CS:3820
Programming Language Concepts

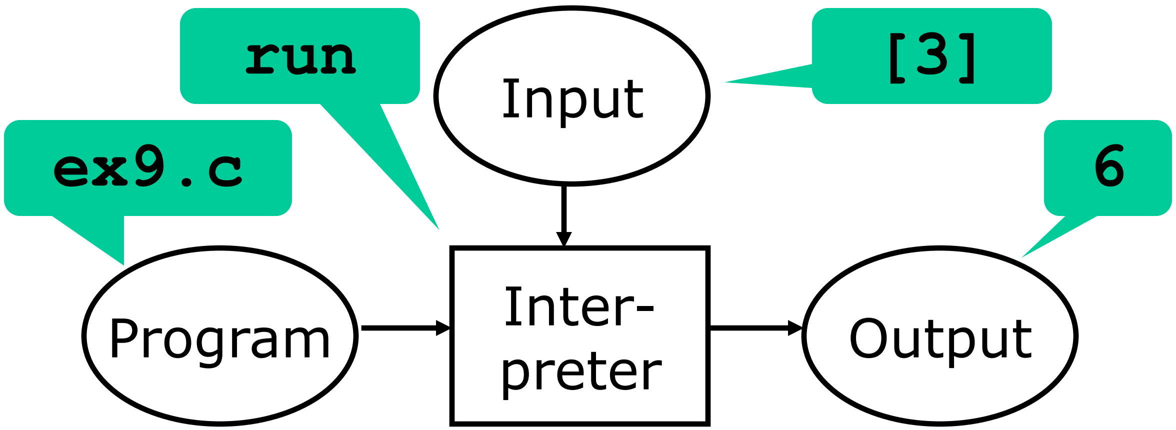
A stack machine for micro-C: Compiling micro-C to stack machine code

Main Topics

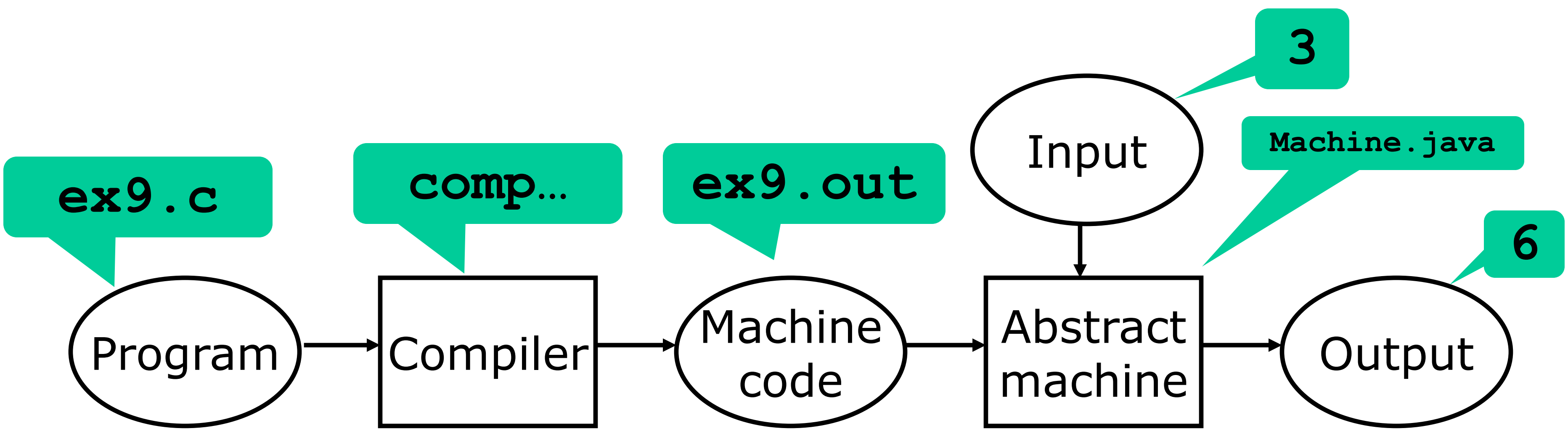
- Stack machine, target for micro-C compiler
  - Stack machine state
  - Instruction set
  - Implementations in Java and C
- Compiling micro-C to stack machine code

Interpretation and compilation

- Interpretation = one-stage execution/evaluation:



- Compilation = two-stage execution/evaluation:



Stack machine state transitions

Instruction	Stack before		Stack after	Effect
0 CSTI i	s	$\Rightarrow$	s, i	Push constant i
1 ADD	s, i_1, i_2	$\Rightarrow$	$s, (i_1 + i_2)$	Add
2 SUB	s, i_1, i_2	$\Rightarrow$	$s, (i_1 - i_2)$	Subtract
3 MUL	s, i_1, i_2	$\Rightarrow$	$s, (i_1 * i_2)$	Multiply
4 DIV	s, i_1, i_2	$\Rightarrow$	$s, (i_1 / i_2)$	Divide
5 MOD	s, i_1, i_2	$\Rightarrow$	$s, (i_1 \% i_2)$	Modulo
6 EQ	s, i_1, i_2	$\Rightarrow$	$s, (i_1 = i_2)$	Equality (0 or 1)
7 LT	s, i_1, i_2	$\Rightarrow$	$s, (i_1 < i_2)$	Less-than (0 or 1)
8 NOT	s, v	$\Rightarrow$	$s, !v$	Negation (0 or 1)
9 DUP	s, v	$\Rightarrow$	s, v, v	Duplicate
10 SWAP	s, v_1, v_2	$\Rightarrow$	s, v_2, v_1	Swap
11 LDI	s, i	$\Rightarrow$	$s, s[i]$	Load indirect
12 STI	s, i, v	$\Rightarrow$	s, v	Store indirect $s[i] = v$
13 GETBP	s	$\Rightarrow$	s, bp	Load base ptr bp
14 GETSP	s	$\Rightarrow$	s, sp	Load stack ptr sp
15 INCSP m	s	$\Rightarrow$	$s, v_1, .., v_m$	Grow stack ($m \geq 0$)
15 INCSP m	$s, v_1, .., v_{-m}$	$\Rightarrow$	s	Shrink stack ($m < 0$)
16 GOTO a	s	$\Rightarrow$	s	Jump to a
17 IFZERO a	s, v	$\Rightarrow$	s	Jump to a if $v = 0$
18 IFNZRO a	s, v	$\Rightarrow$	s	Jump to a if $v \neq 0$
19 CALL m a	$s, v_1, .., v_m$	$\Rightarrow$	$s, r, bp, v_1, .., v_m$	Call function at a
20 TCALL m n a	$s, r, b, u_1, .., u_n, v_1, .., v_m$	$\Rightarrow$	$s, r, b, v_1, .., v_m$	Tailcall function at a
21 RET m	$s, r, b, v_1, .., v_m, v$	$\Rightarrow$	s, v	Return $bp = b$, $pc = r$
22 PRINTI	s, v	$\Rightarrow$	s, v	Print integer v
23 PRINTC	s, v	$\Rightarrow$	s, v	Print character v
24 LDARGS	s	$\Rightarrow$	$s, i_1, .., i_n$	Command line args
25 STOP	s	$\Rightarrow$	$\_$	Halt the machine

Example stack machine program

- A simple program, file prog1:

```
0 20000000 16 7 0 1 2 9 18 4 25
```

Numeric code

```
0 20000000
16 7
0 1
2
9
18 4
25
```

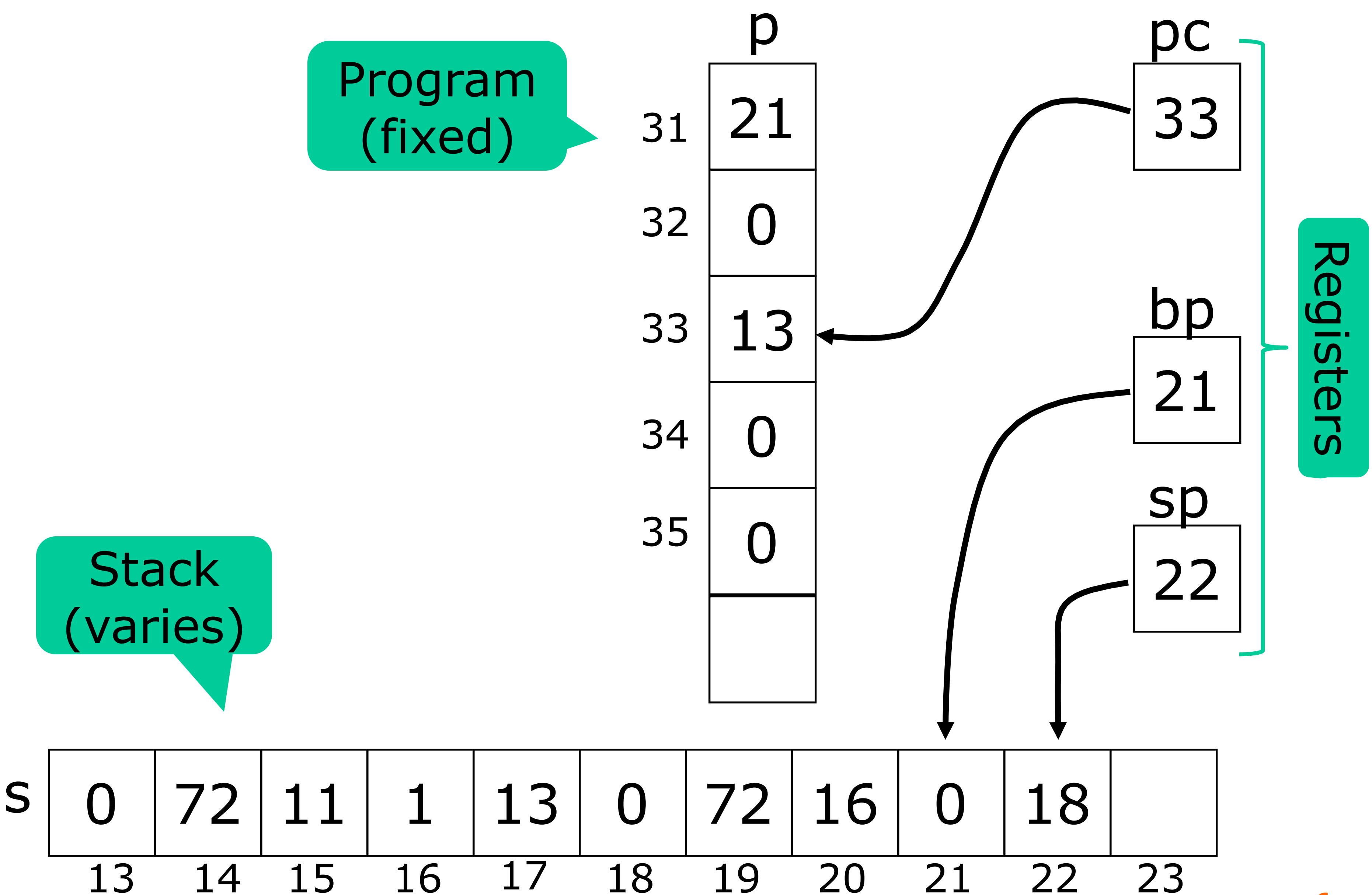
```
 0: CSTI 20000000
 2: GOTO 7
 4: CSTI 1
 6: SUB
 7: DUP
 8: IFNZRO 4
10: STOP
```

Symbolic code

- Running the code in file prog1:

```
C:>java Machine prog1
Ran 0.641 seconds
```

Machine state: p, pc, s, sp, bp



Stack machine for micro-C

- Runtime state:
  - Program **p**, holds the instructions
  - Program counter **pc**, points to next instruction
  - Stack **s**, holds variables and intermediate results
  - Stack pointer **sp**, points to top of stack
  - Base pointer **bp**, points to first local variable in top stack frame

- Structure of the stack
  - Bottom: Global variables
  - One stack frame for each active method

Implementations of the micro-C abstract machine

- File `Machine.java`: An implementation of the abstract machine as a Java program
- File `machine.c`: An implementation of the abstract machine as a C program
- File `Machine.fs`: A definition of the instruction set for use in the compiler `Comp.fs`
  - The instruction numbers in `Machine.fs` agree with `Machine.java` and `machine.c`

Stack machine instruction execution

Java or C or C#

```
for (;;) {
  switch (p[pc++]) {
  case CSTI:
    s[sp+1] = p[pc++]; sp++; break;
  case ADD:
    s[sp-1] = s[sp-1] + s[sp]; sp--; break;
  case EQ:
    s[sp-1] = (s[sp-1] == s[sp] ? 1 : 0); sp--; break;
  case DUP:
    s[sp+1] = s[sp]; sp++; break;
  case LDI:
    s[sp] = s[s[sp]]; break;
  case GOTO:
    pc = p[pc]; break;
  case IFZERO:
    pc = (s[sp--] == 0 ? p[pc] : pc+1); break;
  case ...
  case STOP:
    return sp;
  ...
} }
```

Structure of the micro-C stack

- Computing factorial with MicroC/ex9.c

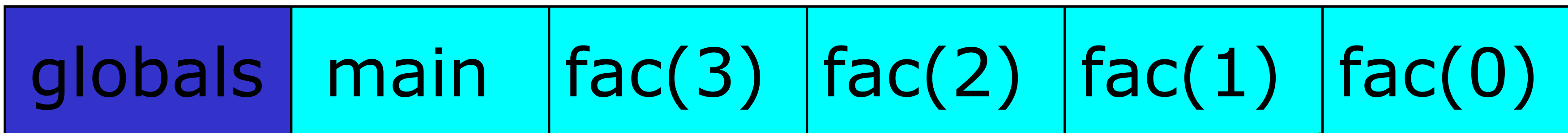
```
void main(int i) {
  int r;
  fac(i, &r);
  print r;
}

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

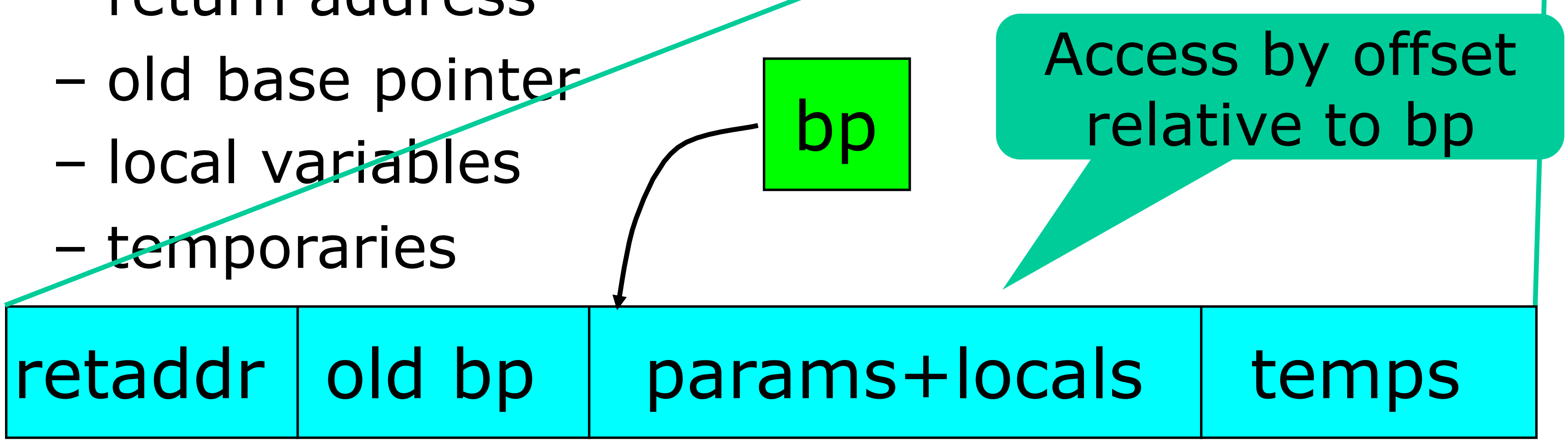
- **n** is input parameter
- **res** is output parameter, a pointer to where to put the result
- **tmp** holds the result of the recursive call
- **&tmp** gets the pointer to **tmp**

Runtime storage: the stack

- The store is an indexable stack
  - bottom: global variables at fixed addresses
  - followed by activation records



- An *activation record* is an executing function
  - return address
  - old base pointer
  - local variables
  - temporaries



Compiling micro-C

- Overall structure of a micro-C program:
  - Global variable declarations **`int x; int y;`**
  - Global function declarations **`void main(…) {…}`**

- Overall structure of the generated code:
  - Code to allocate all global variables
  - Code to load arguments, call **`main`**, and stop
  - Code for each function, including **`main`**

- Structure of code for a function:
  - Code for the function's body statement
  - Code (RET) to return from the function

Observations

- At run time, a local variable’s place within a stack frame is always the same
- This *offset* can be computed at compile time
- The compile time environment in the micro-C compiler maps a local variable to an offset

- The run time environment is the stack of activation records in the abstract machine
- At run time, the base pointer BP points at the bottom of the current activation record
- So a local variable’s address is BP+offset

Variable offsets

- Example MicroC/ex9.c again:

```
              0
void main(int i) {
  int r;  1
  fac(i, &r);
  print r;
}

             0        1
void fac(int n, int *res) {
  if (n == 0)
    *res = 1;
  else {  2
    int tmp;
    fac(n-1, &tmp);
    *res = tmp * n;
  }
}
```

Compile-time environments

- varEnv = variable environment
  - global variable → global address in stack
  - local variable → offset in activation record

- funEnv = function environment
  - function name → (label,
return type,
parameter types)

Main micro-C compiler functions

- **`cStmt stmt varEnv funEnv : instr list`**
  - Compiles **`stmt`** to code that performs the statement's actions
- **`cExpr expr varEnv funEnv : instr list`**
  - Compiles **`expr`** to code that leaves the expr's rvalue on the stack top
- **`cAccess expr varEnv funEnv : instr list`**
  - Compiles **`expr`** to code that leaves the expr's lvalue on the stack top

Main micro-C compiler functions

- **`cProgram topdecs : instr list`**
  - Builds global varEnv and global funEnv
  - Generates code
    - for global variables
    - to call function **`main`**
    - for all functions, including **`main`**

Micro-C abstract syntax

```
type typ =
  | TypI                                   (* Type int                    *)
  | TypC                                   (* Type char                   *)
  | TypA of typ * int option               (* Array type                  *)
  | TypP of typ                            (* Pointer type                *)
and expr =
  | Access of access                       (* x    or  *p    or  a[e]     *)
  | Assign of access * expr                (* x=e  or  *p=e  or  a[e]=e   *)
  | Addr of access                         (* &x   or  &*p   or  &a[e]    *)
  | CstI of int                            (* Constant                    *)
  | Prim1 of string * expr                 (* Unary primitive operator    *)
  | Prim2 of string * expr * expr          (* Binary primitive operator   *)
  | Andalso of expr * expr                 (* Sequential and              *)
  | Orelse of expr * expr                  (* Sequential or               *)
  | Call of string * expr list             (* Function call f(...)        *)
and access =
  | AccVar of string                       (* Variable access        x    *)
  | AccDeref of expr                       (* Pointer dereferencing  *p   *)
  | AccIndex of access * expr              (* Array indexing         a[e] *)
and stmt =
  | If of expr * stmt * stmt               (* Conditional                 *)
  | While of expr * stmt                   (* While loop                  *)
  | Expr of expr                           (* Expression statement   e;   *)
  | Return of expr option                  (* Return from method          *)
  | Block of stmtordec list                (* Block: grouping and scope   *)
and stmtordec =
  | Dec of typ * string                    (* Local variable declaration  *)
  | Stmt of stmt                           (* A statement                 *)
and topdec =
  | Fundec of typ option * string * (typ * string) list * stmt
  | Vardec of typ * string
and program =
  | Prog of topdec list
```

Types

Expressions: rvalue (expr), lvalue (access)

Statements

Declarations

Compiling arithmetic expressions and assignment

- `<e1>` means: the result of compiling **`e1`**

<u>Compile **`17`** as rvalue:</u>

```
CSTI 17
```

<u>Compile **`e1 + e2`** as rvalue:</u>

```
<e1> as rvalue
<e2> as rvalue
ADD
```

<u>Compile **`e1 = e2`** as rvalue:</u>

```
<e1> as lvalue
<e2> as rvalue
STI
```

`cExpr`

Micro-C compiler fragment

```
and cExpr e varEnv funEnv : instr list =
    match e with
    | Access acc      -> cAccess acc varEnv funEnv
                         @ [LDI]
    | Assign(acc, e) -> cAccess acc varEnv funEnv
                         @ cExpr e varEnv funEnv
                         @ [STI]
    | CstI i          -> [CSTI i]
    | Addr acc        -> cAccess acc varEnv funEnv
    | Prim2(ope, e1, e2) ->
      cExpr e1 varEnv funEnv
      @ cExpr e2 varEnv funEnv
      @ (match ope with
         | "*"   -> [MUL]
         | "+"   -> [ADD]
         | "<"   -> [LT]
         | ...)
    | ...
```

Compiling comparisons

Compile **e1 < e2** as rvalue:

```
<e1> as rvalue
<e2> as rvalue
LT
```

cExpr

- Q: How compile >=, >, <= when we have only LT?
- A: Use NOT and SWAP (how?)

Compiling lvalues and rvalues

Compile `x` as lvalue:

```
GETBP
CSTI <xoffset>
ADD
```

Compile `e1[e2]` as lvalue:

```
<e1> as rvalue
<e2> as rvalue
ADD
```

Compile `*e` as lvalue:

```
<e> as rvalue
```

cAccess

Compile `e` as rvalue:

```
<e> as lvalue
LDI
```

Compile `&e` as rvalue:

```
<e> as lvalue
```

cExpr

Compiling blocks

- To compile a block **`{ s1 s2 … sn }`**
  - Make new scope in varEnv
  - Compile **`<s1> <s2> … <sn>`**
  - Drop new scope from varEnv
  - Generate code (INCSP (−m)) to forget m locals

Compiling declarations

- To compile int declaration **`int x`**
  - Generate code to increment stack pointer by 1
- To compile array declaration **`int a[5]`**
  - Generate code to allocate 5 stack places, that is, increment stack pointer by 5
  - Generate code to compute address of the first of those locations, and put it on the stack

Statement compilation schemes

Compile `if (e) s1 else s2:`

```
        <e> as rvalue
        IFZERO L1
        <s1>
        GOTO L2
    L1: <s2>
    L2:
```

Compile `while (e) s:`

```
    GOTO L2
L1: <s>
L2: <e> as rvalue
    IFNZRO L1
```

Compile `e; :`

```
<e> as rvalue
INCSP -1
```

cStmt

Micro-C compiler fragment

```
let rec cStmt stmt varEnv funEnv : instr list =
    match stmt with
    | If(e, stmt1, stmt2) ->
      let labelse = newLabel()
      let labend  = newLabel()
      in cExpr e varEnv funEnv @ [IFZERO labelse]
         @ cStmt stmt1 varEnv funEnv @ [GOTO labend]
         @ [Label labelse] @ cStmt stmt2 varEnv funEnv
         @ [Label labend]
    | While(e, body) ->
      let labbegin = newLabel()
      let labtest  = newLabel()
      in [GOTO labtest; Label labbegin]
         @ cStmt body varEnv funEnv
         @ [Label labtest] @ cExpr e varEnv funEnv
         @ [IFNZRO labbegin]
    | Expr e -> cExpr e varEnv funEnv @ [INCSP -1]
    | ...
```

Exercise

- What code should be generated for a **`do-while`** block:

```
do
   stmt
while (e) ;
```

- What code should be generated for a **`for`** statement:

```
for (e1; e2; e3)
   stmt
```

Micro-C Example ex9.c

```
// return a result via a pointer argument
void main(int i) {
  int r;
  fac(i, &r);
  print r;
}

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

The code generated for ex9.c

init

```
 0 LDARGS
 1 CALL 1 L1
 4 STOP
 5 L1:
```

main

```
 5 CSTI 0
 7 GETBP
 8 CSTI 0
10 ADD
11 LDI
12 GETBP
13 CSTI 1
15 ADD
16 CALL 2 L2
19 INCSP -1
21 GETBP
22 CSTI 1
24 ADD
25 LDI
26 PRINTI
27 INCSP -1
29 INCSP -1
21 RET 0
33 L2:
```

fac

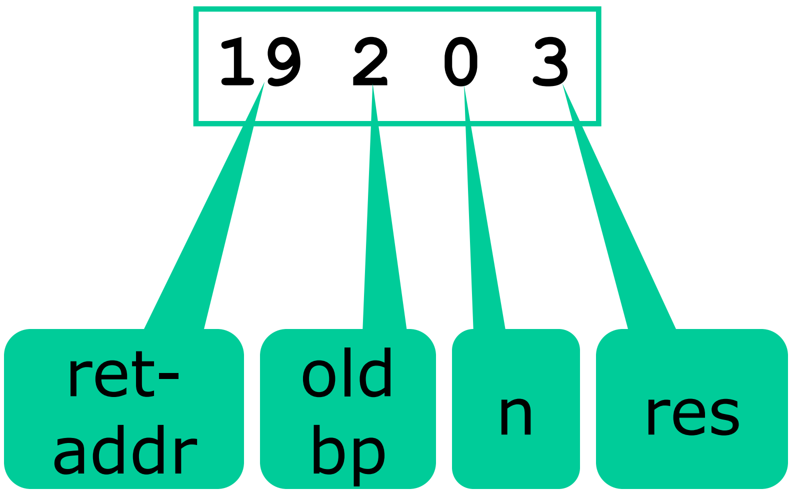
```
33 GETBP
34 CSTI 0
36 ADD
37 LDI
38 CSTI 0
40 EQ
41 IFZERO L3
43 GETBP
44 CSTI 1
46 ADD
47 LDI
48 CSTI 1
50 STI
51 INCSP -1
53 GOTO L4
55 L3:
55 CSTI 0
57 GETBP
58 CSTI 0
60 ADD
61 LDI
62 CSTI 1
64 SUB
65 GETBP
66 CSTI 2
68 ADD
69 CALL 2 L2
72 INCSP -1
74 GETBP
75 CSTI 1
77 ADD
78 LDI
79 GETBP
80 CSTI 2
82 ADD
83 LDI
84 GETBP
85 CSTI 0
87 ADD
88 LDI
89 MUL
90 STI
91 INCSP -1
93 INCSP -1
95 L4:
95 INCSP 0
97 RET 1
```

The code generated for ex9.c

```
 0 LDARGS
 1 CALL 1 L1
 4 STOP
 5 L1:
 5 INCSP 1
 7 GETBP
 8 CSTI 0
10 ADD
11 LDI
12 GETBP
13 CSTI 1
15 ADD
16 CALL 2 L2
19 INCSP -1
21 GETBP
22 CSTI 1
24 ADD
25 LDI
26 PRINTI
27 INCSP -1
29 INCSP -1
31 RET 0
33 L2:
33 GETBP
34 CSTI 0
36 ADD
37 LDI
38 CSTI 0
40 EQ
41 IFZERO L3
43 GETBP
44 CSTI 1
46 ADD
47 LDI
48 CSTI 1
50 STI
51 INCSP -1
53 GOTO L4
55 L3:
55 INCSP 1
57 GETBP
58 CSTI 0
60 ADD
61 LDI
62 CSTI 1
64 SUB
65 GETBP
66 CSTI 2
68 ADD
69 CALL 2 L2
72 INCSP -1
74 GETBP
75 CSTI 1
77 ADD
78 LDI
79 GETBP
80 CSTI 2
82 ADD
83 LDI
84 GETBP
85 CSTI 0
87 ADD
88 LDI
89 MUL
90 STI
91 INCSP -1
93 INCSP -1
95 L4:
95 INCSP 0
97 RET 1
```

Running ex9.c on 0: The stack of frames

- Example ex9.c: computing fac(0)
- Stack frame for fac(0):



- What stack frame?

```
4 -999 0 0
```

```
[ ]{0: LDARGS}
[ 0 ]{1: CALL 1 5}
[ 4 -999 0 ]{5: CSTI 0}
[ 4 -999 0 0 ]{7: GETBP}
[ 4 -999 0 0 2 ]{8: CSTI 0}
[ 4 -999 0 0 2 0 ]{10: ADD}
[ 4 -999 0 0 2 ]{11: LDI}
[ 4 -999 0 0 0 ]{12: GETBP}
[ 4 -999 0 0 0 2 ]{13: CSTI 1}
[ 4 -999 0 0 0 2 1 ]{15: ADD}
[ 4 -999 0 0 0 3 ]{16: CALL 2 33}
[ 4 -999 0 0 19 2 0 3 ]{33: GETBP}
[ 4 -999 0 0 19 2 0 3 6 ]{34: CSTI 0}
[ 4 -999 0 0 19 2 0 3 6 0 ]{36: ADD}
[ 4 -999 0 0 19 2 0 3 6 ]{37: LDI}
[ 4 -999 0 0 19 2 0 3 0 ]{38: CSTI 0}
[ 4 -999 0 0 19 2 0 3 0 0 ]{40: EQ}
[ 4 -999 0 0 19 2 0 3 1 ]{41: IFZERO 55}
[ 4 -999 0 0 19 2 0 3 ]{43: GETBP}
[ 4 -999 0 0 19 2 0 3 6 ]{44: CSTI 1}
[ 4 -999 0 0 19 2 0 3 6 1 ]{46: ADD}
[ 4 -999 0 0 19 2 0 3 7 ]{47: LDI}
[ 4 -999 0 0 19 2 0 3 3 ]{48: CSTI 1}
[ 4 -999 0 0 19 2 0 3 3 1 ]{50: STI}
[ 4 -999 0 1 19 2 0 3 1 ]{51: INCSP -1}
[ 4 -999 0 1 19 2 0 3 ]{53: GOTO 95}
[ 4 -999 0 1 19 2 0 3 ]{95: INCSP 0}
[ 4 -999 0 1 19 2 0 3 ]{97: RET 1}
[ 4 -999 0 1 3 ]{19: INCSP -1}
[ 4 -999 0 1 ]{21: GETBP}
[ 4 -999 0 1 2 ]{22: CSTI 1}
[ 4 -999 0 1 2 1 ]{24: ADD}
[ 4 -999 0 1 3 ]{25: LDI}
[ 4 -999 0 1 1 ]{26: PRINTI}
1 [ 4 -999 0 1 1 ]{27: INCSP -1}
[ 4 -999 0 1 ]{29: INCSP -1}
[ 4 -999 0 ]{31: RET 0}
```

Highlights from computing fac(3)

```
[ ]{0: LDARGS}
[ 3 ]{1: CALL 1 5}
[ 4 -999 3 ]{5: CSTI 0}
[ 4 -999 3 0 ]{7: GETBP}
...
[ 4 -999 3 0 3 3 ]{16: CALL 2 33}
[ 4 -999 3 0 19 2 3 3 ]{33: GETBP}
...
[ 4 -999 3 0 19 2 3 3 0 2 8 ]{69: CALL 2 33}
[ 4 -999 3 0 19 2 3 3 0 72 6 2 8 ]{33: GETBP}
...
[ 4 -999 3 0 19 2 3 3 0 72 6 2 8 0 1 13 ]{69: CALL 2 33}
[ 4 -999 3 0 19 2 3 3 0 72 6 2 8 0 72 11 1 13 ]{33: GETBP}
...
[ 4 -999 3 0 19 2 3 3 0 72 6 2 8 0 72 11 1 13 0 0 18 ]{69: CALL 2 33}
[ 4 -999 3 0 19 2 3 3 0 72 6 2 8 0 72 11 1 13 0 72 16 0 18 ]{33: GETBP}
...
[ 4 -999 3 0 19 2 3 3 0 72 6 2 8 0 72 11 1 13 1 72 16 0 18 ]{97: RET 1}
[ 4 -999 3 0 19 2 3 3 0 72 6 2 8 0 72 11 1 13 1 18 ]{72: INCSP -1}
...
[ 4 -999 3 0 19 2 3 3 0 72 6 2 8 1 72 11 1 13 ]{97: RET 1}
[ 4 -999 3 0 19 2 3 3 0 72 6 2 8 1 13 ]{72: INCSP -1}
...
[ 4 -999 3 0 19 2 3 3 2 72 6 2 8 ]{97: RET 1}
[ 4 -999 3 0 19 2 3 3 2 8 ]{72: INCSP -1}
...
[ 4 -999 3 6 19 2 3 3 ]{97: RET 1}
...
[ 4 -999 3 6 3 ]{25: LDI}
[ 4 -999 3 6 6 ]{26: PRINTI}
6 [ 4 -999 3 6 6 ]{27: INCSP -1}
[ 4 -999 3 6 ]{29: INCSP -1}
[ 4 -999 3 ]{31: RET 0}
[ 3 ]{4: STOP}
```

ret-addr
old bp
n
res

Compiler shortcomings

- The compiler often generates inefficient code

```
GETBP
CSTI 0
ADD
LDI
```

could be

```
GETBP
LDI
```

```
INCSP -1
INCSP -1
```

could be

```
INCSP -2
```

- The compiler itself is inefficient, using (`@`) a lot:

```
| If(e, stmt1, stmt2) ->
  let labelse = newLabel()
  let labend  = newLabel()
  in cExpr e varEnv funEnv @ [IFZERO labelse]
     @ cStmt stmt1 varEnv funEnv @ [GOTO labend]
     @ [Label labelse] @ cStmt stmt2 varEnv funEnv
     @ [Label labend]
```

- Tail calls are not executed in constant space
- We can fix these problems with an optimizing compiler

Exercise

- Adding a switch-statement to micro-C:
  - each case has an int constant and a block
  - implicit **break**, no fall-through; no explicit **break** or **default**

```
switch (month) {
  case 2:
    { days = 28; if (y%4==0) days = 29; }
  case 3:
    { days = 31; }
  case 1:
    { days = 31; }
}
```

- May be compiled as a sequence of tests
- The abstract syntax may be as simple as this:

```
Switch of expr * (int * stmt) list
```