CS406: Compilers

Spring 2022

Week 7: Intermediate Code Generation (if, do-while, for)

If construct with semantic actions

- If_stmt->if #start_if <b_expr> #testif then <stmt_list> <else_part> endif; #gen_out_label
- else_part->else #gen_jump #gen_else_label <stmt_list>

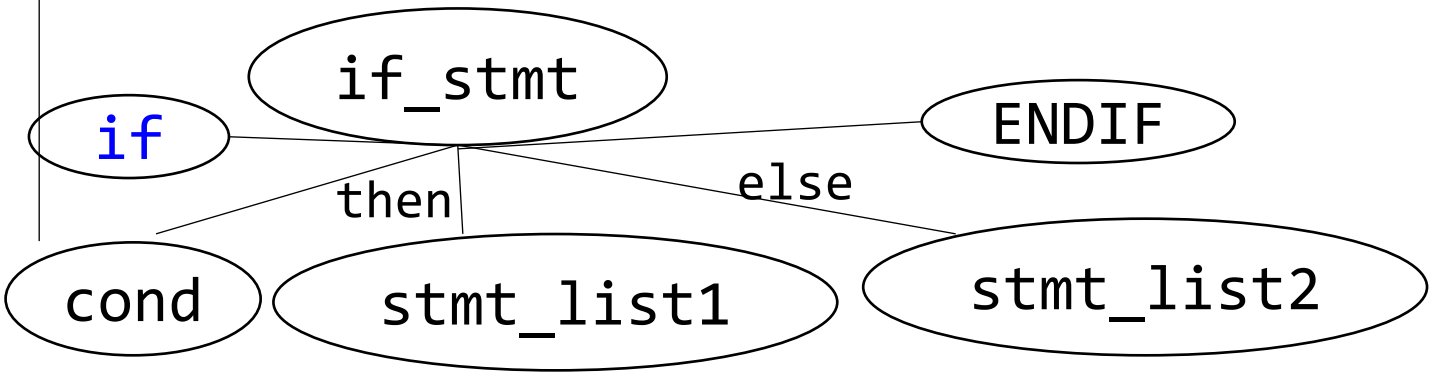
Code-generation – if-statement

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 (label1)

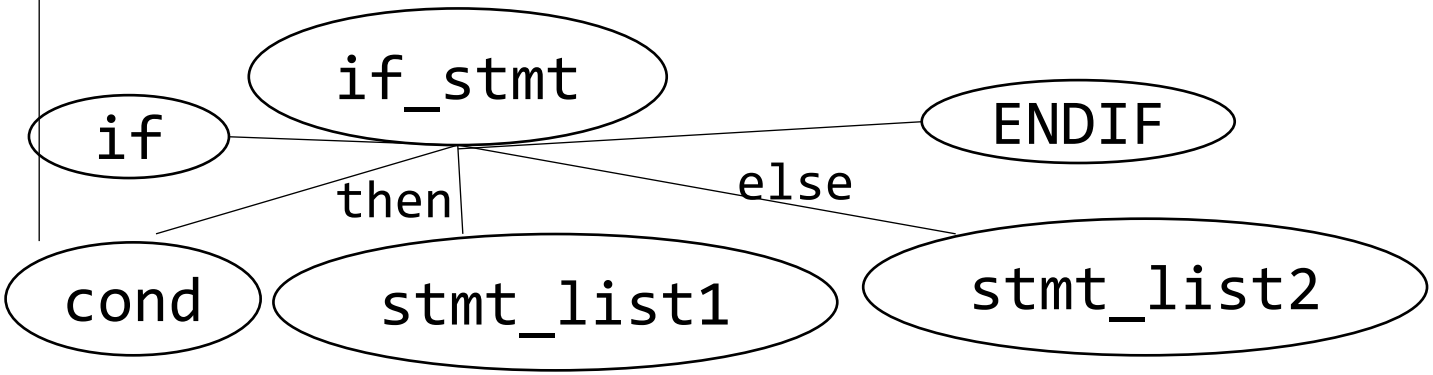
The #start_if routine is responsible for this

Code-generation – if-statement

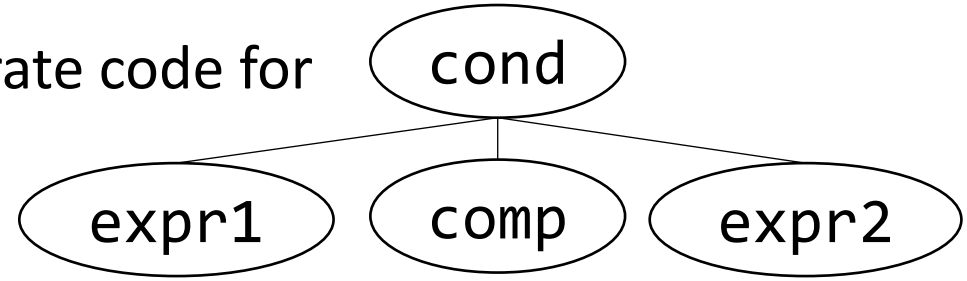
Program text | **3AC**

```
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
```

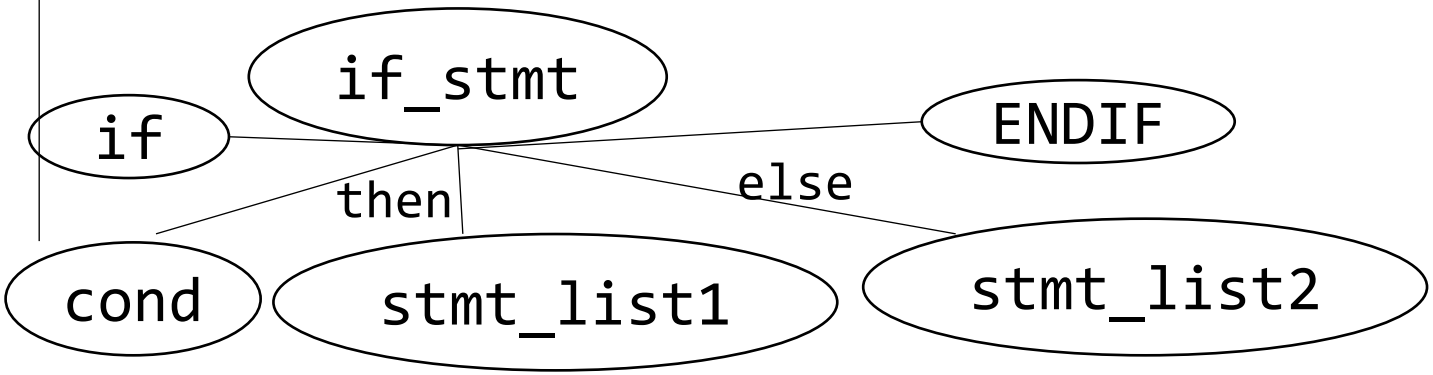
Code-generation – if-statement

Program text

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

3AC

```
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
```

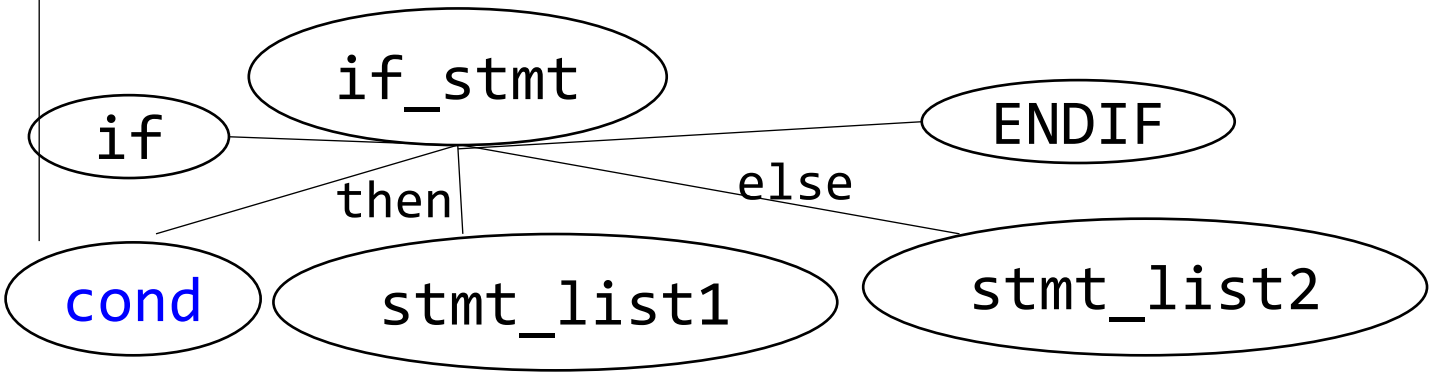
Code-generation – if-statement

Program text | **3AC**

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

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

The #testif routine handles this.



2. Create a label for the next else part (label 2). Generate statement: `JUMP0 T2 label2`

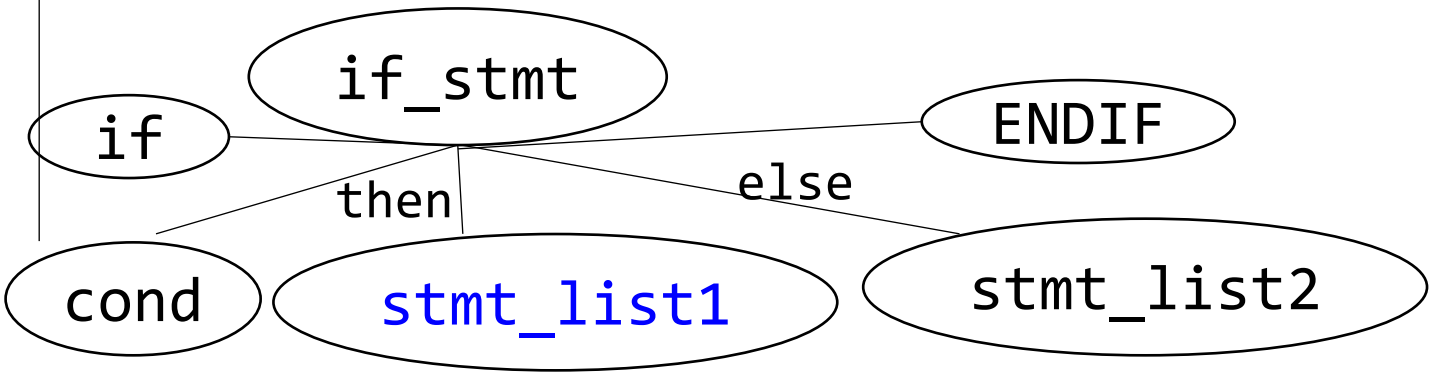
Code-generation – if-statement

Program text

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

3AC

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



3. Generate code for `stmt_list1`

```
STOREI 1 T3
STOREI T3 b
```

Code-generation – if-statement

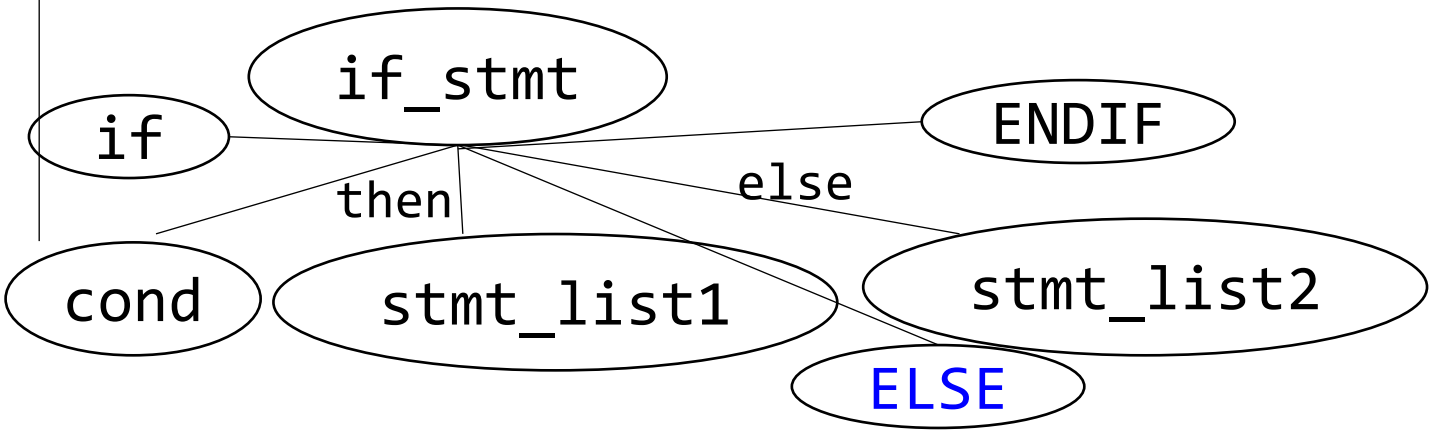
Program text

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

3AC

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

Routine #gen_jump handles this



4. Generate unconditional jump to out label (label1).

```
JUMP label1
```

Code-generation – if-statement

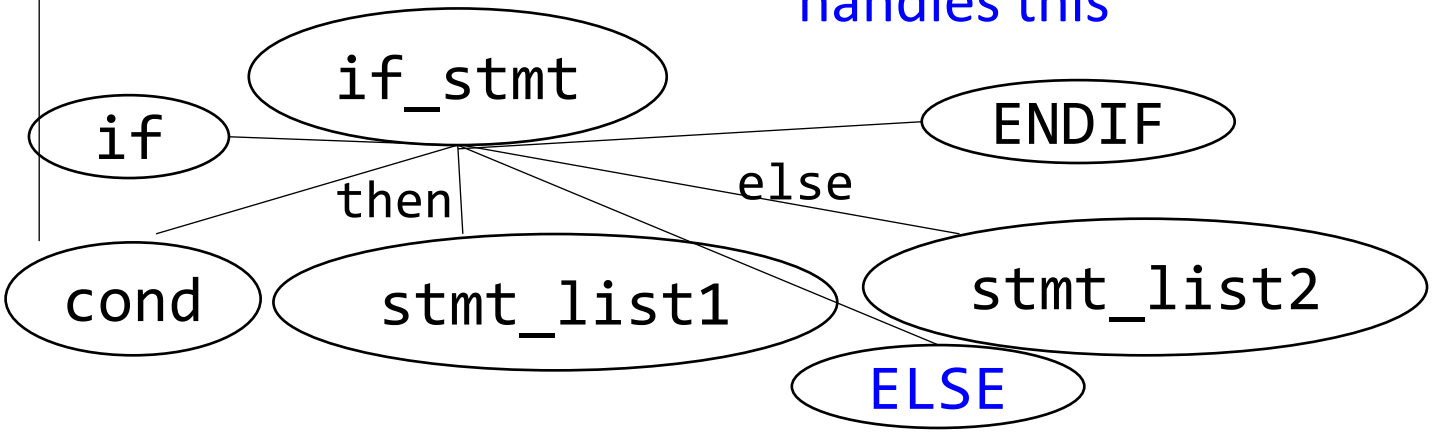
Program text

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

3AC

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

Routine #gen_else_label handles this



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

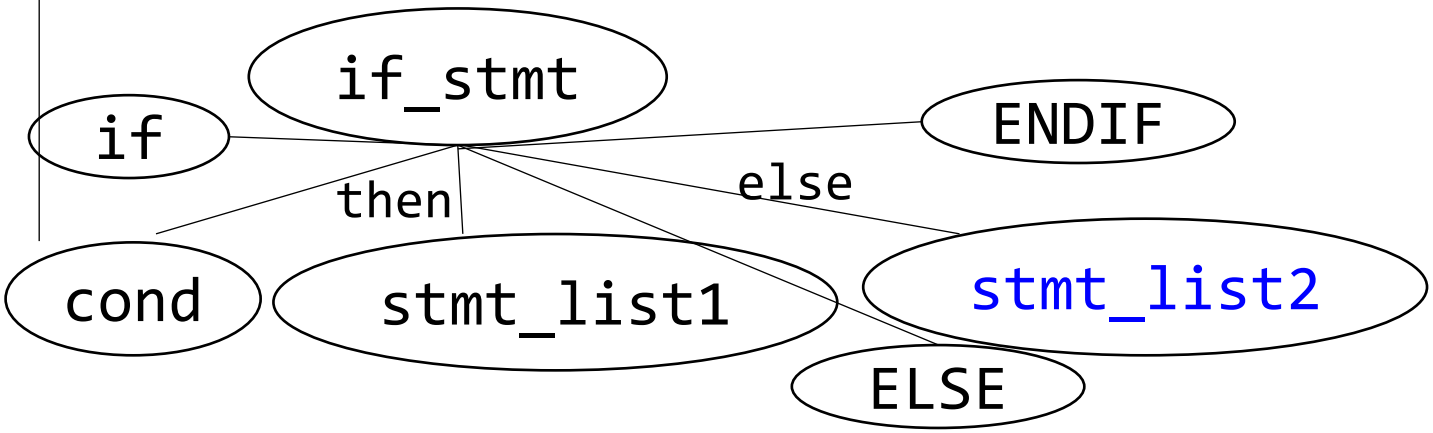
Code-generation – if-statement

Program text

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

3AC

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



5. Generate code for `stmt_list2`

```
STOREI 2 T4
STOREI T4 b
```

Code-generation – if-statement

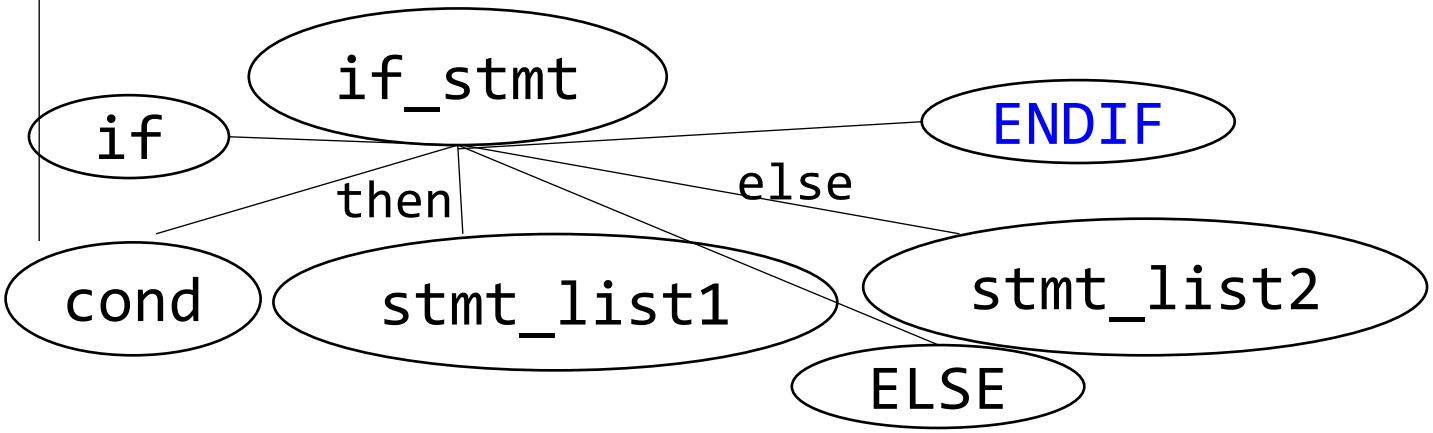
Program text

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

3AC

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

Routine #gen_out_label handles this



5. Associate out label (`label1`) with address of next instruction
   i.e. generate a statement: `LABEL label1`

Observations

- We added semantic actions with tokens IF, ELSE, ENDIF
- 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?

If construct with semantic actions

- If_stmt->if #start_if <b_expr> #testif then <stmt_list> <optional_elsif_part> <else_part> endif; #gen_out_label
- <optional_elsif_part>-> elsif #gen_jump #gen_else_label <b_expr> #testif then <stmts>
- Else_part->else #gen_jump #gen_else_label <stmt_list>

Exercise: augment the grammar rule to handle elsif blocks.

Code-generation – if-statement

Program text	3AC

INT a, b;
a := 2;
IF (a = 1)
    b := 1;
ELSIF (TRUE)
    b := 2;
ENDIF
	STOREI 2 T1 //a := 2
	STOREI T1 a
	STOREI 1 T2 //a = 1?
	NE a T2 label1
	STOREI 1 T3 //b := 1
	STOREI T3 b
	JUMP label2 //to out label
	LABEL label1 //elsif label
	STOREI 1 T4 //TRUE can be handled by checking 1 = 1?
	STOREI 1 T5
	NE T4 T5 label3 //jump to the next elsif label
	STOREI 2 T6 //b := 2
	STOREI T6 b
	JUMP label2 //jump to out label
	LABEL label3 //out label
	LABEL label2 //out label

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

```
do #beginloop {S} while(B) #testloop ; #endloop
```

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

```
#beginloop
    create labels LOOP and OUT
    generate LABEL LOOP

#testloop
    Check if the conditional statement B
    has the correct type (Boolean)

#endloop
    Generate JUMP0 OUT
    Generate JUMP LOOP
```

repeat-until

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

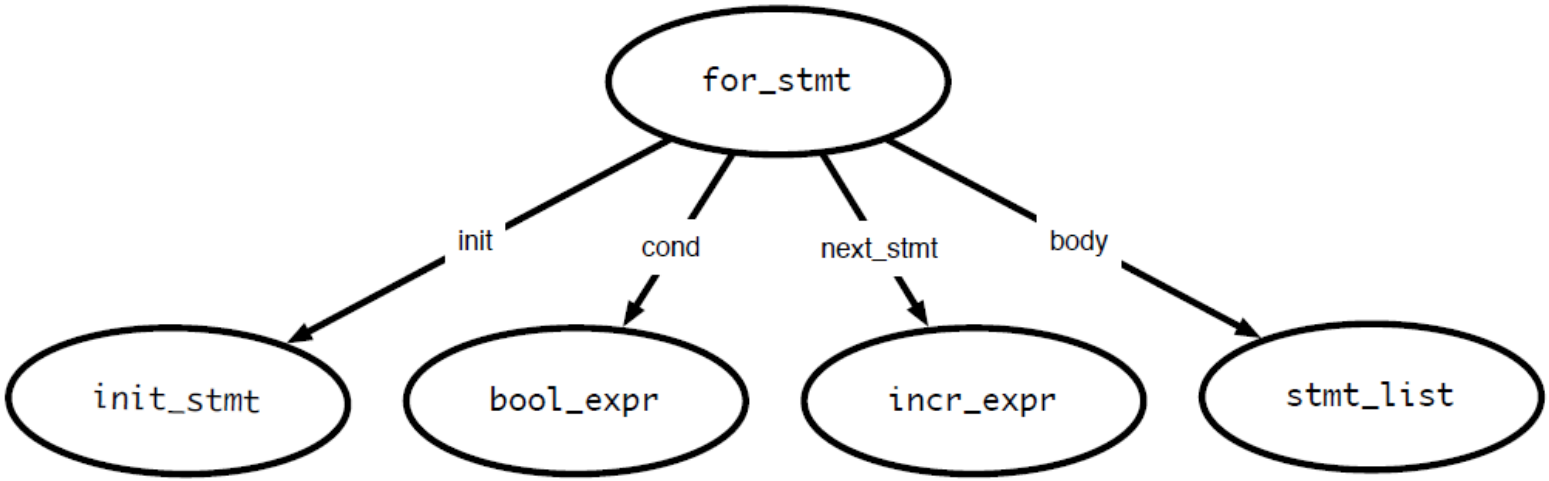
repeat-until

- `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

Slide courtesy: Milind Kulkarni

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

Suggested Reading

- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D.Ullman: Compilers: Principles, Techniques, and Tools, 2/E, AddisonWesley 2007
  - Chapter 2 (2.8), Chapter 6(6.2, 6.3, 6.4)
- Fisher and LeBlanc: Crafting a Compiler with C
  - Chapter 7 (7.1, 7.3), Chapter 11 (11.2)

Review/Practice

Which of the following is a valid string in the language specified by the CFG:

```
1. S -> aXa$
2. X -> λ
3.    | bY
4. Y -> λ
5.    | cXc
```

1. abcba
2. acca
3. aba
4. abcbcba

Hint: make higher-level reasoning with the help of grammar rules. E.g. rule 5 implies that ‘c’s appear in pairs. Rule 3 implies that a ‘c’ can appear only if ‘b’ is present. Rules 3 and 5 together imply that the last ‘b’ in any string must appear before a ‘c’.

Review/Practice

In bottom-up parsing, what is the sequence of derivations that you would get to match the input string: `-(id+id)+id`

The Grammar:

```
A -> B
   |B+A
B -> -B
   | id
   | (A)
```

```
-(id+id)+id
-(id+id)+B
-(id+id)+A
-(B+id)+A
-(B+B)+A
-(B+A)+A
-(A)+A
-B+A
B+A
A
```

- -(id+id)+id
- -(B+id)+id
- -(B+B)+id
- -(B+B)+B
- -(B+A)+B
- -(A)+B
- -B+B
- B+B
- B+A
- A

- -(id+id)+id
- -(B+id)+id
- -(B+B)+id
- -(B+A)+id
- -(A)+id
- -B+id
- B+id
- B+B
- B+A
- A

- -(id+id)+id
- -(id+B)+id
- -(id+A)+id
- -(B+A)+id
- -(A)+id
- -B+id
- B+id
- B+B
- B+A
- A

Hint: right-most derivation in reverse.

Review/Practice

In recursive-descent parsing, what is the sequence of derivations that you would have to try before matching the input string: `id+id`

The Grammar:

```
A -> B
   |B+A
B -> -B
   | id
   | (A)
```

- A
- B
- id
- B+A
- id+A
- id+B
- id+id

- A
- B+A
- id+A
- id+B
- id+id

- A
- B
- B+A
- id+A
- id+B
- id+id

- A
- B
- -B
- id
- (A)
- B+A
- -B+A
- id+A
- id+B
- id+-B
- id+id

Hint: try all productions in order shown in the grammar.

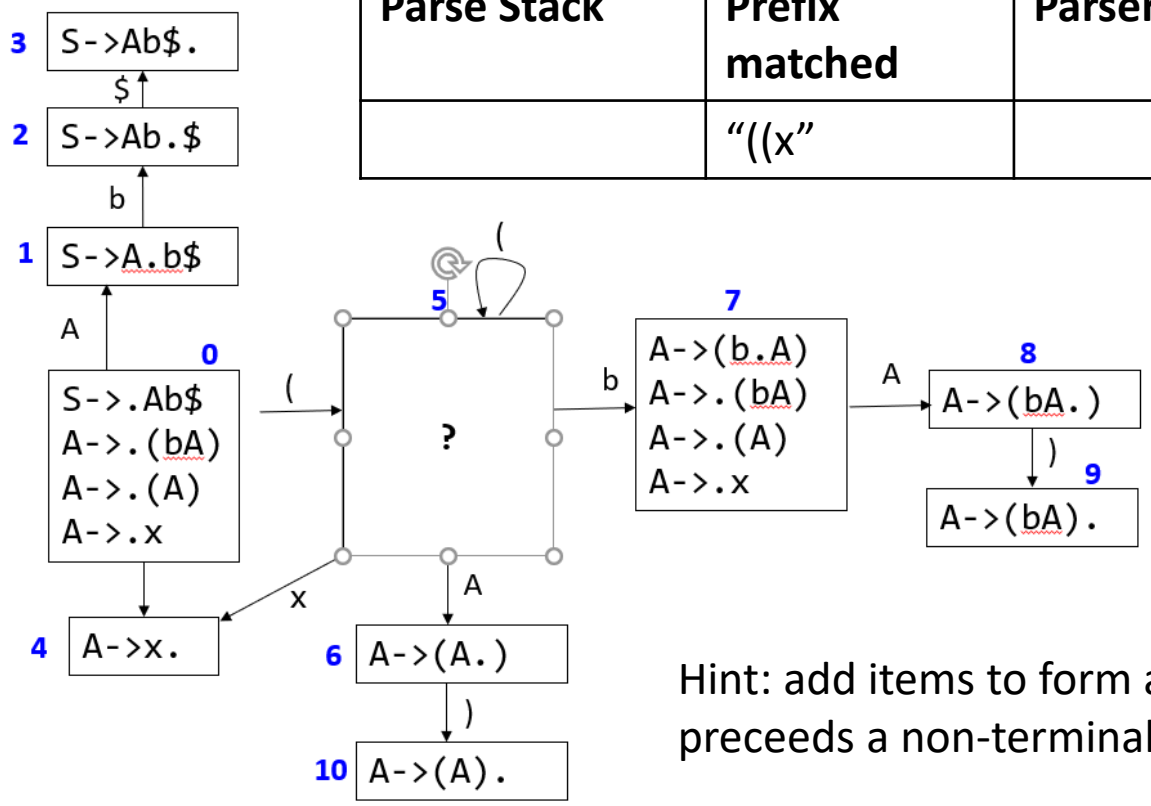
Review/Practice

The Grammar:

```
S -> Ab$
A -> (bA)
A -> (A)
A -> x
```

1. Complete the CFSM (fill state 5)
2. Fill the table and and add new entries if needed

Parse Stack	Prefix matched	Parser Action
	"((x"	



Hint: add items to form a closure if . preceeds a non-terminal.

Review/Practice

- Draw the AST for the expression and generate 3-address code `a := b + c * d + 1 ;`
  - assume bison declarations:

```
%left *
%left +
```

Hint: + has higher priority than * and both operators are left associative. So, the resulting expression is treated as: `a := ((b + c) * (d + 1)) ;`

Review/Practice

- Your language has a looping construct like C's **`do-while`** construct:

`do{`S_1`;…;`S_n`;}while(cond`$_1$`);` Statements $S_1 \ldots S_n$ are executed once before evaluating the condition `cond`$_1$. The statements are executed repeatedly till the condition `cond`$_1$ becomes `false`.

- Pascal has the **`repeat-until`** construct:

`repeat{`R_1`;…;`R_n`;}until(cond`$_2$`);` Statements $R_1 \ldots R_n$ are executed once before evaluating the condition `cond`$_2$. The statements are executed repeatedly till the condition `cond`$_2$, becomes `true`.

- Now, you want to *remove* the `do-while` feature in your language and *introduce* a **`repeat-while`** construct:

`repeat{`T_1`;…;`T_n`;}while(cond`$_3$`);` Statements $T_1 \ldots T_n$ are executed once before evaluating the condition `cond`$_3$. The statements are executed repeatedly till the condition `cond`$_3$ becomes `false`.

What phase(s) of the compiler you *must* change to implement the repeat-while construct? (explanation in support of your choices are welcome).

Assume keywords cannot be used as identifiers in your language

Hint: notice that meaning stays the same (for do-while and repeat-while). Only the keyword has changed.

Review/Practice

- Is the Grammar LL(1) ?

```
1.S-> A$
2.A-> xBC
3.A->CB
4.B->yB
5.B->λ
6.C->x
```

Hint: look at rules 2, 3, and 6. ..xB..->..xy.. is a prefix that can be derived using more than one way.