_, stc1) = loop stmts (env, sto)
    in (env, stc1) end
```

Notes on the micro-C interpreter

Micro-C arrays are more similar to B's semantics than C's.

Arrays: micro-C differs from real C.

In micro-C an array is a variable which holds the address of the first array element.

This was how B (the predecessor of C) represented array variables.

This is how C handles array-type parameters, but not how C handles array-type variables.