

Programming Languages, F2003
Lecture 7, Wednesday 19 March 2003

- 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

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).

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 arithmetics.

Operations on SML references

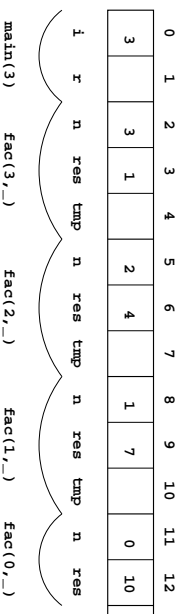
SML operation	Meaning	C analogue
<code>val r = ref v</code>	Create reference, initialized to <code>v</code>	<code>*r</code>
<code>!r</code>	Dereference: get value	<code>*r</code>
<code>r := u</code>	Update reference	<code>*r = u</code>

The micro-C interpreter executing recursive function calls (tmp/ex9.c)

```
void main(int i) {
  int r;
  fac(i, &r);
  print r;
}

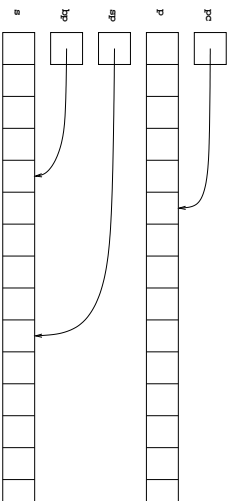
void fac(int n, int *res) {
  print &n;
  if (n == 0)
    *res = 1;
  else {
    int tmp;
    fac(n-1, &tmp);
    *res = tmp * n;
  }
}
```

The store becomes a stack of activation records:



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 stack machine 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

IT-C

Programming Languages, F2003

Page 7-5

Stack machine instructions, part 1

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

IT-C

Programming Languages, F2003

Page 7-6

Stack machine instructions, part 2

Instruction	Stack before	Stack after	Effect
GOTO a	s	s	Jump to a
IFZERO a	s, v	s	Jump to a if $v = 0$
IFNZRO a	s, v	s	Jump to a if $v \neq 0$
CALL m, a	s, v_1, \dots, v_m	$s, r, bp, v_1, \dots, v_m$	Call function at a
TCALL m, n, a	s, r, b, v_1, \dots, v_m	s, r, b, v_1, \dots, v_m	Tail-call function at a
RET m	$s, r, b, v_1, \dots, v_m, v$	s, v	Return: $bp := b, pc := r$
PRINTI	s, v	s, v	Print v as decimal integer
PRINTC	s, v	s, v	Print character v
LDARGS	s	s, i_1, \dots, i_n	Command line args
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.

IT-C

Programming Languages, F2003

Page 7-7

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;
    }
}

```

IT-C

Programming Languages, F2003

Page 7-8

Standard ML representation of the stack machine code (file `imp/Machine.sml`)

```
datatype label =
  Lab of string
datatype instr =
  | Label of label
  | CSTI of int
  | ...
  | GOTO of label
  | ...
(* symbolic label; pseudo-instruc. *)
(* constant
(* go to label
*)
```

There is a function to convert a symbolic instr list into an absolute int list:

```
val code2ints = fn : instr list -> int list
```

It traverses the instruction list twice:

- In pass 1 it builds an environment mapping each label to the code address it labels:

```
case instr of
  Label lab => defLabel lab
  | ...
```

- In pass 2 it emits converts instructions and labels to integers.

```
case instr of
  ...
  | GOTO lab => (out CODEGOTO; outLabel lab)
```

Compiling expressions with arithmetic and logical operators

```
and cExpr (e : expr) (env : env) : instr list =
  case e of
    ...
  | Prim2(ope, e1, e2) =>
      cExpr e1 env
    @ cExpr e2 env
    @ (case ope of
```

```
  "+" => [MUL]
  | "-" => [ADD]
  | "*" => [SUB]
  | "/" => [DIV]
  | "%" => [MOD]
  | "=" => [EQ]
  | "!=" => [EQ, NOT]
  | "<" => [LT]
  | ">" => [LT, NOT]
  | ">=" => [SWAP, LT]
  | "<=" => [SWAP, LT, NOT]
  | _ => raise Fail "unknown primitive 2")
```

Using logical negation NOT and swap SWAP we need only == and < in the machine.

Thus `11 <= 22` is compiled to `[CSTI 11, CSTI 22, SWAP, LT, NOT]`.

Compiling micro-C expressions to stack machine code (file `imp/comp.sml`)

```
and cExpr (e : expr) (env : env) : instr list =
  case e of
    Access acc => cAccess acc env @ [LDI]
  | Assign(acc, e) => cAccess acc env @ cExpr e env @ [STI]
  | Cst (CSTI i) => [CSTI i]
  | Cst CstN => [CSTI 0]
  | Addr acc => cAccess acc env
  | Prim1(ope, e1) =>
      cExpr e1 env
    @ (case ope of
        "i" => [NOT]
        | "printi" => [PRINTI]
        | "printc" => [PRINTC]
        | _ => raise Fail "unknown primitive 1")
  | Prim2(ope, e1, e2) => ...
  | Andalso(e1, e2) => ...
  | Orelse(e1, e2) => ...
  | Call(f, es) => callFun f es env
```

Thus `!false` which is `Prim1("i", Cst(CSTI 0))` is compiled to:

```
[CSTI 0, NOT]
```

Net effect principle:

If expression `e` compiles to code `instrs`, then execution of `instrs` will leave the value of `e` on the stack top.

Compiling an access expression to stack machine code

The compile-time environment maps a variable name to a variable description and a micro-C type:

```
type env = (string, var * typ) env * int
```

The compile-time environment also keeps track of the next available offset.

A variable is described as a global variable or a local variable (parameter or variable declared in function):

```
datatype var =
  GVar of int (* absolute address in stack *)
  | LVar of int (* offset in current stack frame *)
type env = (string, var * typ) env
```

Compiling an access expression (variable `x` or pointer dereferencing `*p` or array indexing `acc[i]`):

```
and cAccess (AccVar x) env =
  (case lookup (#1 env) x of
    (GVar addr, _) => [CSTI addr]
  | (LVar addr, _) => [GETBP, CSTI addr, ADD])
  @ cExpr e env
  | cAccess (AccDeref e) env =
    cExpr e env
  | cAccess (AccIndex(acc, idx)) env =
    cAccess acc env @ [LDI] @ cExpr idx env @ [ADD]
```

Idea: The code generated for an access expression leaves an address on the stack top.

What is `xs[i]` compiled to when `xs` and `i` are local variables at offsets 2 and 5?

Compiling AndAlso to stack machine code

```
and cExpr (e : expr) (env : venv) : instr list =
  case e of
  ...
  | AndAlso(e1, e2) =>
    let val labend = newLabel()
    val labfalse = newLabel()
    in
      cExpr e1 env
      @ [IFZERO labfalse]
      @ cExpr e2 env
      @ [GOTO labend, Label labfalse, CSRR 0, Label labend]
    end
  end
```

Thus the code for e1 && e2 is

```
<elcode>
IFZERO L1:
<e2code>
GOTO L2:
L1: 0
L2:
```

What code is generated for L1 < 22 && 33 < 44?

The code for OrElse is dual to that for AndAlso. Use IFNZRO and I instead of IFZERO and 0.

Compiling while-loops

```
fun cStm stmt (env : venv) : instr list =
  case stmt of
  ...
  | While(e, body) =>
    let val labbegin = newLabel()
    val labtest = newLabel()
    in
      [GOTO labtest, Label labbegin] @ cStm body env
      @ [Label labtest] @ cExpr e env @ [IFNZRO labbegin]
    end
  | ...
  end
```

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

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

Compiling if-else statements

A statement is compiled in a compile-time environment. The result is a list of instructions:

```
fun cStm stmt (env : venv) : instr list =
  case stmt of
  If(e, stmt1, stmt2) =>
    let val labelse = newLabel()
    val labend = newLabel()
    in
      cExpr e env @ [IFZERO labelse]
      @ cStm stmt1 env @ [GOTO labend]
      @ [Label labelse] @ cStm stmt2 env
      @ [Label labend]
    end
  | While(e, body) => ...
  | Expr e => cExpr e env @ [INCSR "1"]
  | Block stmts => ...
  | Return _ => ...
  end
```

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

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

Compiling blocks

```
fun cStm stmt (env : venv) : instr list =
  case stmt of
  ...
  | Block stmts =>
    let fun loop [] env = (#2 env, [])
        | loop (s1::sr) env =
            let val (env1, code1) = cStmForDec s1 env
                val (fdepthr, coder) = loop sr env1
            in (fdepthr, code1 @ coder) end
        val (fdepthnd, code) = loop stmts env
    in code @ [INCSR("#2 env - fdepthnd)] end
    | ...
    end
  and cStmForDec (Stm stmt) env : venv * instr list =
    (env, cStm stmt env)
  | cStmForDec (Dec (typ, x)) env =
    allocate Locvar (typ, x) env
  end
```

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.

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 *iap[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:

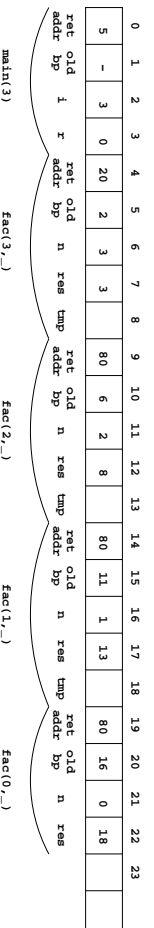
```
GETBP; LDI; IFNZRO L2;  
CSTI 33; PRINTI; RET 1;  
L2: CSTI 44; PRINTI; RET 1
```

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