CS406: Compilers

Spring 2021

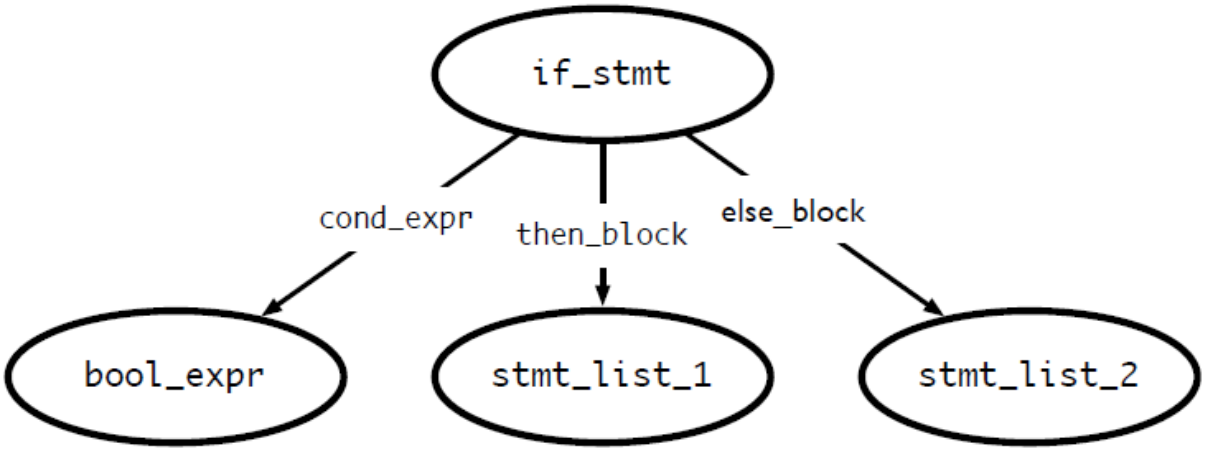
Week 9: IR code for if- statement, loops, switch, functions

If statements

```
if <bool_expr_1>
  <stmt_list_1>
else
  <stmt_list_2>
endif
```


Slide courtesy: Milind Kulkarni

If statements



Slide courtesy: Milind Kulkarni

Code-generation – if-statement

Program text	3AC
`INT a, b;`	

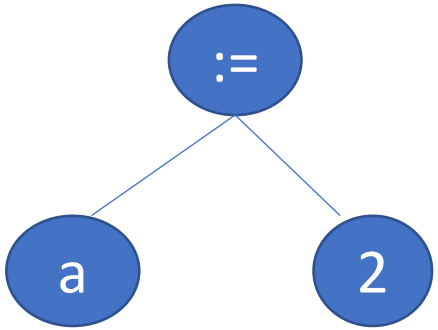
Code-generation – if-statement

Program text	3AC	
`INT a, b;`		Make entries in the symbol table

Code-generation – if-statement

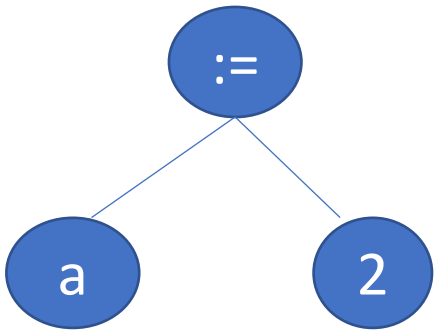
Program text	3AC
`INT a, b;` `a := 2;`	

Code-generation – if-statement

Program text	3AC
`INT a, b;` `a := 2;`	

1. “a” is left-child, type=l-val. No code generated. *Return an object containing identifier details after verifying that “a” is present in the symbol table.*

Code-generation – if-statement

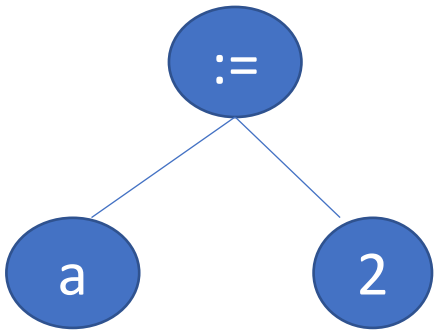
Program text	3AC
`INT a, b;` `a := 2;`	

1. “a” is left-child, type=l-val. No code generated. Pass up the identifier.
2. “2” is right-child, type=const. No code generated.

Code-generation – if-statement

Program text **3AC**

```
INT a, b;
a := 2;
```

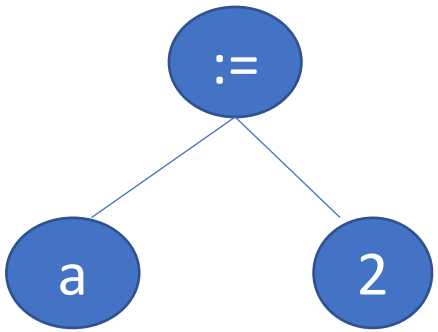


1. “a” is left-child, type=l-val. No code generated. Pass up the identifier.
2. “2” is right-child, type=const. No code generated.
3. Create a temporary T1 to store the result of the expression

Code-generation – if-statement

Program text **3AC**

```
INT a, b;
a := 2;
```



1. "a" is left-child, type=l-val. No code generated. Pass up the identifier.
2. "2" is right-child, type=const. No code generated.
3. Create a temporary T1 to store the result of the expression
   - Current node stores the op ':='. A call to `process_op` stores the RHS data in LHS

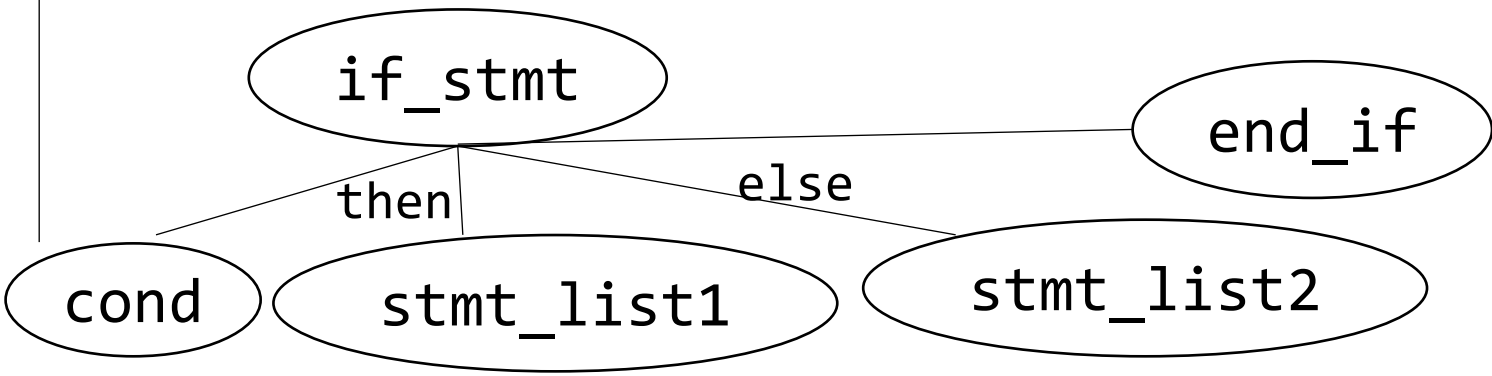
Code-generation – if-statement

Program text	3AC
`INT a, b;`	`STOREI 2 T1`
`a := 2;`	`STOREI T1 a`

Code-generation – if-statement

Program text	3AC

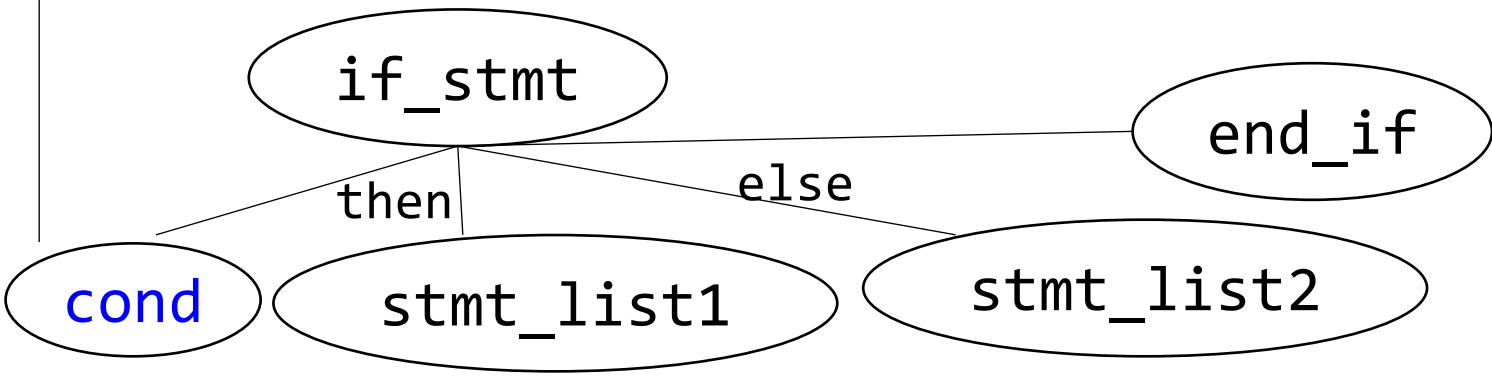
INT a, b;
a := 2;
IF (a = 1)
  b := 1;
ELSE
  b := 2;
ENDIF
	STOREI 2 T1
	STOREI T1 a



Code-generation – if-statement

Program text	3AC

INT a, b;
a := 2;
IF (a = 1)
  b := 1;
ELSE
  b := 2;
ENDIF
	STOREI 2 T1
	STOREI T1 a

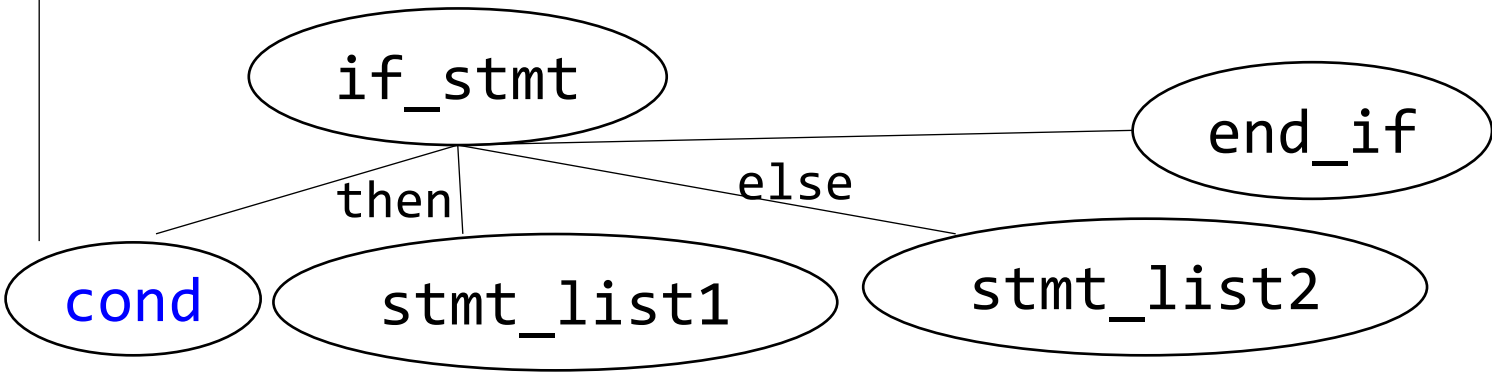


1. Generate code for cond

Code-generation – if-statement

Program text	3AC

INT a, b;
a := 2;
IF (a = 1)
  b := 1;
ELSE
  b := 2;
ENDIF
	STOREI 2 T1
	STOREI T1 a



1. Generate code for cond

```
STOREI 1 T2
NE a T2 label1
```

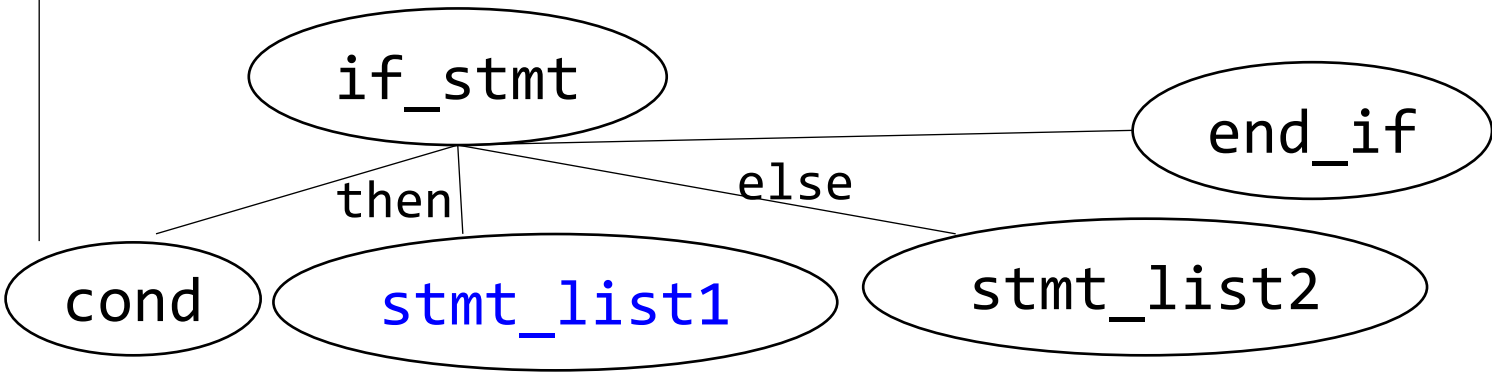
Code-generation – if-statement

Program text

```
INT a, b;
a := 2;
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```

3AC

```
STOREI 2 T1
STOREI T1 a
```

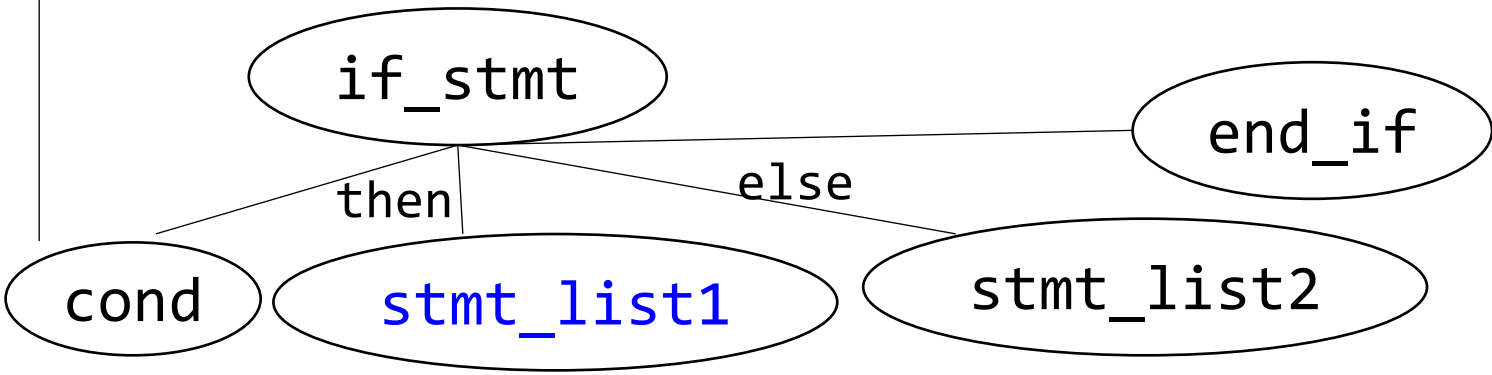


2. Generate code for stmt_list1

Code-generation – if-statement

Program text	3AC

INT a, b;
a := 2;
IF (a = 1)
  b := 1;
ELSE
  b := 2;
ENDIF
	STOREI 2 T1
	STOREI T1 a



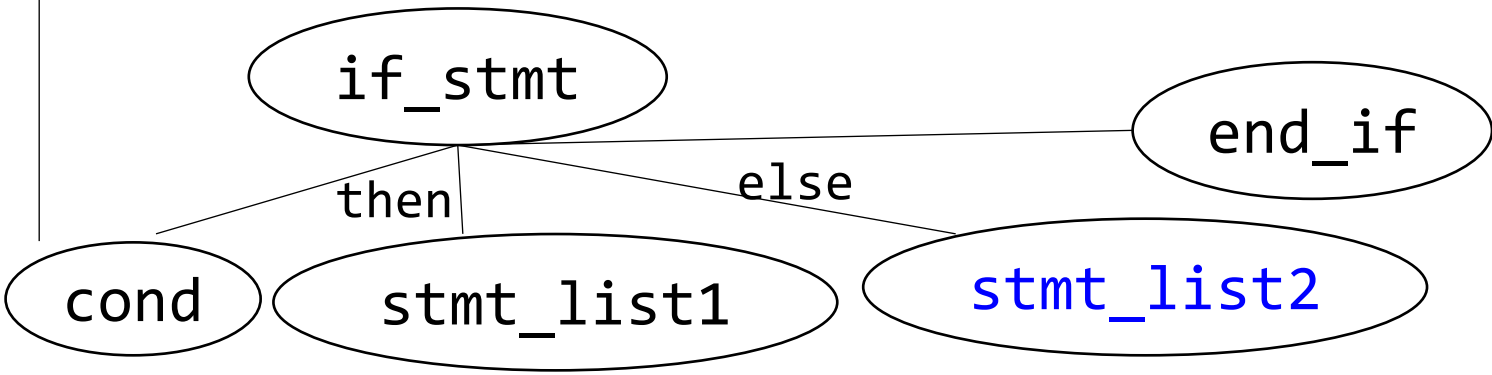
2. Generate code for stmt_list1

```
STOREI 1 T3
STOREI T3 b
JUMP label2
```

Code-generation – if-statement

Program text	3AC

INT a, b;
a := 2;
IF (a = 1)
  b := 1;
ELSE
  b := 2;
ENDIF
	STOREI 2 T1
	STOREI T1 a



3. Generate code for stmt_list2

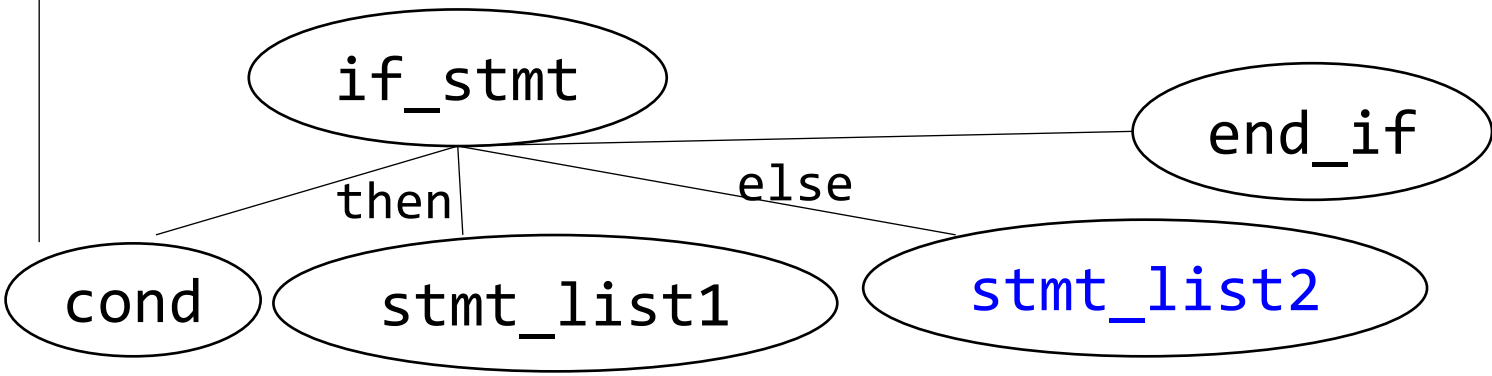
Code-generation – if-statement

Program text

```
INT a, b;
a := 2;
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```

3AC

```
STOREI 2 T1
STOREI T1 a
```



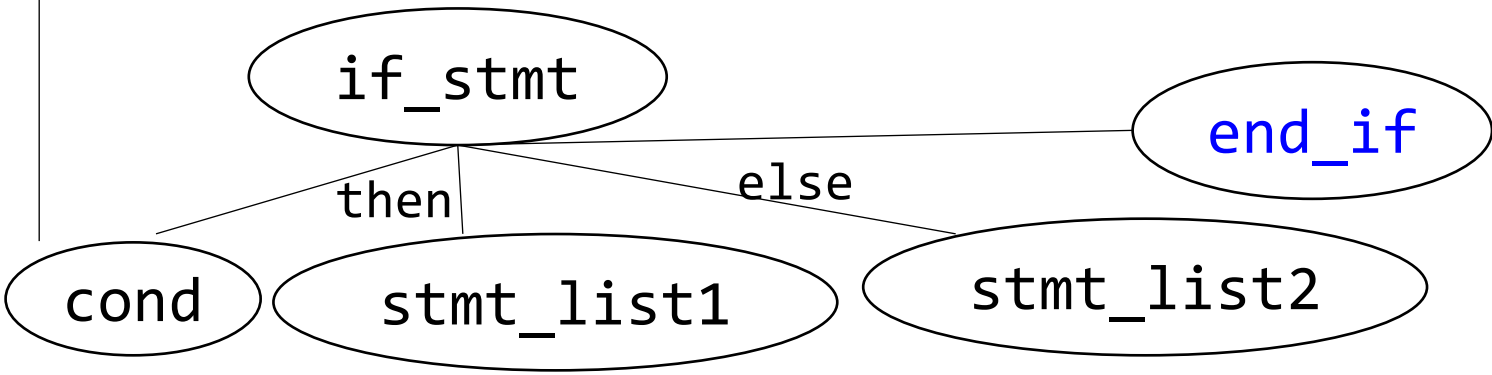
3. Generate code for stmt_list2

```
LABEL label1
STOREI 2 T4
STOREI T4 b
JUMP label2
```

Code-generation – if-statement

Program text	3AC

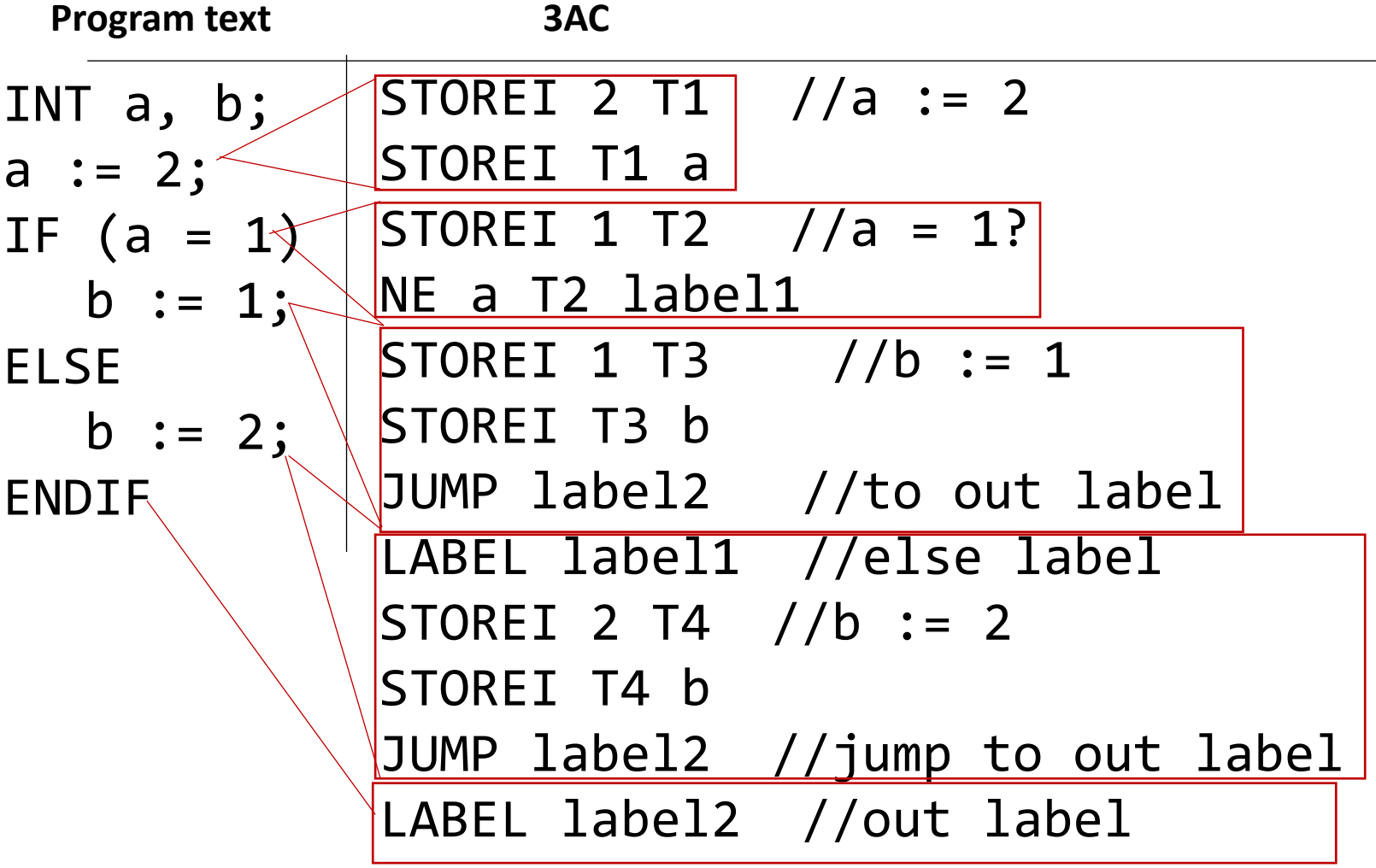
INT a, b;
a := 2;
IF (a = 1)
  b := 1;
ELSE
  b := 2;
ENDIF
	STOREI 2 T1
	STOREI T1 a



4. Generate code for end_if

LABEL label2

Code-generation – if-statement



Jumps and Labels?

- Who will generate labels?
- When will the labels be generated?
- To what addresses will the labels be associated with?

How are targets of jumps decided?

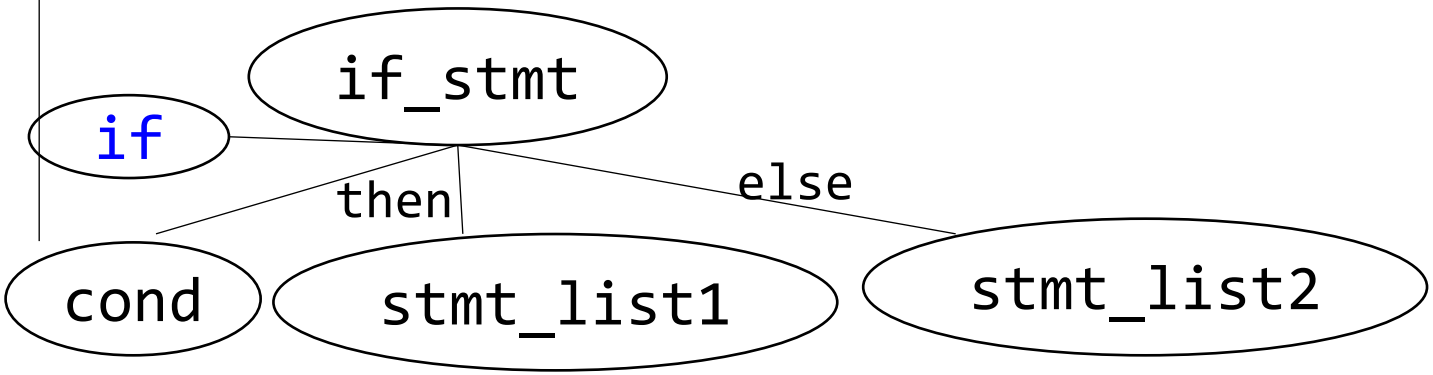
Code-generation – if-statement

Program text

```
INT a, b;
a := 2;
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```

3AC

```
STOREI 2 T1
STOREI T1 a
```



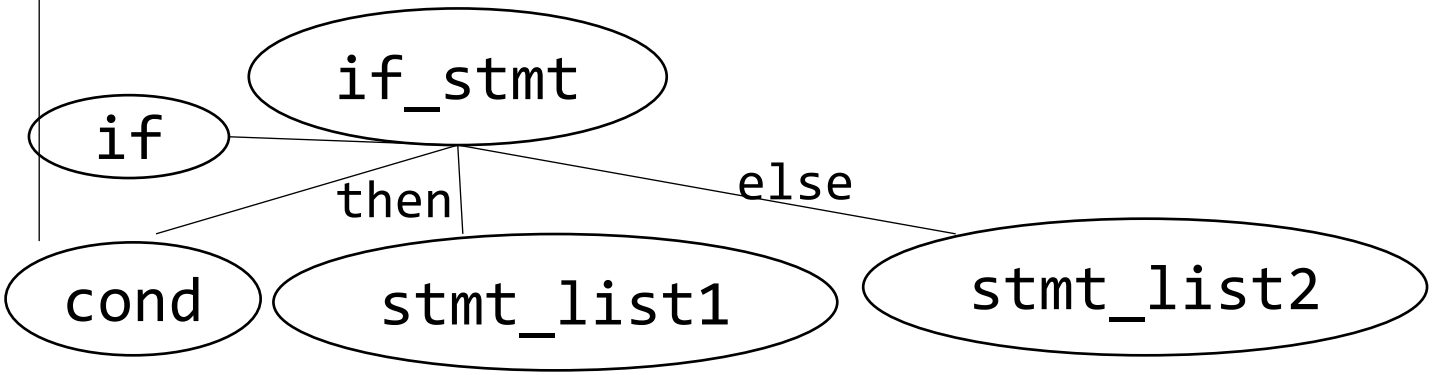
1 Generate out label and store it in semantic record of if_stmt (label2)

Code-generation – if-statement

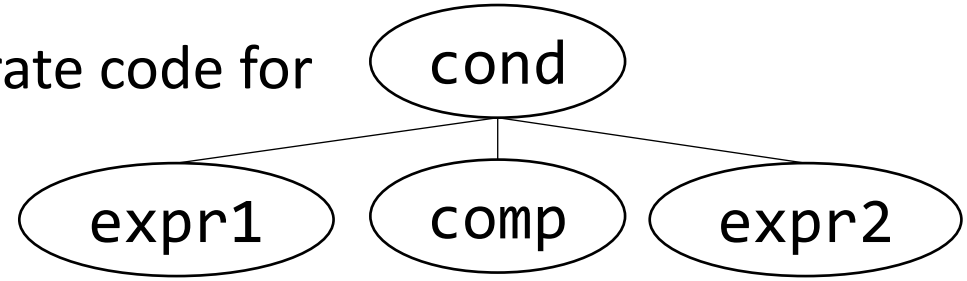
Program text | **3AC**

```
INT a, b;
a := 2;
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```

```
STOREI 2 T1
STOREI T1 a
```



2 Generate code for



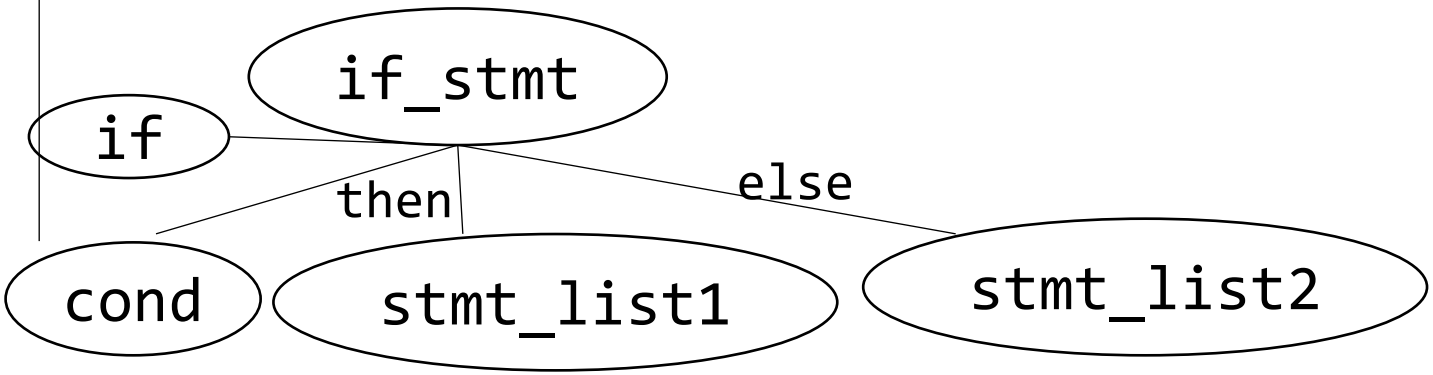

```
STOREI 1 T2
```

Code-generation – if-statement

Program text | **3AC**

```
INT a, b;
a := 2;
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```

```
STOREI 2 T1
STOREI T1 a
```



2. Store the result of calling `process_op`, where op is "=", in the node `cond` (`bool_expr1=false`)

```
STOREI 1 T2
```

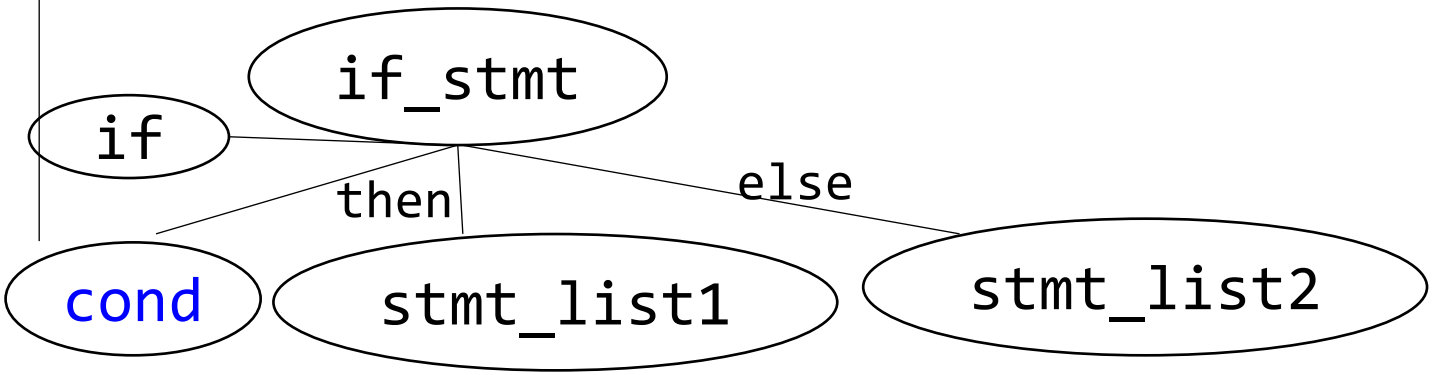
Code-generation – if-statement

Program text

```
INT a, b;
a := 2;
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```

3AC

```
STOREI 2 T1
STOREI T1 a
```



2. Cond has been matched. a) Generate label for else part (`label1`) b) generate statement: `JUMP0 bool_expr1 label1`

The generated statement conditionally jumps to the else part if cond is false.

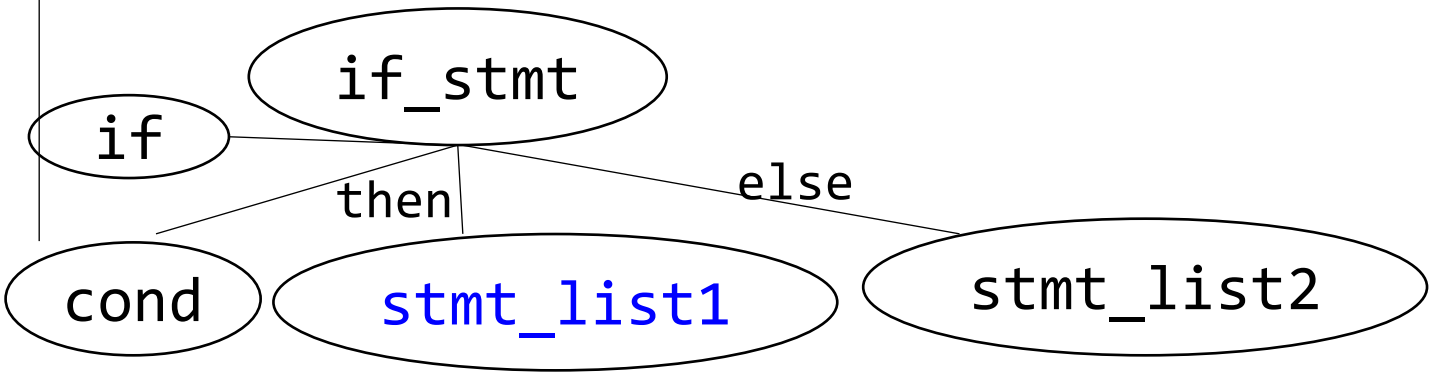
Code-generation – if-statement

Program text

```
INT a, b;
a := 2;
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```

3AC

```
STOREI 2 T1
STOREI T1 a
```



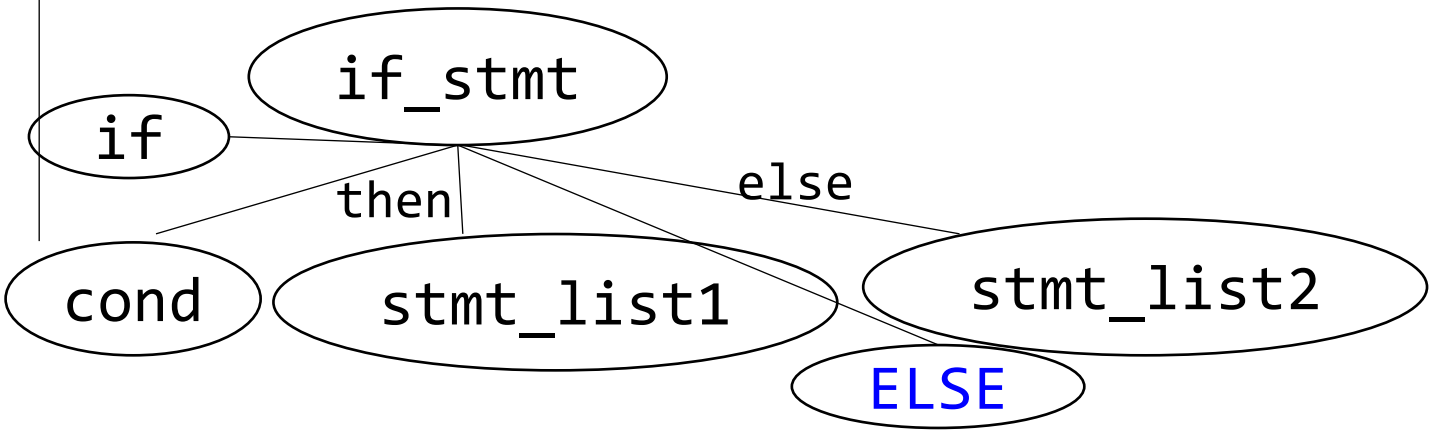
3. Generate code for `stmt_list1`

(`STOREI 1 T3`
`STOREI T3 b`)

Code-generation – if-statement

Program text | **3AC**

```
INT a, b;          STOREI 2 T1
a := 2;            STOREI T1 a
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```



4. Generate unconditional jump to out label (label2). `Label2` can be obtained from the semantic record of `if` (slide 26)

```
JUMP label2
```

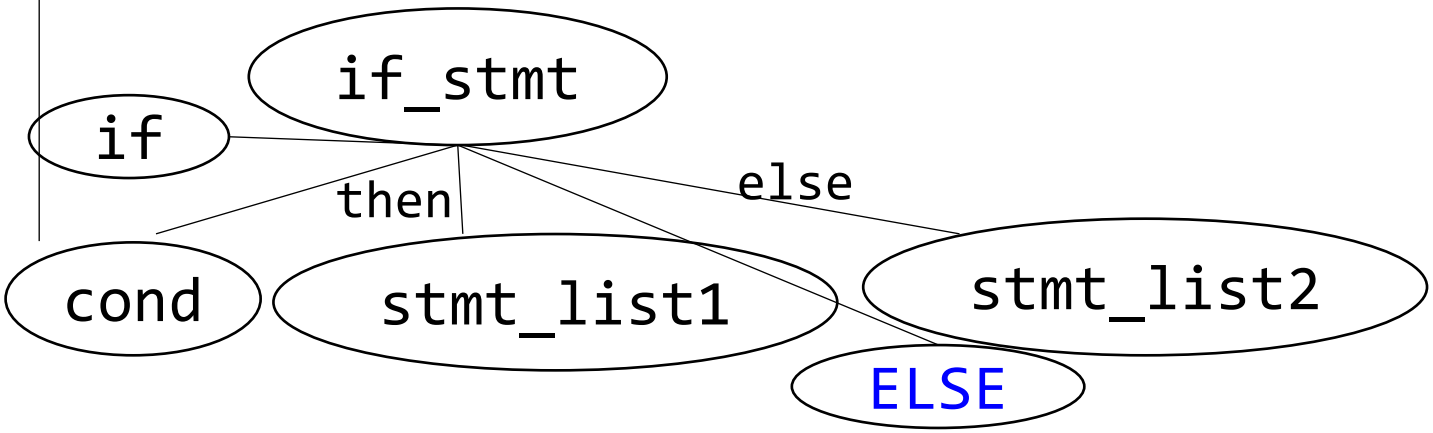
Code-generation – if-statement

Program text

```
INT a, b;
a := 2;
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```

3AC

```
STOREI 2 T1
STOREI T1 a
```



4. Associate else part label (`label1`) with address of next instruction i.e. generate a statement: `LABEL label1`
`Label1` can be obtained from semantic record of `if` updated by `cond (slide 29)`

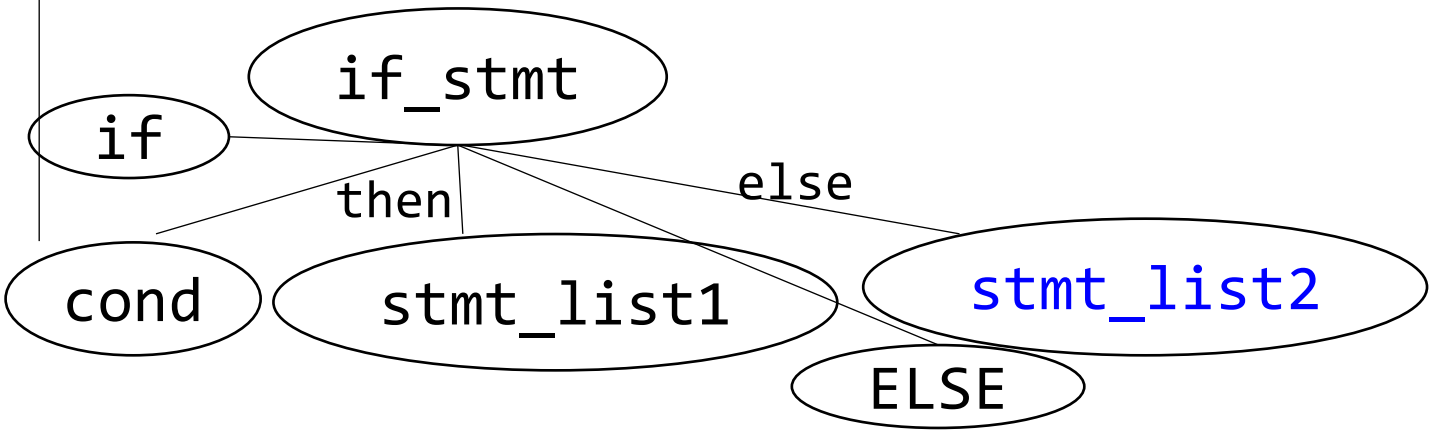
Code-generation – if-statement

Program text

```
INT a, b;
a := 2;
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```

3AC

```
STOREI 2 T1
STOREI T1 a
```



5. Generate code for `stmt_list2`

```
(STOREI 2 T4
STOREI T4 b)
```

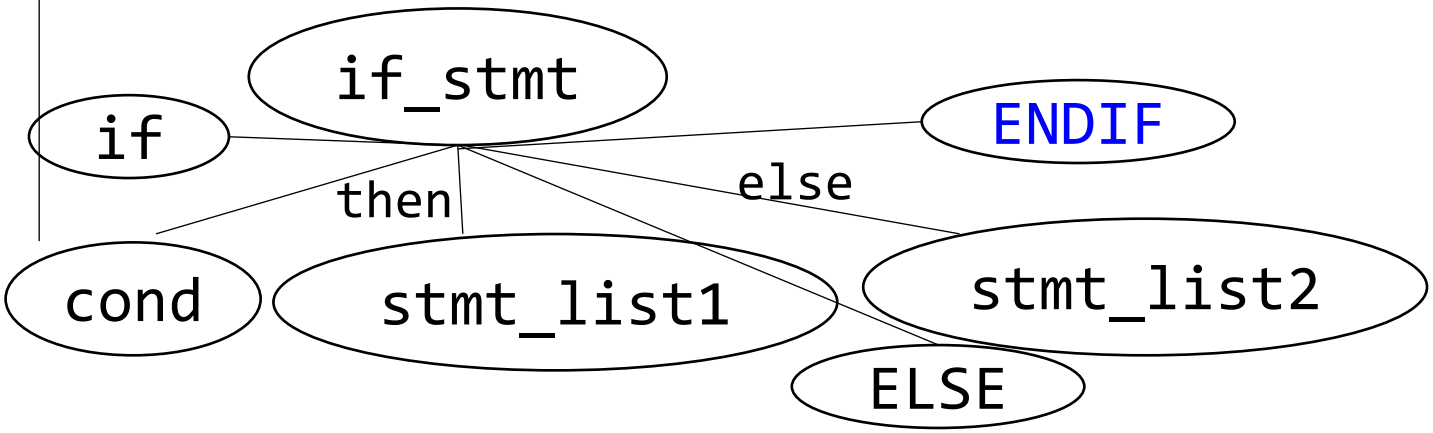
Code-generation – if-statement

Program text

```
INT a, b;
a := 2;
IF (a = 1)
   b := 1;
ELSE
   b := 2;
ENDIF
```

3AC

```
STOREI 2 T1
STOREI T1 a
```



5. Associate out label (`label2`) with address of next instruction
   i.e. generate a statement: `LABEL label2`

Observations

- We added tokens IF, ELSE, ENDIF to AST
- Generated code is equivalent but not exact
  - e.g. “NE a T2 label1” is replaced with an equivalent “JUMP0 bool_expr label1”
- Done in one pass

Will this approach work when generating machine code directly?

Generating code for ifs

```
if <bool_expr_1>
  <stmt_list_1>
else
  <stmt_list_2>
endif
```

```
  <code for bool_expr_1>
  j<!op> ELSE_1
  <code for stmt_list_1>
  jmp OUT_1
ELSE_1:
  <code for stmt_list_2>
OUT_1:
```


Slide courtesy: Milind Kulkarni

Notes on code generation

- The <op> in j<!op> is dependent on the type of comparison you are doing in <bool_expr>
- When you generate JUMP instructions, you should also generate the appropriate LABELs
- Remember: labels have to be unique!

Slide courtesy: Milind Kulkarni

do-while

- `do{S}while(B);` `//S is executed at least once and again and again and again... while B remains true`

do-while

- do{S}while(B); //S is executed at least once and again and again and again... while B remains true

```
LOOP:
   <stmt_list>
   <bool_expr>
   j<!op> OUT
   jmp LOOP
OUT:
```

repeat-until

- `repeat{S}until(B);` `//S is executed at least once and again and again and again... while B remains false`

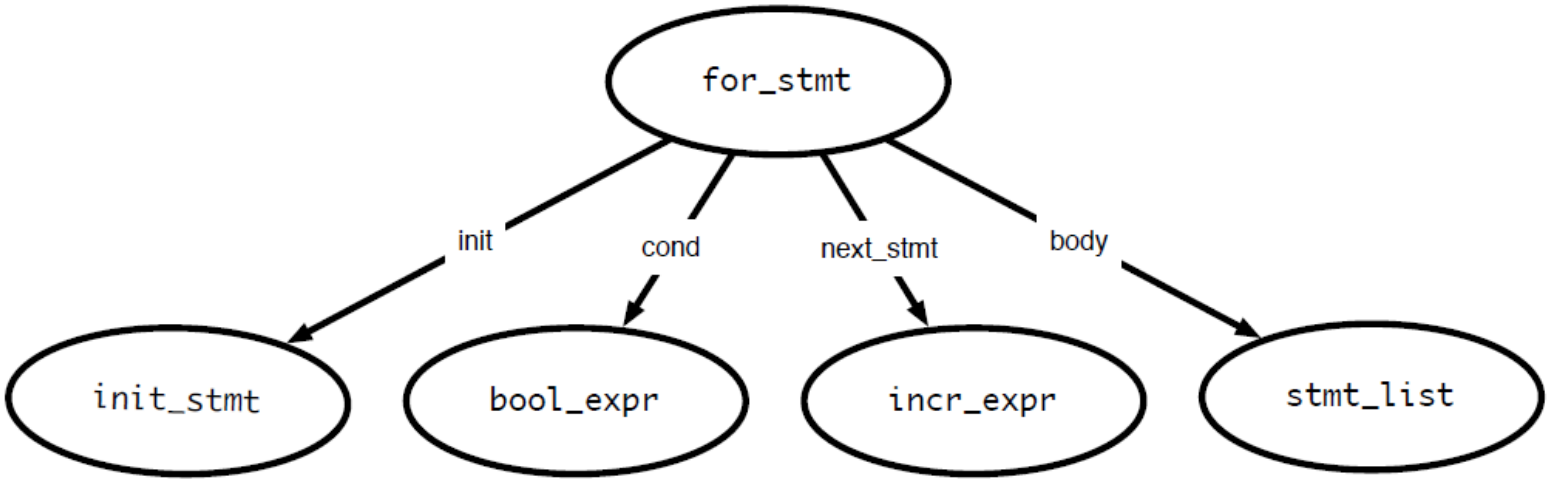
repeat-until

- `repeat{S}until(B);` `//S is executed at least once and again and again and again... while B remains false`

```
LOOP:
   <stmt_list>
   <bool_expr>
   j<!op> LOOP
OUT:
```

For loops

```
for (<init_stmt>;<bool_expr>;<incr_stmt>)
  <stmt_list>
end
```



Slide courtesy: Milind Kulkarni

Generating code: for loops

```
for (<init_stmt>;<bool_expr>;<incr_stmt>)
  <stmt_list>
end
```

```
  <init_stmt>
LOOP:
  <bool_expr>
  j<!op> OUT
  <stmt_list>
INCR:
  <incr_stmt>
  jmp LOOP
OUT:
```

- Execute init_stmt first
- Jump out of loop if bool_expr is false
- Execute incr_stmt after block, jump back to top of loop
- Question: Why do we have the INCR label?

Slide courtesy: Milind Kulkarni

Switch statements

```
switch (<expr>)
  case <const_list>: <stmt_list>
  case <const_list>: <stmt_list>
  ...
  default: <stmt_list>
end
```

- Generated code should evaluate <expr> and make sure that some case matches the result
- Question: how to decide where to jump?

Slide courtesy: Milind Kulkarni

Deciding where to jump

- Problem: do not know *which label* to jump to until switch expression is evaluated
- Use a jump table: an array indexed by case values, contains address to jump to
  - If table is not full (i.e., some possible values are skipped), can point to a default clause
    - If default clause does not exist, this can point to error code
  - Problems
    - If table is sparse, wastes a lot of space
    - If many choices, table will be very large

Slide courtesy: Milind Kulkarni

Jump table example

Consider the code:
((xxxx) is address of code)

Case x is
(0010) When 0: stmts
(0017) When 1: stmts
(0192) When 2: stmts
(0198) When 3 stmts;
(1000) When 5 stmts;
(1050) Else stmts;

Table only has one
Unnecessary row
(for choice 4)

Jump table has 6 entries:

0	JUMP 0010
1	JUMP 0017
2	JUMP 0192
3	JUMP 0198
4	JUMP 1050
5	JUMP 1000

Jump table example

Consider the code:
((xxxx) Is address of code)

Case x is
(0010) When 0: stmts0
(0017) When 1: stmts1
(0192) When 2: stmts2
(0198) When 3 stmts3
(1000) When 987 stmts4
(1050) When others stmts5

Jump table has 6 entries:

0	JUMP 0010
1	JUMP 0017
2	JUMP 0192
3	JUMP 0198
4	JUMP 1050
. . .	JUMP 1050
986	JUMP 1050
987	JUMP 1000

Table only has 983 unnecessary rows. Doesn't appear to be the right thing to do! ***NOTE: table size is proportional to range of choice clauses, not number of clauses!***

Slide courtesy: Milind Kulkarni

Linear search example

Consider the code:
(xxxx) Is offset of local Code start from the Jump instruction

Case x is
(0010) When 0: stmts
(0017) When 1: stmts
(0192) When 2: stmts
(1050) When others stmts;

If there are a small number of choices, then do an in-line linear search. A straightforward way to do this is generate code analogous to an IF THEN ELSE.

If (x == 0) then stmts1;
Elseif (x = 1) then stmts2;
Elseif (x = 2) then stmts3;
Else stmts4;

O(n) time, n is the size of the table, for each jump.

Slide courtesy: Milind Kulkarni

Dealing with jump tables

```
switch (<expr>)
  case <const_list>: <stmt_list>
  case <const_list>: <stmt_list>
  ...
  default: <stmt_list>
end
```

```
    <expr>
    <code for jump table>
  LABEL0:
    <stmt_list>
  LABEL1:
    <stmt_list>
  ...
  DEFAULT:
    <stmt_list>
  OUT:
```

- Generate labels, code, then build jump table
  - Put jump table after generated code
- Why do we need the OUT label?
  - In case of `break` statements

Slide courtesy: Milind Kulkarni

Functions

Terms

```
void foo() {
  int a, b;
  ...
  bar(a, b);
}


void bar(int x, int y) {
   ...
}
```

- foo is the *caller*
- bar is the *callee*
- a, b are the *actual parameters* to bar
- x, y are the *formal parameters* of bar
- Shorthand:
  - argument = actual parameter
  - parameter = formal parameter

Slide courtesy: Milind Kulkarni

Different Kinds of Parameters

- Value
- Reference
- Result
- Value-Reference
- Read-only
- Call-by-Name

Value parameters

- "Call-by-value"
- Used in C, Java, default in C++
- Passes the value of an argument to the function
- Makes a copy of argument when function is called
- Advantages? Disadvantages?

Advantage: 'side-effect' free – caller can be sure that the argument is not modified by the callee

Disadvantage: Not efficient for larger sized arguments.

Slide courtesy: Milind Kulkarni

Value parameters

```
int x = 1;
void main () {
    foo(x, x);
    print(x);
}

void foo(int y, int z) {
    y = 2;
    z = 3;
    print(x);
}
```

- What do the print statements print?
- Answer:

  print(x); //prints 1

  print(x); //prints 1

Reference parameters

- "Call-by-reference"
- Optional in Pascal (use "var" keyword) and C++ (use "&")
- Pass the *address* of the argument to the function
- If an argument is an expression, evaluate it, place it in memory and then pass the address of the memory location
- Advantages? Disadvantages?

Advantage: Efficiency – for larger sized arguments
Disadvantage: results in clumsy code at times (e.g. check for null pointers)

Slide courtesy: Milind Kulkarni

Reference parameters

```
int x = 1;
void main () {
    foo(x, x);
    print(x);
}

void foo(int &y, int &z) {
    y = 2;
    z = 3;
    print(x);
    print(y);
}
```

- What do the print statements print?
- Answer:

  print(x); //prints 3

  print(x); //prints 3

  print(y); //prints 3!

Slide courtesy: Milind Kulkarni

Result Parameters

- To capture the return value of a function
- Copied at the end of function into arguments of the caller
- E.g. `output` ports in Verilog `module` definitions

Slide courtesy: Milind Kulkarni

Result Parameters

```
int x = 1
void main () {
  foo(x, x);
  print(x);
}

void foo(int y, result int z) {
  y = 2;
  z = 3;
  print(x);
}
```

- What do the following statements print?

- Answer:

```
print(x); //prints 3
print(x) //prints 1
```

Value-Result Parameters

- “Copy-in copy-out”
- Evaluate argument expression, copy to parameters
- After subroutine is done, copy values of parameters back into arguments
- Results are often similar to pass-by-reference, but there are some subtle situations where they are different

Slide courtesy: Milind Kulkarni

Value-Result Parameters

```
int x = 1
void main () {
  foo(x, x);
  print(x);
}

void foo(int y, value result int z) {
  y = 2;
  z = 3;
  print(x);
}
```

- What do the following statements print?

- Answer:

```
print(x); //prints 3
print(x) //prints 1
```


Slide courtesy: Milind Kulkarni

Read-only Parameters

- Used when callee will not change value of parameters
- Read-only restriction must be enforced by compiler
- E.g. `const` parameter in C/C++
- Enforcing becomes tricky when in the presence of aliasing and control flow. E.g.

```
void foo(readonly int x, int y) {
int * p;
if (...) p = &x else p = &y
*p = 4
}
```


Slide courtesy: Milind Kulkarni

Call-by-name Parameters

- The arguments are passed to the function before evaluation
  - Usually, we evaluate the arguments before passing them
- Not used in many languages, but Haskell uses a variant

```
int x = 1
void main () {
    foo(x+2);
    print(x);
}

void foo(int y) {
    z = y + 3; //expands to z = x + 2 + 3
    print(z);
}
```


Slide courtesy: Milind Kulkarni

Call-by-name Parameters

- Why is this useful?
  - E.g. to analyze certain properties of a program/function – termination

```
void main () {
    foo(bar());
}

void foo(int y) {
    z = 3;
    if(z > 3)
        z = y + z;
}
```

  - Even if bar has an infinite loop, the program terminates.

Slide courtesy: Milind Kulkarni

Program Layout in Memory

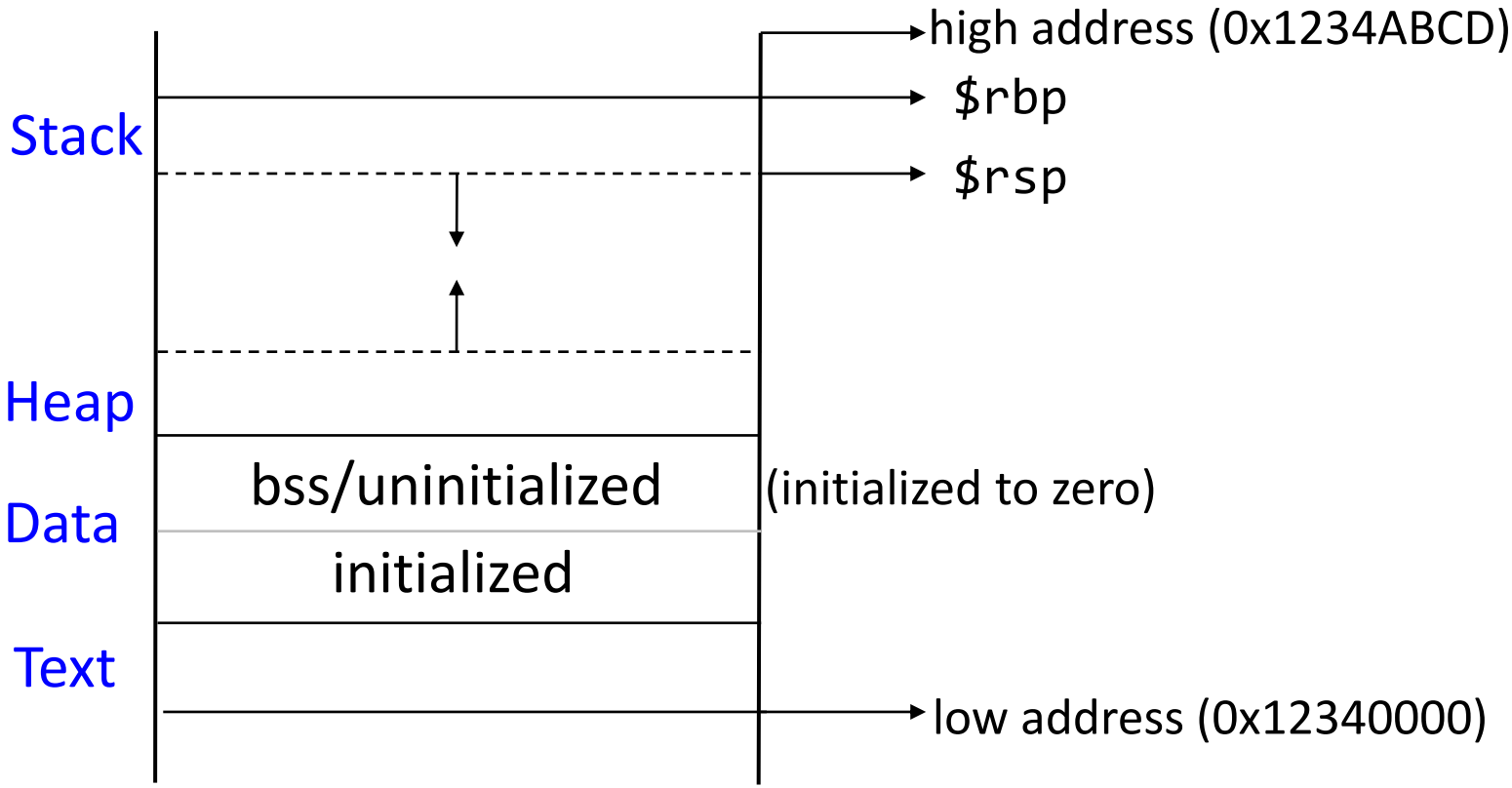
- Compiler assumes a *runtime environment* for execution of the program.
- A C/C++ program in Linux OS has 4 segments of memory
  - Every memory location is a box holding data/instruction

Program Layout in Memory

- A program's memory space is divided into four segments:

1. Text
   - source code of the program
2. Data
   - Broken into uninitialized and initialized segments; contains space for global and static variables. E.g. `int x = 7; int y;`
3. Heap
   - Memory allocated using malloc/calloc/realloc
4. Stack
   - Function arguments, return values, local variables, special registers.

Program Layout in Memory



Activation

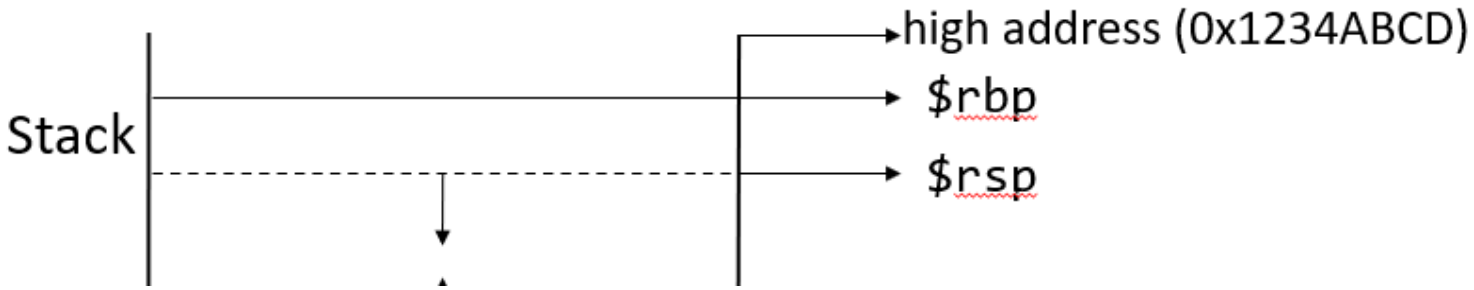
- A function call or invocation is termed an *activation*
- Calls to functions in a program form *activation tree*
  - Postorder traversal of the tree shows return sequence i.e. the order in which control returns from functions
  - Preorder traversal of the tree shows calling sequence
- In a sequential program, at any point in time, *control of execution is in any one activation*
  - All the ancestors of that activation are active i.e. have not returned

Activation

- Activations are managed through the help of *control stack*
- A function call (activation) results in allocating a chunk of memory called *activation record* or *frame* on the stack

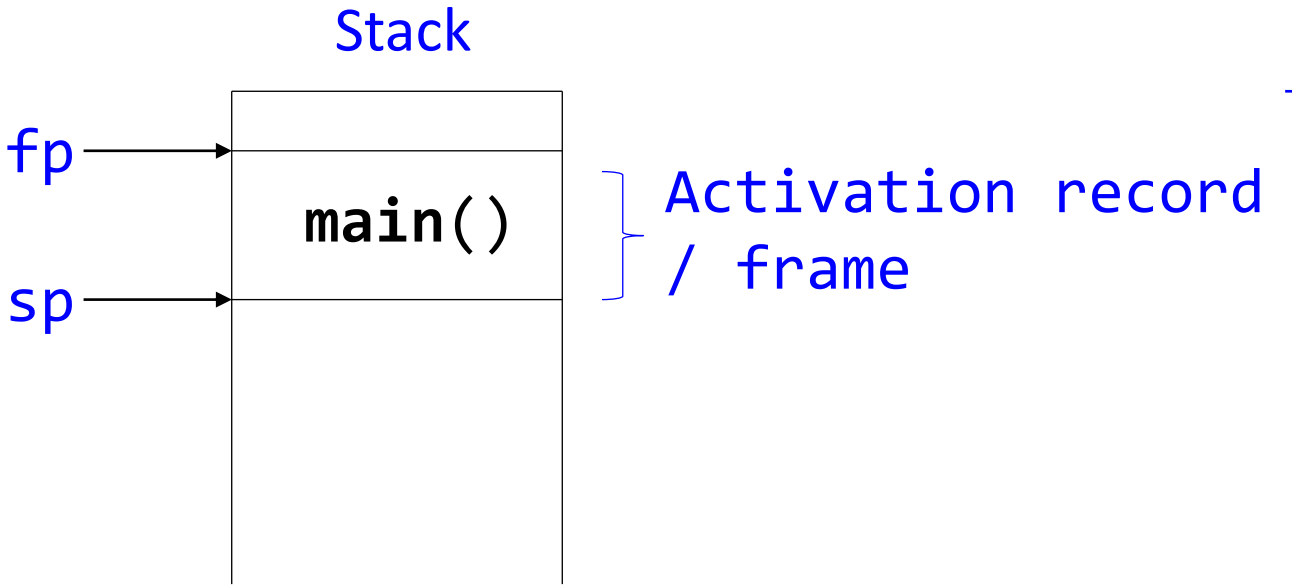
Activation Record

- A sub-segment of memory on the stack
  - Special registers `$rbp` and `$rsp` track the bottom and top of the stack frame. These are the names in x86 architecture.



  - `$rbp` – base pointer or frame pointer (fp)
  - `$rsp` – stack pointer (sp)

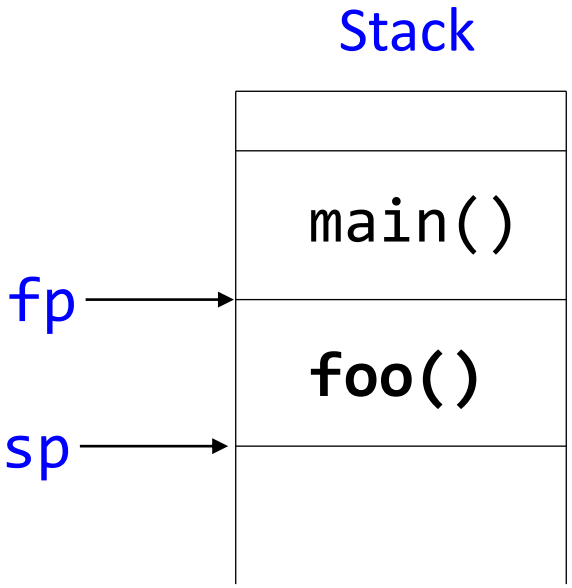
Activation Record - Example




```
main() {
    …
    foo();
    …
}

foo() {
    bar();
    …
    baz();
}
```

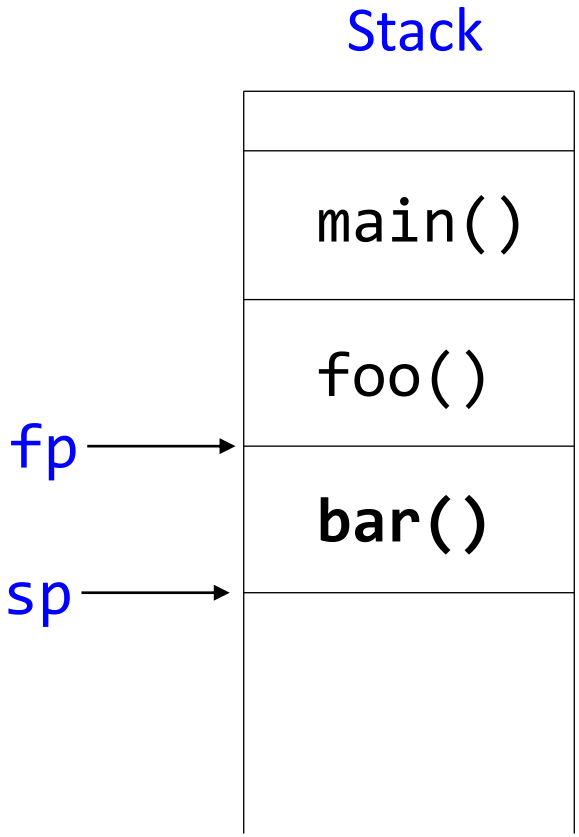
Activation Record - Example




```
main() {
    …
    foo();
    …
}

foo() {
   bar();
   …
   baz();
}
```

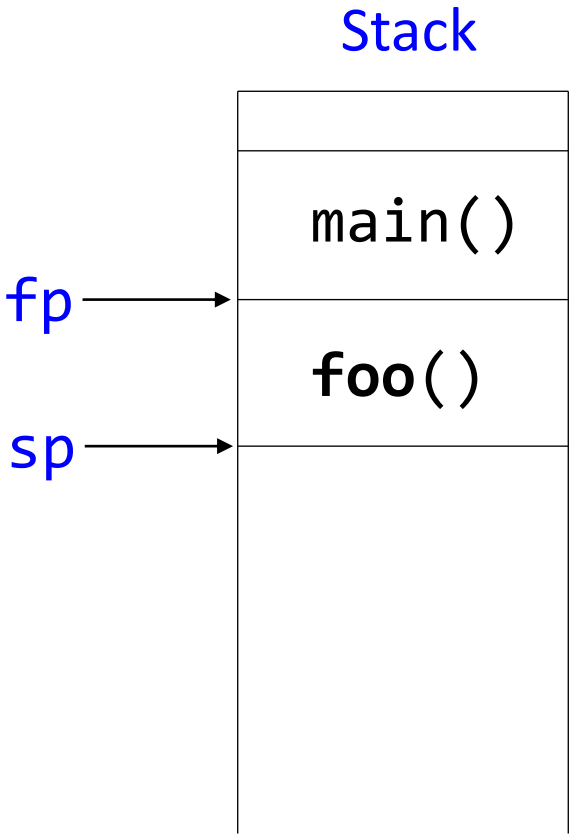
Activation Record - Example




```
main() {
   …
   foo();
   …
}

foo() {
   bar();
   …
   baz();
}
```

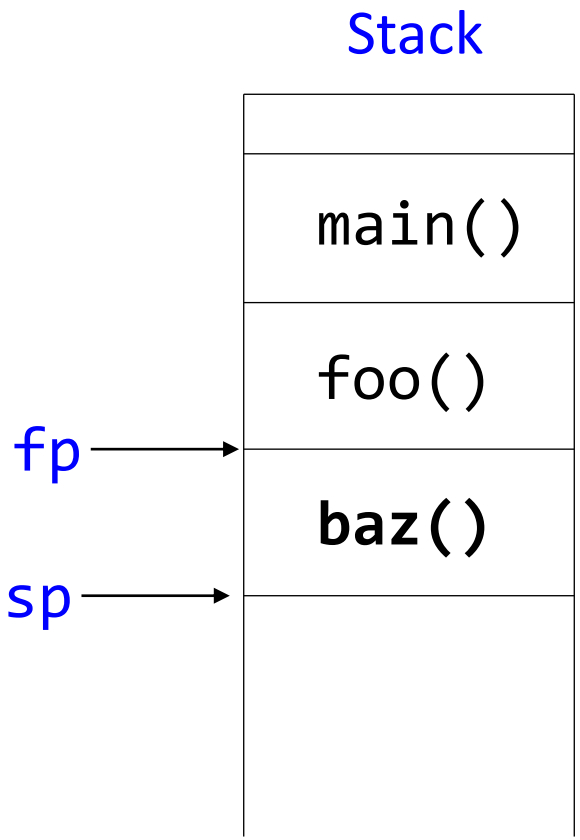
Activation Record - Example




```
main() {
   …
   foo();
   …
}

foo() {
   bar();
   …
   baz();
}
```

Activation Record - Example




```
main() {
   …
   foo();
   …
}

foo() {
   bar();
   …
   baz();
}
```

Activation Record - Observations

- What happens when a function is called?
  1. `fp` and `sp` get adjusted
  2. Memory for the activation record is allocated on stack
     - The size of the memory allocated depends on local variables used by the called function (consult function's symbol table for this)

Activation Record

- What is stored in the activation record?

  Depends on the language being implemented:
  - Temporaries
  - Local vars
  - Saved registers
  - Return address, previous `fp`
  - Return value
  - Actual Params
- Who stores this information?
  - Caller
  - Callee

  } together execute *calling sequence* and *return sequence*

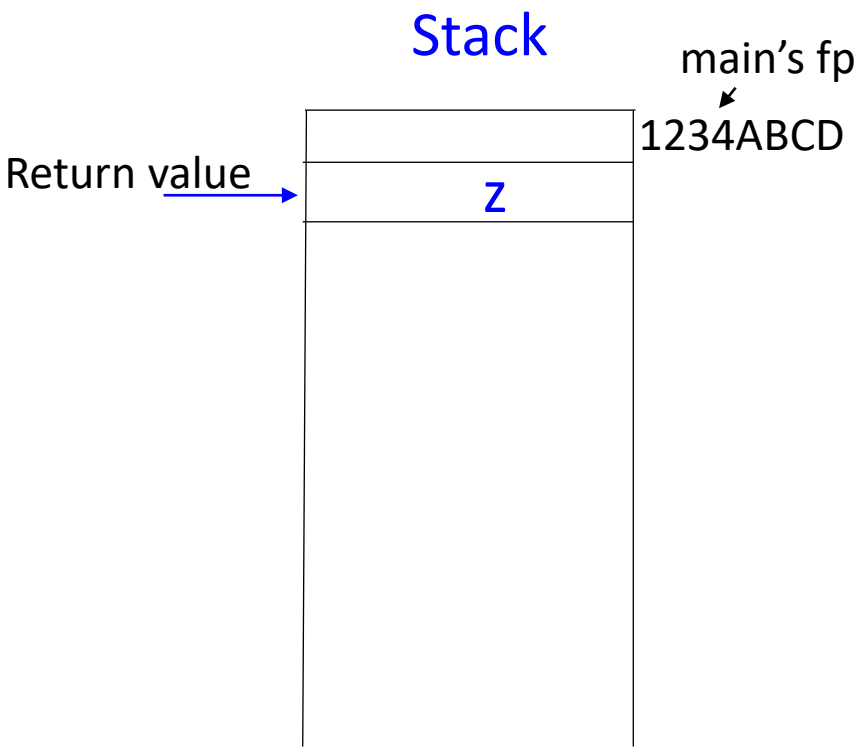
Activation Record - Observations

- When `main` calls function `foo`
  1. The following are pushed on to stack:
     1. `foo`'s arguments
     2. Space for `foo`'s return value
     3. Address of the next instruction executed (in `main`) when `foo` returns
     4. Current value of `$rbp`

     `$rsp` is automatically updated (decremented) to point to current top of the stack.

  2. `$rbp` is assigned the value of `$rsp`

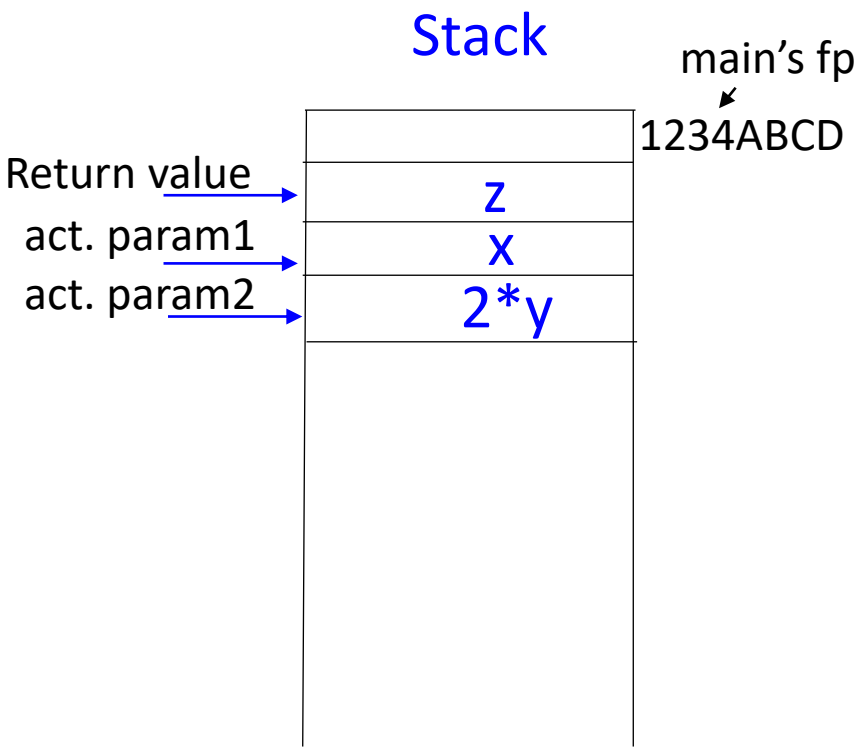
Activation Record - Example




```
main() {
   z=foo(x, 2*y);
   return;
}

int foo(int a, int b) {
   int l1, l2
   l1=a;
   l2=b;
   return l1+l2;
}
```

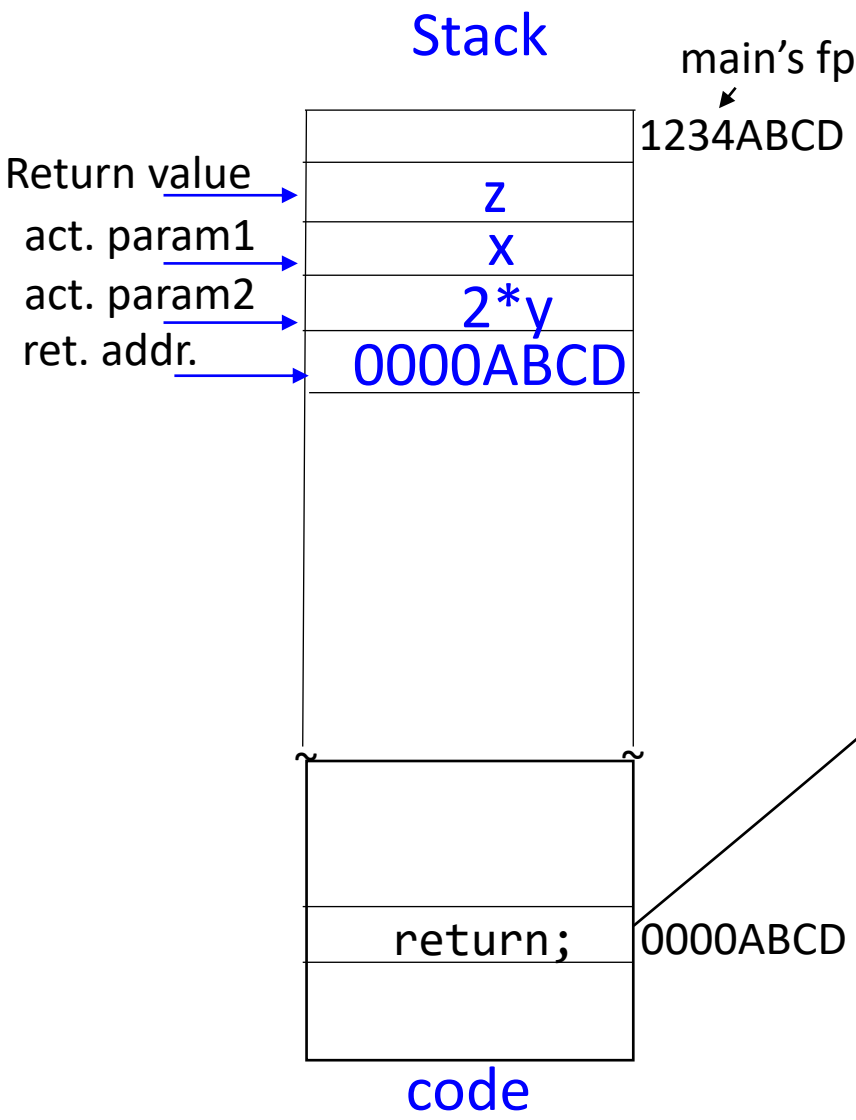
Activation Record - Example




```
main() {
   z=foo(x, 2*y);
   return;
}

int foo(int a, int b) {
   int l1, l2
   l1=a;
   l2=b;
   return l1+l2;
}
```

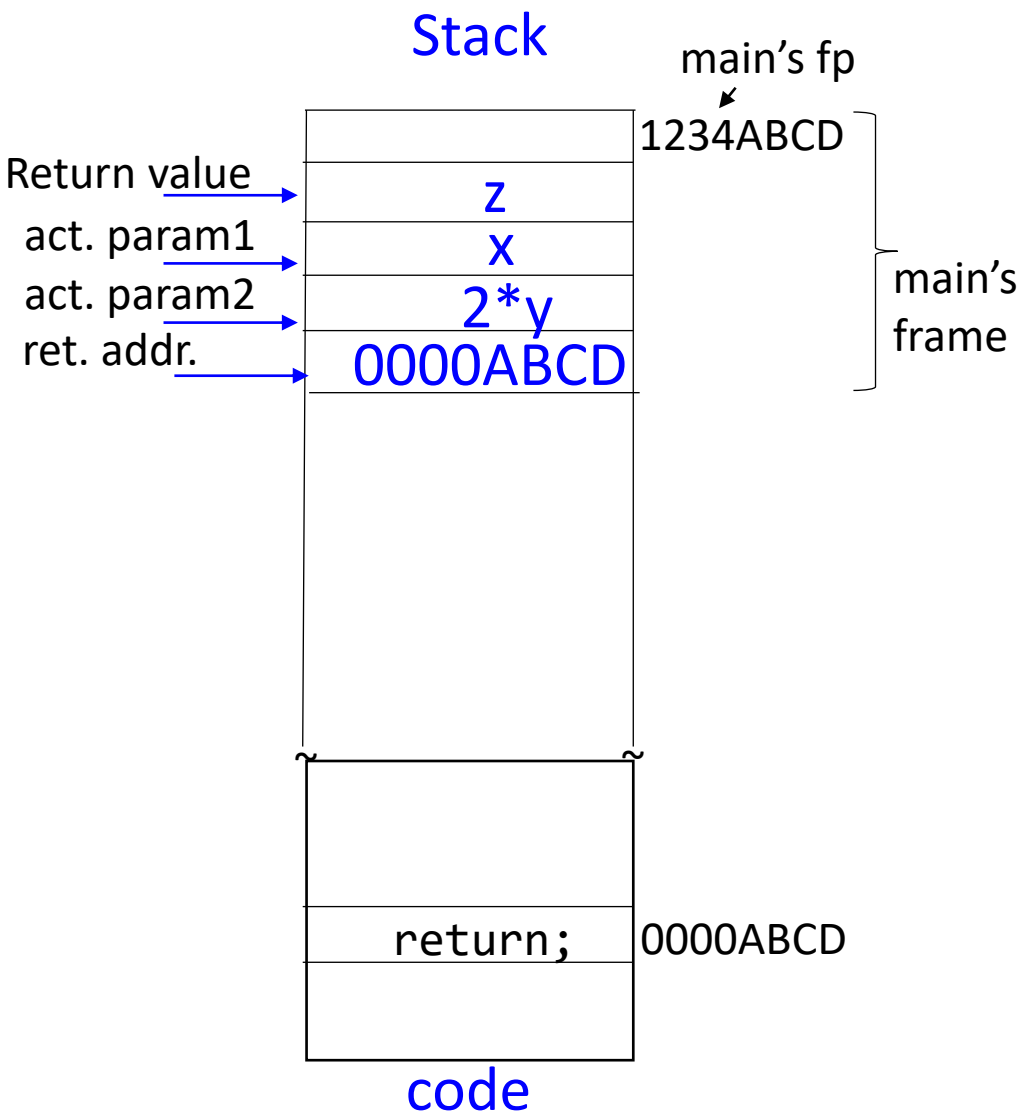
Activation Record - Example




```
main() {
   z=foo(x, 2*y);
   return;
}

int foo(int a, int b) {
   int l1, l2
   l1=a;
   l2=b;
   return l1+l2;
}
```

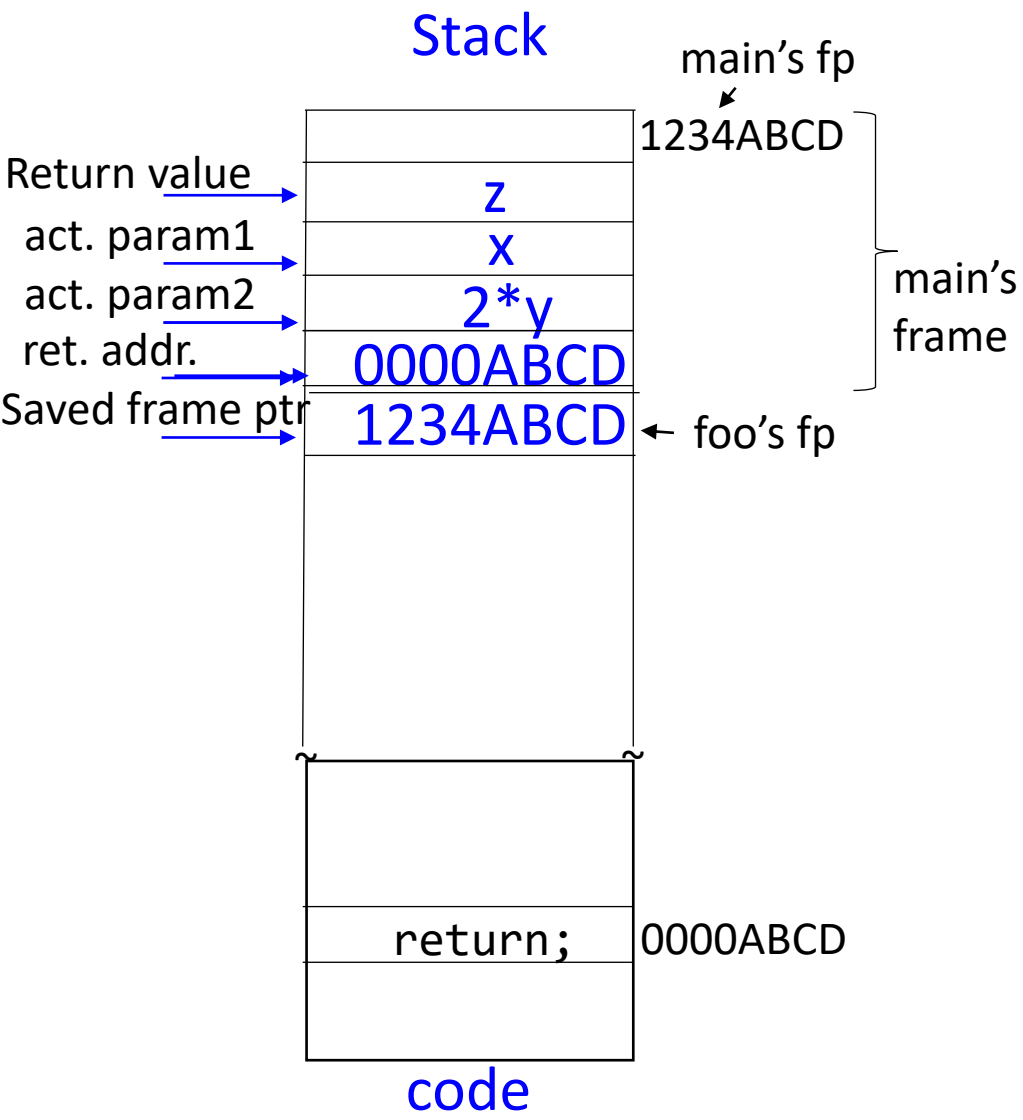
Activation Record - Example




```
main() {
   z=foo(x, 2*y);
   return;
}

int foo(int a, int b) {
   int l1, l2
   l1=a;
   l2=b;
   return l1+l2;
}
```

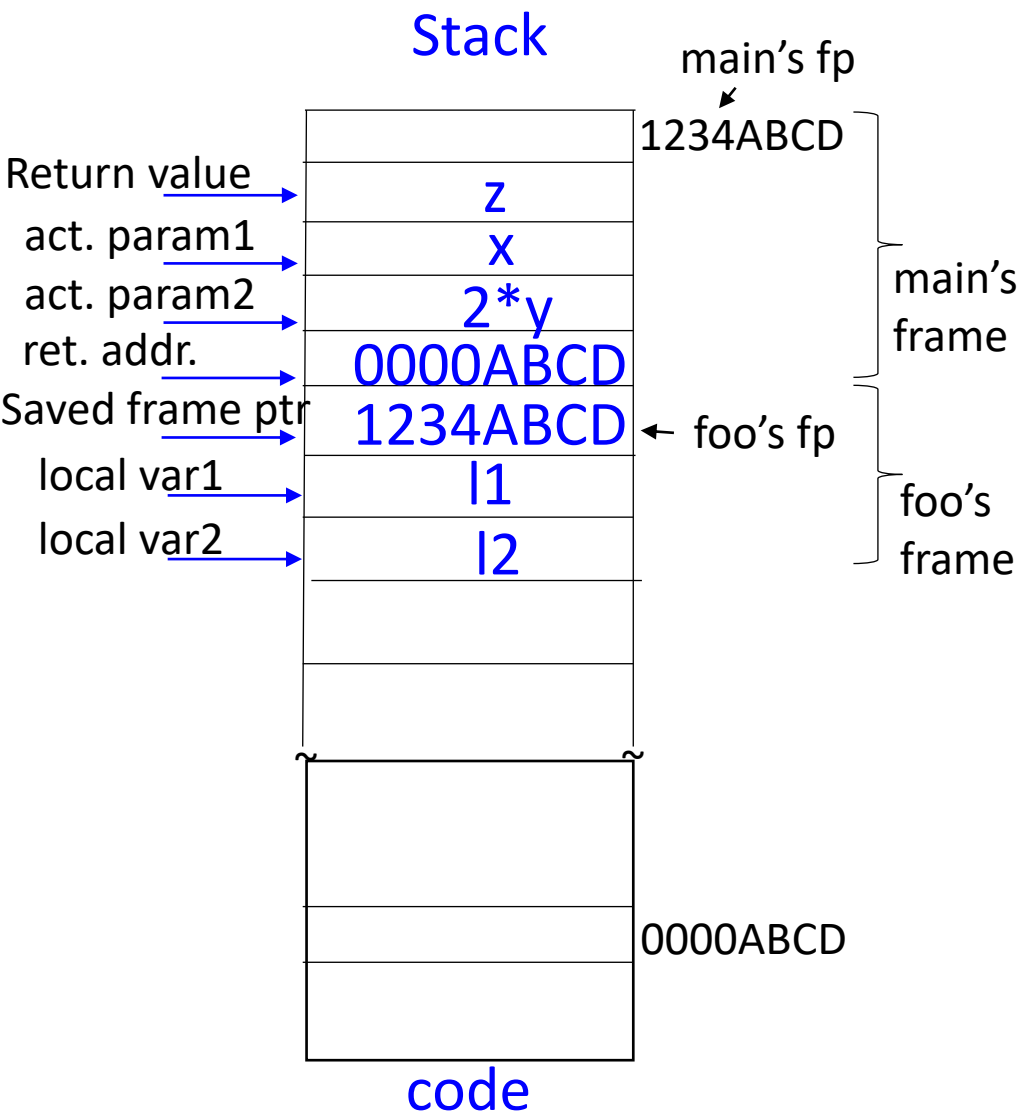
Activation Record - Example




```
main() {
   z=foo(x, 2*y);
   return;
}

int foo(int a, int b) {
   int l1, l2
   l1=a;
   l2=b;
   return l1+l2;
}
```

Activation Record - Example



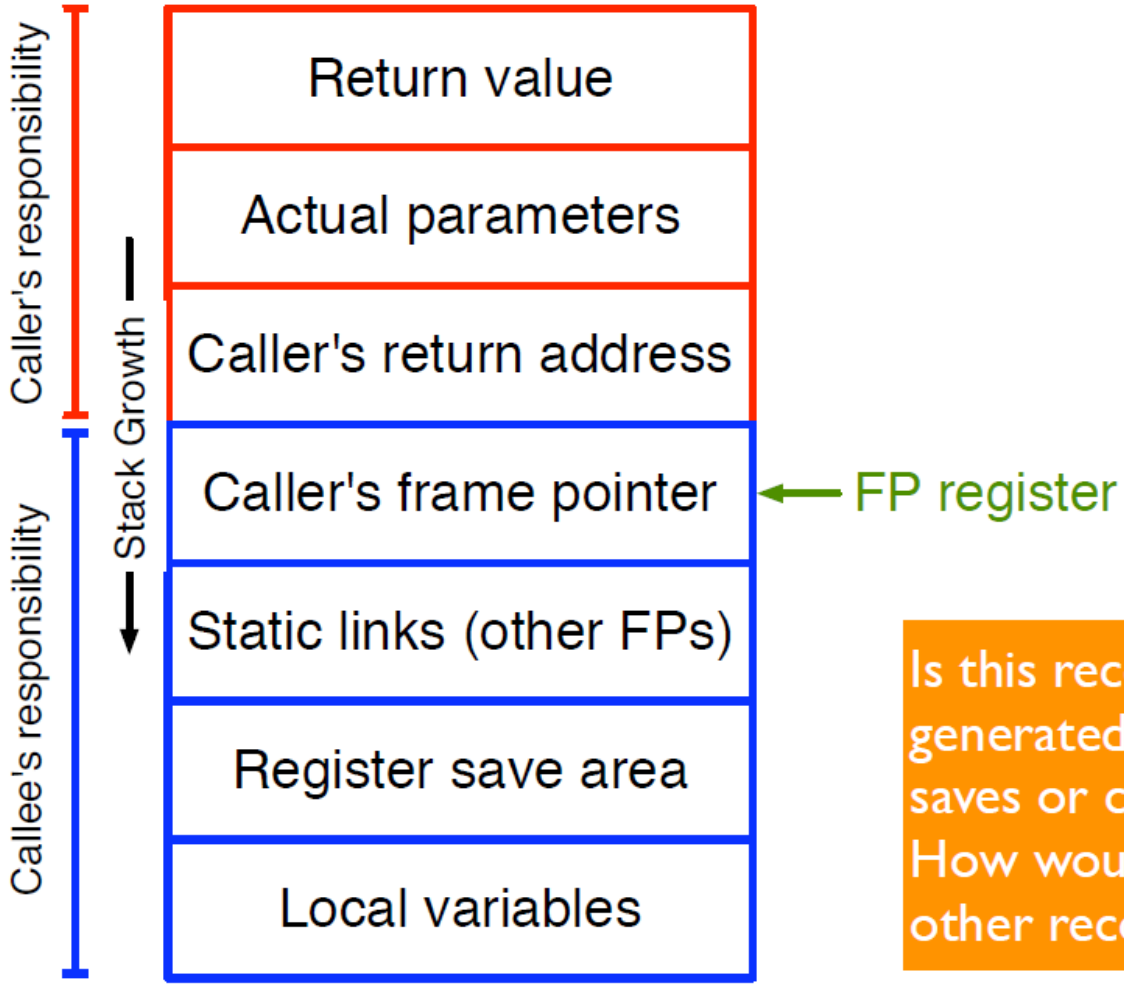

```
main() {
   z=foo(x, 2*y);
   return;
}

int foo(int a, int b) {
   int l1, l2
   l1=a;
   l2=b;
   return l1+l2;
}
```

Activation Record – Saving Registers

- Did not save registers in the previous example
- Two options: caller saves or callee saves
- Caller Saves
  - Caller pushes all the registers it is using on to the stack before calling the function
  - Restores the registers after the function returns
- Callee Saves
  - Callee pushes all the registers it is *going to use* on the stack immediately after being called
  - Restores the registers just before it returns

Activation records



Is this record generated for callee-saves or caller-saves? How would the other record look?

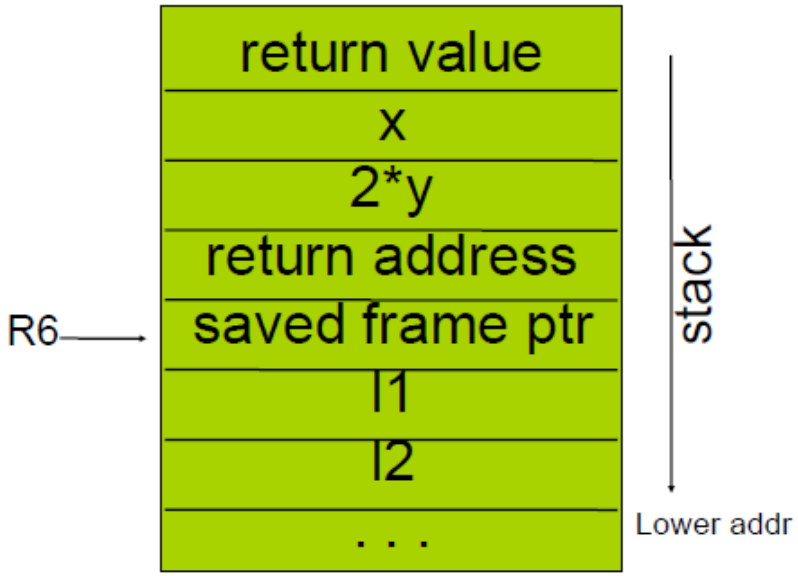
Slide courtesy: Milind Kulkarni

The frame pointer

- Manipulate with instructions like `link` and `unlink`
  - Link: push current value of FP on to stack, set FP to top of stack
  - Unlink: read value at current address pointed to by FP, set FP to point to that value
  - In other words: link pushes a new frame onto the stack, unlink pops it off

Slide courtesy: Milind Kulkarni

Example Subroutine Call and Stack Frame



3-address code:

```
push
push x
mul  2  y  t1
push t1
jsr SubOne
pop
pop
pop  z
```

assembly code:

```
push
push x
load  y R1
muli  2 R1
push R1
jsr SubOne
pop
pop
pop  R1
store R1 z
```

```
z = SubOne(x,2*y);
```

```
int SubOne(int a, int b) {
      int l1, l2;
      l1 = a;
      l2 = b;
      return l1+l2;
   };
```

```
link  3
move $P1 $L1
move $P2 $L2
add $L1 $L2 t2
move t2 $R
unlink
ret
```

```
link R6 3
load  3(R6)  R1
store R1 -1(R6)
load  2(R6)  R2
store R2 -2(R6)
load  -1(R6)  R1
add   -2(R6)  R1
store R1  4(R6)
unlink
ret
```

Slide courtesy: Milind Kulkarni

Application Binary Interface (ABI)

- Specification on how data is provided to functions
  - Caller saves or callee saves
- Meant to deliver interoperability between different compilers
  - Create object code using one compiler, Link object code with other code compiled using a different compiler

Question ?

Where are the command-line arguments stored?

How about environment variables such as LD_LIBRARY_PATH and PATH?