

**Programming Languages, F2002**  
**Lecture 7, Wednesday 20 March 2002**

- Micro-C arrays versus C arrays, and a correction
- Imperative programming in SML: `ref` values
- An abstract machine with program counter, evaluation stack, and a uniform store
- Compilation of micro-C for the abstract machine
- Critique of the generated code

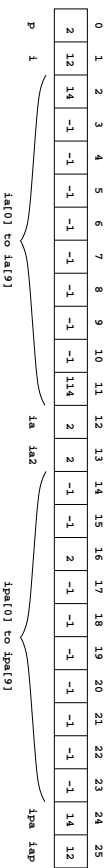
**Arrays: micro-C differs from real C**

We represent an array as a variable which holds the address of the first array element.

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

This is consistent with C's array-type parameters, but not with C's array-type variables.

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



**Imperative programming in SML**

Standard ML is a mostly functional language, not purely functional.

A value of type `t ref` is an *updatable reference* to a value of type `t`.

It is very similar to `t *`, or pointer to `t`, in C.

But SML references are safe (no General Protection Faults or NullPointerExceptions):

- An SML reference must point to a value, it cannot be `null`.
- There is no pointer arithmetic.

**Operations on SML references**

SML operation	Meaning	C analogue
<code>val r = ref v</code>	Create reference, initialized to <code>v</code>	<code>*r</code>
<code>!r</code>	Dereference; get value	<code>*r = u</code>
<code>r := u</code>	Update reference	

**Example use of references: generate new labels**

Some operations are very cumbersome without imperative features.

```
val nextlab = ref ^1;
fun newLabel () =
  (nextlab := 1 + !nextlab; "L" ^ Int.toString (!nextlab));
- newLabel ();
> val it = "L0" : string
- newLabel ();
> val it = "L1" : string
- newLabel ();
> val it = "L2" : string
```

Every call to `newLabel` returns a new label, because `nextLab` gets updated.

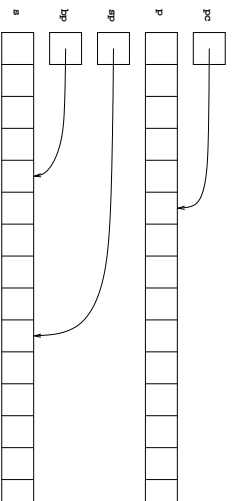
**Using `local - in - end` to encapsulate state**

```
local
  val nextlab = ref 0
in
  fun resetLabels () = nextlab := 0
  fun newLabel () =
    (nextlab := 1 + !nextlab; "L" ^ Int.toString (!nextlab))
end;
```

Now `nextLab` can only be operated on via `resetLabels` and `newLabel` (similar to a private static field).

### An abstract stack machine

Machine state: program, stack, registers



The state components

Name	Description	Use
pc	Program counter	Points to next instruction in $p[]$
$p[]$	Program store	Array of instructions
sp	Stack pointer	Points to current stack top in $s[]$
bp	Base pointer	Points to base of current stack frame in $s[]$
$s[]$	Data store, as a stack	Array of integers and addresses

### Stack machine instructions, part 1

Instruction	Stack before	Stack after	Effect
CST $i$	$s$	$i : s$	Push constant $i$
ADD	$i_2 : i_1 : s$	$(i_1 + i_2) : s$	Add
SUB	$i_2 : i_1 : s$	$(i_1 - i_2) : s$	Subtract
MUL	$i_2 : i_1 : s$	$(i_1 * i_2) : s$	Multiply
DIV	$i_2 : i_1 : s$	$(i_1 / i_2) : s$	Divide
MOD	$i_2 : i_1 : s$	$(i_1 \% i_2) : s$	Modulo
EQ	$i_2 : i_1 : s$	$(i_1 = i_2) : s$	Equality test (0 or 1)
LT	$i_2 : i_1 : s$	$(i_1 < i_2) : s$	Less-than test (0 or 1)
NOT	$v : s$	$!v : s$	Logical negation (0 or 1)
DUP	$v : s$	$v : v : s$	Duplicate
SWAP	$v_2 : v_1 : s$	$v_1 : v_2 : s$	Swap
LDI	$i : s$	$s[i] : s$	Load indirect
STI	$v : i : s$	$v : s$	Store indirect, set $s[i] := v$
GETBP	$s$	$bp : s$	Load base pointer $bp$
GETSP	$s$	$sp : s$	Load stack pointer $sp$
INCSF $m$	$s$	$v_m : \dots : v_1 : s$	Increment $sp$ when $m \geq 0$ ; $v_m : \dots : v_m$ arbitrary
INCSF $m$	$v_{-m} : \dots : v_1 : s$	$s$	Decrement $sp$ when $m < 0$

### Stack machine instructions, part 2

Instruction	Stack before	Stack after	Effect
GOTO $a$	$s$	$s$	Jump to $a$
IFZERO $a$	$v : s$	$s$	Jump to $a$ if $v = 0$
IFNZRO $a$	$v : s$	$s$	Jump to $a$ if $v \neq 0$
CALL $m a$	$v_m : \dots : v_1 : s$	$v_m : \dots : v_1 : bp : r : s$	Call function at $a$
TCALL $m n a$	$v_m : \dots : v_1 : u_n : \dots : u_1 : b : r : s$	$v_m : \dots : v_1 : b : r : s$	Tail-call function at $a$
RET $m$	$v : v_m : \dots : v_1 : b : r : s$	$v : s$	Return: $bp := b, pc := r$
PRINTI	$v : s$	$v : s$	Print $v$ as decimal integer
PRINTC	$v : s$	$v : s$	Print character $v$
STOP	$s$	$s$	Halt the machine

Some restrictions built-in to this machine:

- Input can come only from the command line; output goes to standard output.
- No support for indirect jumps and indirect calls; hence no function pointers.

### Stack machine implementation (file `imp/Machine.java`)

- The program  $p$  is an array of integers;  $pc$  is an index into that array.
- The store  $s$  is an array of integers;  $sp$  and  $bp$  are indexes into that array.
- The instruction interpreter is an infinite loop with a `switch` on the next instruction. Extract:
 

```

for ( ; ) {
    switch (p[pc++]) {
        case CST:
            s[sp+1] = p[pc++]; sp++; break;
        case ADD:
            s[sp-1] = s[sp-1] + s[sp]; sp--; break;
        case NOT:
            s[sp] = (s[sp] == 0 ? 1 : 0); break;
        case DUP:
            s[sp+1] = s[sp]; sp++; break;
        case SWAP:
            { int tmp = s[sp]; s[sp] = s[sp-1]; s[sp-1] = tmp; } break;
        case LDI:
            s[sp] = s[s[sp]]; break; // load indirect
        case STI:
            s[s[sp-1]] = s[sp]; s[sp-1] = s[sp]; sp--; break; // store indirect, keep value on top
        case GETSP:
            s[sp+1] = sp; sp++; break;
        case GOTO:
            pc = p[pc]; break;
        case IFZERO:
            pc = (s[sp-1] == 0 ? p[pc] : pc+1); break;
        ...
    }
}
            
```

### Local variable addressing via the base pointer bp

In the interpreter (`imp / c . sm1`) the run-time environment would map a variable to its address in the store.

This is impossible in the compiler, because a function may be called from other functions, and from itself.

Instead the compile-time environment maps a variable to its offset from the bottom of the current stack frame.

This offset can be computed at compile-time.

At run-time, the variable's address is the base pointer `bp` plus the variable's offset.

The address of a variable at offset 3 can be computed like this:

```
GETBP ; CSTI 3 ; ADD
```

In principle, one could address variables from the top of the current stack frame (using `sp`) instead.

This requires the compiler to keep track of the stack frame depth at any time.

It is possible, but looks a little complicated and is error-prone.

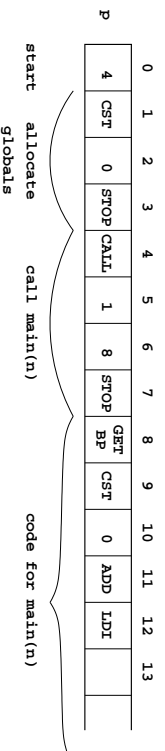
IT-C

### Machine code file layout

A machine code file has four components:

- The address *start* of the program entry code
- Initialization code to allocate global variables; must end with STOP.
- At address *start*: code to call the `main` function; usually followed by STOP.
- Code for each function in the program (including the `main` function).

Example (file `imp/ex14.c`):



IT-C

### Machine code program to print 1, 2, ...

```
2: STOP ; PRINTI ; 1 ; ADD ; GOTO 2
```

In raw machine code that is (file `imp/prog0`):

```
2 24 22 0 1 1 16 2
```

To run it, execute

```
java Machine prog0 117
```

To run it with tracing, execute

```
java MachineTrace prog0 117
```

### Machine code program to loop 20 million times

```
2: STOP ; 20000000 ; GOTO 9 ; 1 ; SUB ; DUP ; IFNZRO 6 ; STOP
```

In raw machine code (file `imp/prog1`):

```
2 24 0 20000000 16 9 0 1 2 9 18 6 24
```

IT-C

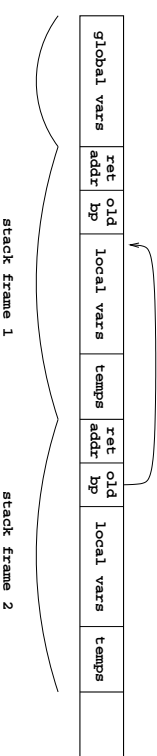
### Micro-C store layout convention: global variables and frame stack

The program's global variables are at the lower addresses of the store.

On top of the global variables, there is a stack of activation records or stack frames.

A *stack frame* or *activation record* corresponds to an active function call.

It contains the function's parameters and local variables, and temporary values of expressions being computed.



### Nested blocks, temporaries on the stack, and function calls in expressions

In micro-C, all code is within a function. So code executes in some function call's stack frame.

An inner block { ... } just allocates more variables on the function's stack frame.

A function call in an expression `117 + g(4)` will put `g`'s stack frame on top of the temporary value `117`.

IT-C



### Compiling if-else statements

A statement is compiled in a compile-time environment

```
fun cStmt stmt (env : venv) : Instr list =
  case stmt of
  | If(e, stmt1, stmt2) =>
      let val labe1se = newLabel()
          val labe1d = newLabel()
          val code1 = cStmt stmt1 env
          val code2 = cStmt stmt2 env
      in
        cExpr e env
        @ [IFZRO labe1se] @ code1 @ [GOTO labe1d]
        @ [Label labe1se] @ code2
        @ [Label labe1d]
      end
  | While(e, body) => ...
  | Expr e => cExpr e env @ [INCS "1"]
  | Block stmts => ...
  | Return _ => ...
```

The if-else statement `If(e1, stmt1, stmt2)` is compiled as

```
<ecode>
IFZRO L1:
<stmt1code>
GOTO L2:
L1: <stmt2code>
L2:
```

F-C

### Compiling blocks

```
fun cStmt stmt (env : venv) : Instr list =
  case stmt of
  ...
```

```
  | Block stmts =>
      let fun loop [] env = (#2 env, [])
          | loop (s1::sr) env =
              let val (env1, code1) = cStmtORDec s1 env
                  val (fdepthr, coder) = loop sr env1
                  in (fdepthr, code1 @ coder) end
              val (fdepthend, code) = loop stmts env
              in code @ [INCSP(#2 env - fdepthend)] end
          | ...
```

```
and cStmtORDec (Stmt stmt) env : venv * Instr list =
```

```
(env, cStmt stmt env)
| cStmtORDec (Dec (typ, x)) env =
  allocate Locvar (typ, x) env
```

A block { dec ... stmt ... dec ... } contains a list of declarations and statements.

A statement does not change the environment, but generates some instructions.

A declaration generates allocation code and extends the environment; its scope is the rest of the block.

F-C

### Compiling while-loops

```
fun cStmt stmt (env : venv) : Instr list =
  case stmt of
```

```
  ...
  | While(e, body) =>
      let val labe1begin = newLabel()
          val labe1test = newLabel()
          val codebody = cStmt body env
      in
        [GOTO labe1test, Label labe1begin] @ codebody
        @ [Label labe1test] @ cExpr e env @ [IFNZRO labe1begin]
        | ...
      end
  | ...
```

The while-loop statement `While(e, body)` is compiled as

```
GOTO L2:
L1: <bodycode>
L2: <ecode>
IFNZRO L1:
```

F-C

### Compiling return statements

A `return`; statement without an expression should execute a `RET (m-1)` to discard `m` local variables, yet leave some junk on the stack top, restore the old base pointer `bpj`, and return to the calling function.

A function call statement `F()`; will remove a value from the stack top, so `F()` must leave some junk value.

A `return e`; statement with an expression should evaluate `e`, then execute a `RET i` to leave `e`'s value on the stack top.

```
fun cStmt stmt (env : venv) : Instr list =
  case stmt of
  ...
  | Return NONE => [RET (#2 env - 1)]
  | Return (SOME e) => cExpr e env @ [RET (#2 env)]
```

F-C

### The code generated for a variable declaration

Compilation of a variable declaration

- extends the compile-time environment, so it maps the variable to its offset and type;
- generates code that will set aside stack space for the variable at run-time.

The generated code:

Declaration	Generated code
int i	CSTI 0
int *ip	CSTI 0
int ia[3]	INCSP 3; GETSP; CSTI(-2); SUB
int *ipa[3]	INCSP 3; GETSP; CSTI(-2); SUB
int (*iap)[3]	CSTI 0
int a[2][3]	(not permitted by micro-C)

IT-C

### Shortcomings of this compiler

- Tail-calls are not executed in constant space (example `imp/ex12.c`).
- Clumsy code is generated, especially for `if`- and `while`-conditions:

```
void main(int x) {
    if (x == 0) print 33; else print 44;
}
```

This produces the following code:

```
GETBP; CSTI 0; ADD; LDI; CSTI 0; EQ; IFZERO L2;
CSTI 33; PRINTI; INCSP ~1; GOTO L3;
L2: CSTI 44; PRINTI; INCSP ~1;
L3: INCSP 0; RET 0
```

Later we shall improve the compiler to produce this instead:

```
GETBP; LDI; IFNZRO L2;
CSTI 33; PRINTI; RET 1;
L2: CSTI 44; PRINTI; RET 1
```

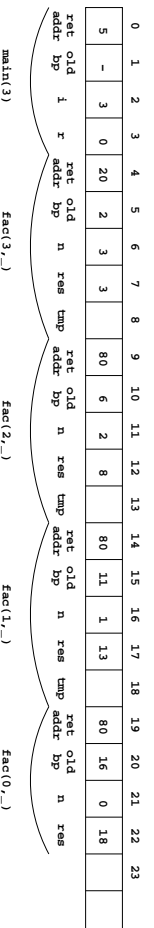
IT-C

### Recursive function calls revisited (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 as a stack of activation records:



IT-C