40-414 Compiler Design

Intermediate Code Generation

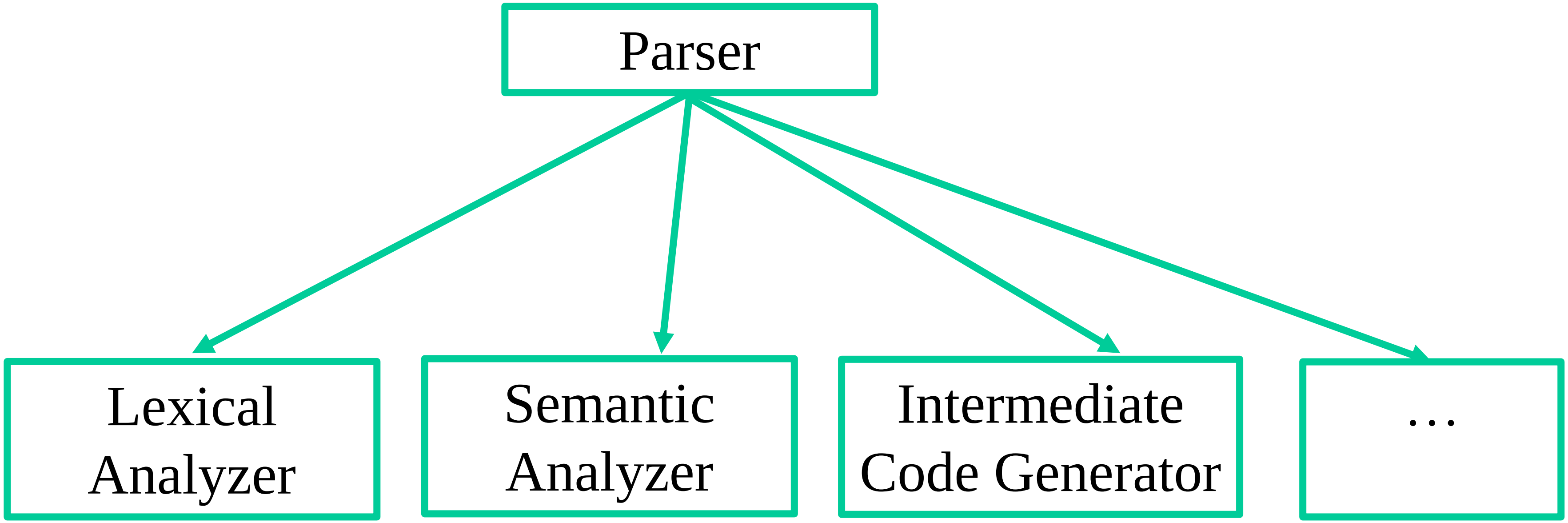
Lecture 8

Recommended Reading

P. Tremblay, J. Sorenson, *Theory and Practice of Compiler Writing*, McGraw Hill, Chapter 11.

(Not required if you attend this lecture and the next Exercise session)

Intermediate Code Generation



Intermediate Representation

- Translating source program into an "intermediate language."
  - Simple
  - CPU Independent,
  - …yet, close in spirit to machine language.

- Three Address Code (quadruples)

- Two Address Code, etc.

Three Address Codes

- Statements of general form x:=y op z
  here represented as (op, y, z, x)

- No built-up arithmetic expressions are allowed.

- As a result, x:=y + z * w should be represented as:
  t_1:=z * w
  t_2:=y + t_1
  x:=t_2

- Three-address code is useful: related to machine-language/ simple/ optimizable.

Types of Three-Address Statements

x:=y op z	(op, y, z, x)
x:=op z	(op, z, x,)
x:=z	(:=, z, x,)
goto L	(jp, L, ,)
if x relop y goto L	(relop, x, y, L), or (jpf, *A1*, *A2*,), (jpt, *A1*, *A2*,), etc.

Different Addressing Modes

(+, 100, 101, 102)

location 102 $\leftarrow$ content of 100 + content of 101

(+, #100, 101, 102)

location 102 $\leftarrow$ constant 100 + content of 101

(+, @100, 101, 102)

location 102 $\leftarrow$ content of content of 100 + content of 101

Definitions

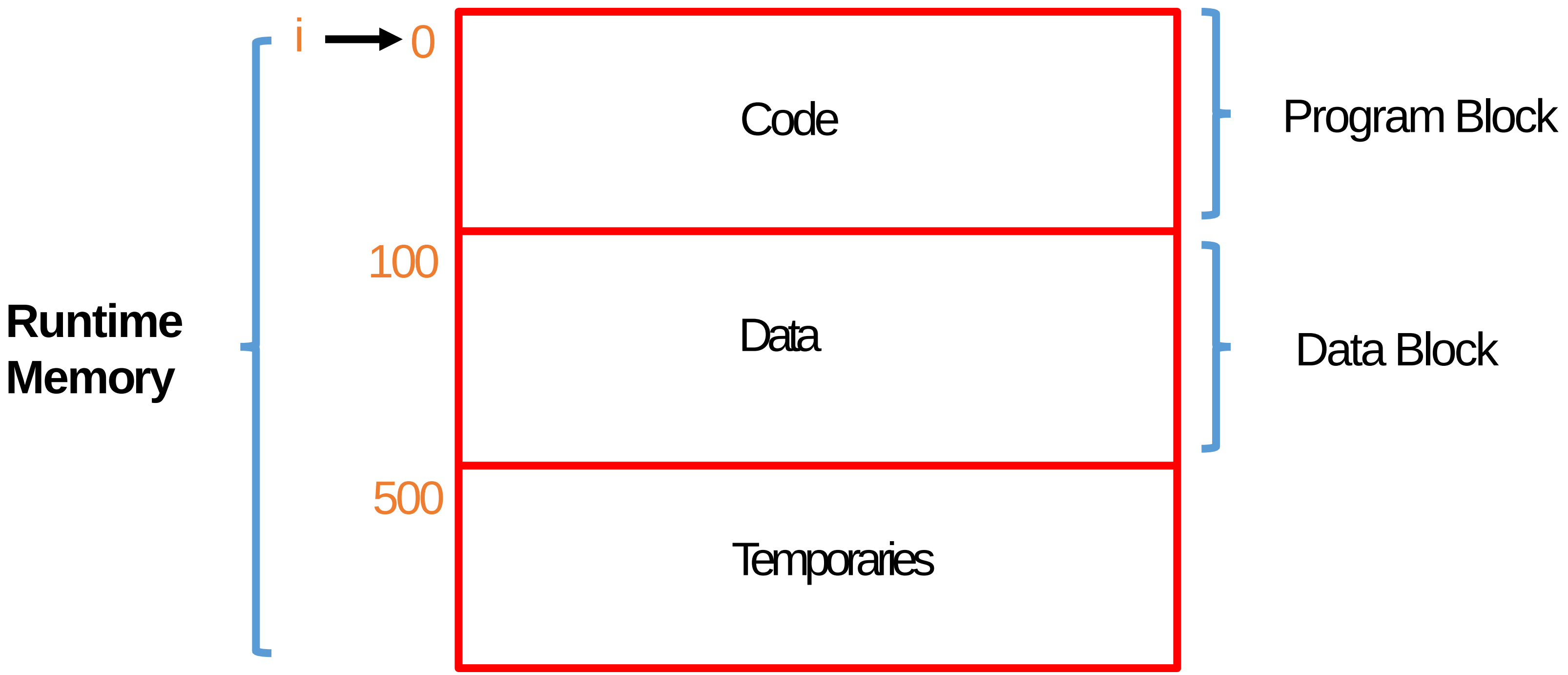
- Action Symbols (eg., #pid, #add, #mult, etc.): special symbols added to the grammar to signal the need for code generation

- Semantic Action (or, Semantic Routine): Each action symbol is associated with a sub-routine to perform

- Semantic Stack (here referred to by "ss"): a stack dedicated to the both semantic analyzer and intermediate code generator to store and use the required information

- Program Block (here referred to by "PB"): part of run time memory to be filled by the generated code

Run Time Memory Organization



Top-Down vs. Bottom-Up Generation

- Intermediate Code Generation can be performed in a top-down or bottom-up fashion (depending on the parsing direction)

- We first explain the Top-Down Approach

- Then we explain the minor differences that exist between the two approaches

Top-Down Intermediate Code Generation

PRODUCTION Rules with *action symbols*:

1. S → #pid id := E #assign
2. E → T E'
3. E' → ε
4. E' → + T #add E'
5. T → F T'
6. T' → ε
7. T' → * F #mult T'
8. F → (E)
9. F → #pid id

e.g., input: a := b + c * d

Code Generator Program

```
Proc codegen(Action)
    case (Action) of
        #pid : begin
                p ← findaddr(input);
                push(p)
        end
        #add | #mult : begin
                t ← gettemp
                PB[i] ← (+ | *, ss(top), ss(top-1), t);
                i ← i + 1; pop(2); push(t)
        end
        #assign : begin
                PB[i] ← (:=, ss(top), ss(top-1),);
                i ← i + 1; pop(2)
        end
    end
End codegen
```

- Function *gettemp* returns a new temporary variable that we can use.
- Function *findaddr(input)* looks up the current input's address from Symbol Table.

Example (LL1 implementation)

S → #pid id := E #assign
E → T E'
E' → ε | + T #add E'
T → F T'
T' → ε | * F #mult T'
F → (E)
F → #pid id

Parse Table

	id	+	*	(	)	:=	$
S	#pid id := E #assign						
E	T E'			T E'			
E'		+ T #add E'			ε		ε
T	F T'			F T'			
T'		ε	* F #mult T'		ε		ε
F	#pid id			(E)			

Example (LL1 implementation)

Parse Stack	Input	Operations
S $	id1 := id2 + id3 * id4 $	pop
#pid id := E #assign $	id1 := id2 + id3 * id4 $	codegen(#pid), pop
id := E #assign $	id1 := id2 + id3 * id4 $	pop, pop
T E’ #assign $	id2 + id3 * id4 $	Pop
F T’ E’ #assign $	id2 + id3 * id4 $	pop
#pid id T’ E’ #assign $	id2 + id3 * id4 $	codegen(#pid), pop
id T’ E’ #assign $	id2 + id3 * id4 $	pop
E’ #assign $	+ id3 * id4 $	pop
+ T #add E’ #assign $	+ id3 * id4 $	pop
F T’ #add E’ #assign $	id3 * id4 $	Pop
#pid id T’ #add E’ #assign $	id3 * id4 $	codegen(#pid), pop
id T’ #add E’ #assign $	id3 * id4 $	pop, pop
* F #mult T’ #add E’ #assign $	* id4 $	pop
#pid id #mult T’ #add E’ #assign $	id4 $	codegen(#pid), pop
id #mult T’ #add E’ #assign $	id4 $	pop
#mult T’ #add E’ #assign $	$	codegen(#mult), pop
T’ #add E’ #assign $	$	pop
#add E’ #assign $	$	codegen(#add), pop
E’ #assign $	$	pop
#assign $	$	codegen(#assign), pop
$	$	Finish!!

Example (Recursive Descent implementation)

S → #pid id := E #assign
E → T E'
E' → ε | + T #add E'
T → F T'
T' → ε | * F #mult T'
F → (E)
F → #pid id

Recursive Descent Parser Subroutines for S, E, E', T, T', and F :

procedure S ;
{ if *lookahead* = **id** then
{call codegen (#pid); call Match (**id**); call Match('**:=**');
call E; call codegen (#assign)}
else *error*
}

Example (Recursive Descent implementation)

$S \to$ #pid id := E #assign
$E \to$ T E'
$E' \to \varepsilon$ | + T #add E'
$T \to$ F T'
$T' \to \varepsilon$ | * F #mult T'
$F \to$ (E)
$F \to$ #pid id

```
procedure E ;
   { if lookahead ∈ { id, ( } then
        { call T; call E' }
     else  error
   }
```

```
procedure E' ;
   { if lookahead = '+' then
        { call Match( '+' ); call T; call codegen ( #add ); call E' }
     else  error
   }
```

Example (Recursive Descent implementation)

S → #pid id := E #assign
E → T E'
E' → ε | + T #add E'
T → F T'
T' → ε | * F #mult T'
F → (E)
F → #pid id

procedure T ;
{ if *lookahead* ∈ { **id, (** } then
{ call F; call T' }
else *error*
}

procedure T' ;
{ if *lookahead* = '*' then
{ call Match ('*'); call F; call codegen (#mult); call T' }
else *error*
}

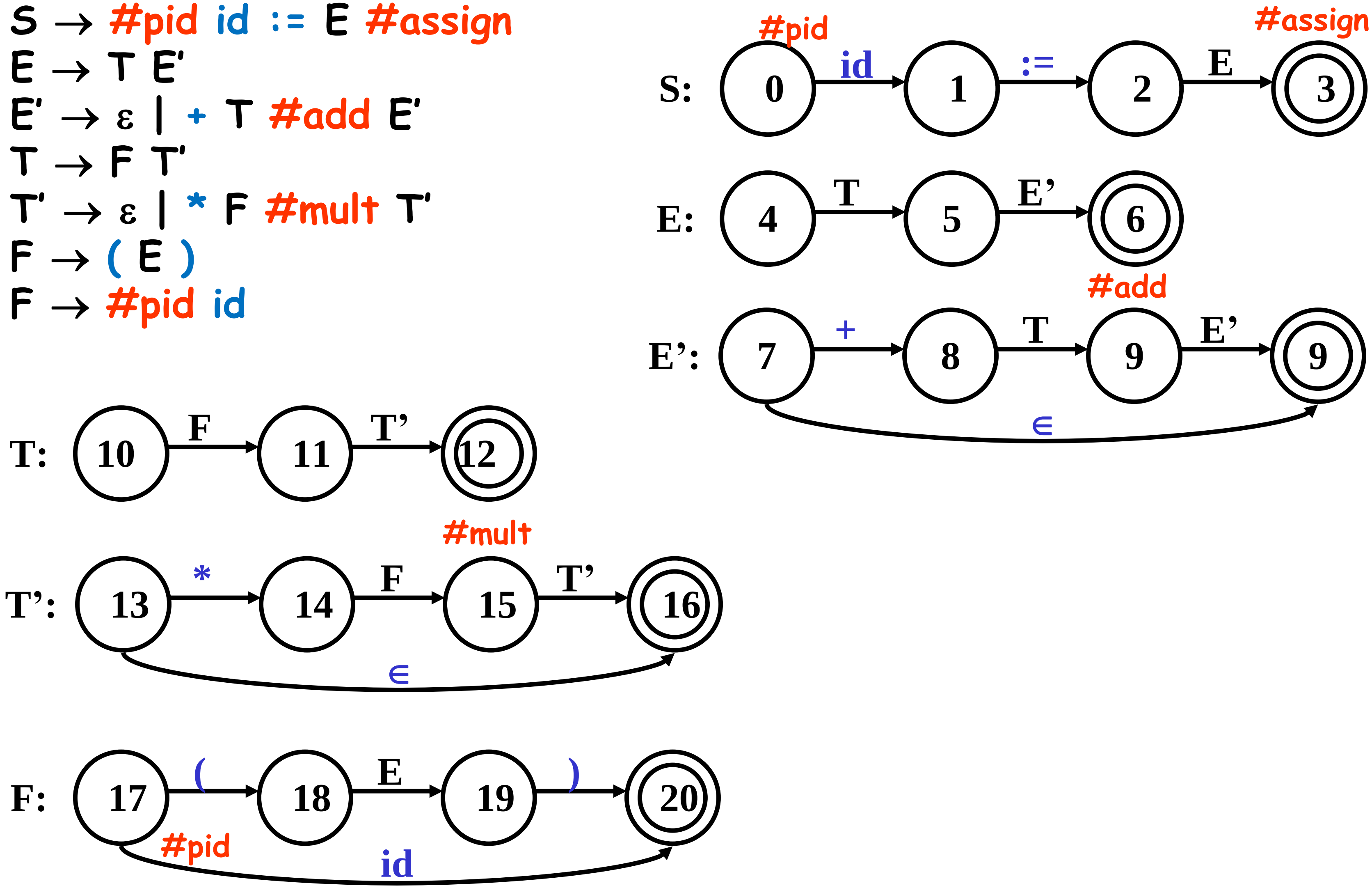
Example (Recursive Descent implementation)

S → #pid id := E #assign
E → T E'
E' → ε | + T #add E'
T → F T'
T' → ε | * F #mult T'
F → (E)
F → #pid id

procedure F ;
{ if *lookahead* = **id** then { call codegen (#pid); call Match (**id**) }
else if *lookahead* = '**(**' then { call Match ('**(**'); call E; call Match ('**)**') ; }
else *error*
}

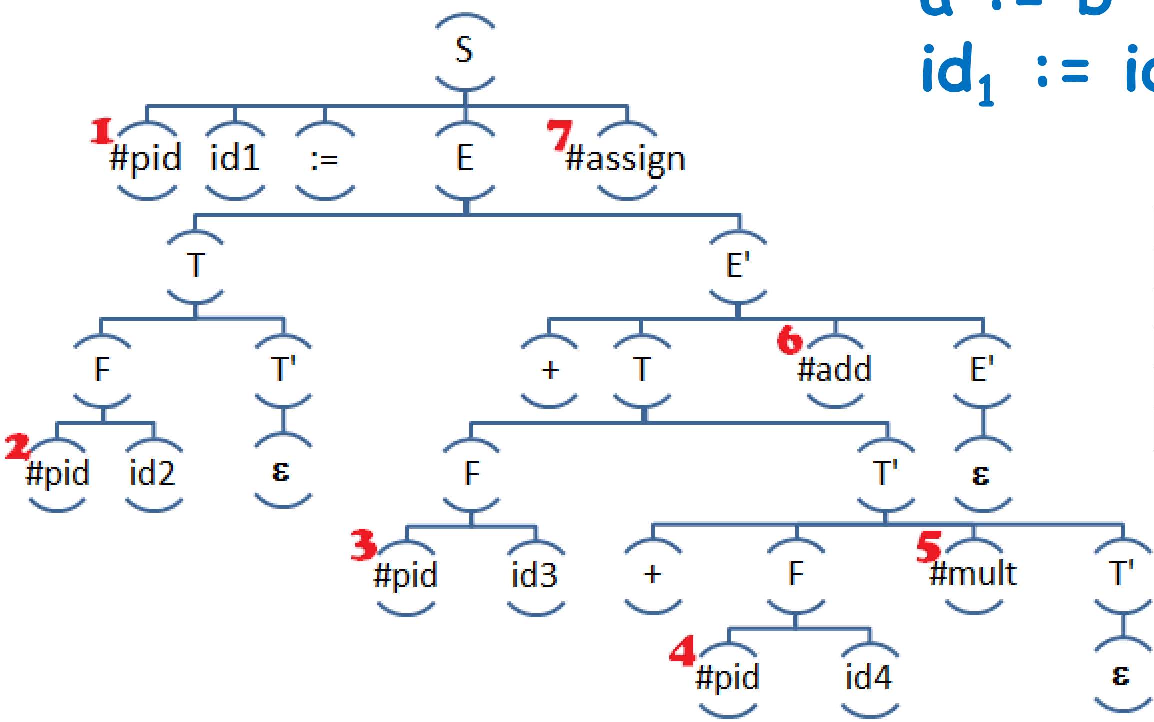
Example (Transition Diagrams implementation)

S → #pid id := E #assign
E → T E'
E' → ε | + T #add E'
T → F T'
T' → ε | * F #mult T'
F → (E)
F → #pid id



Example (Parse Tree)

- The order of Semantic Routines can be shown by the parse tree of the input sentence:



Input:

a := b + c * d

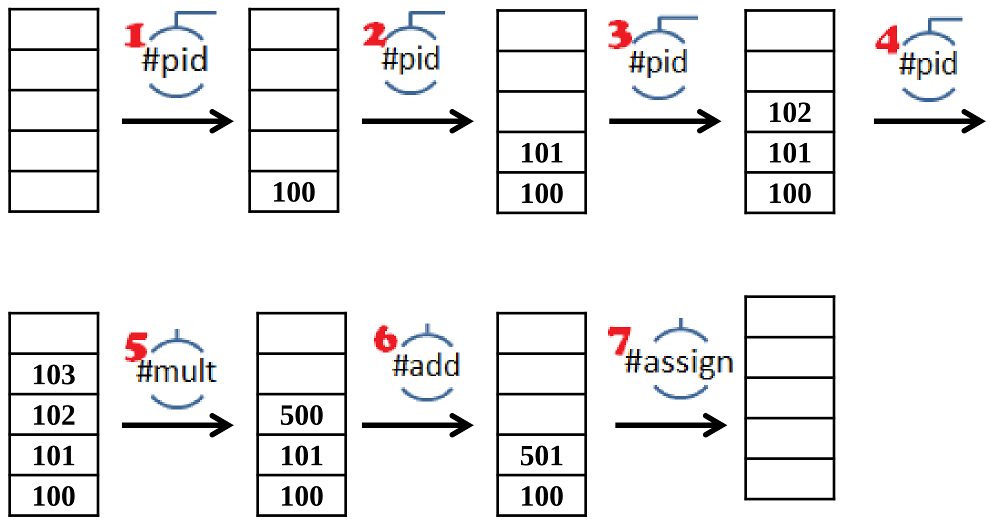
$id_1 := id_2 + id_3 * id_4$

no	lexeme	address
1	a	100
2	b	101
3	C	102
4	d	103

Symbol Table

Example (Semantic Stack and Program Block)

- Semantic Stack (SS) status during code generation:



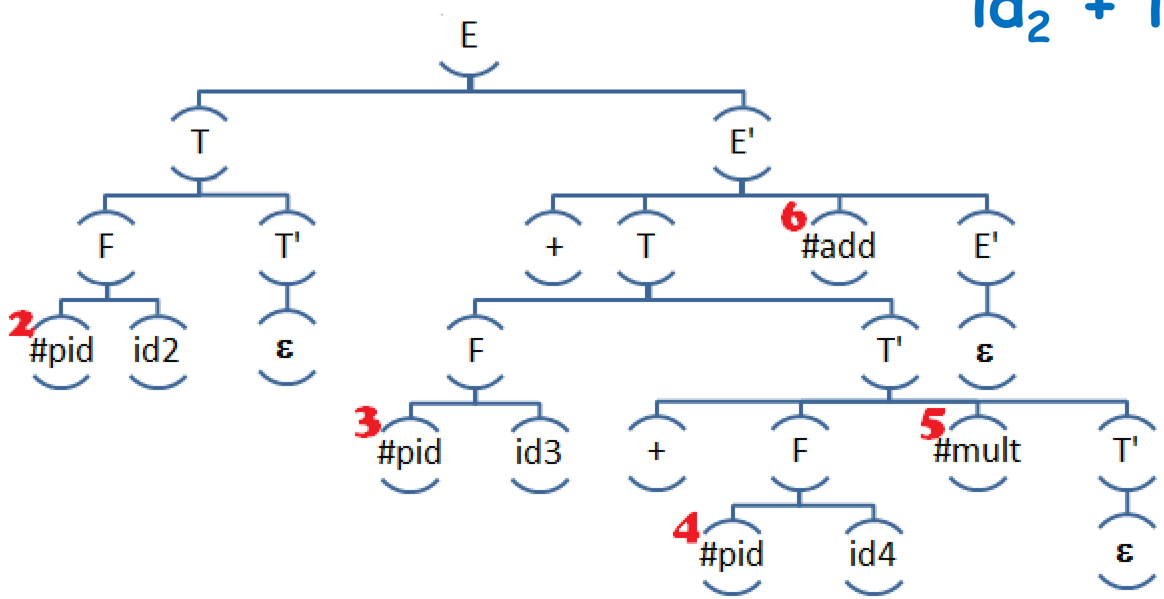
- Program Block (PB) :

i	PB[i]	Semantic Action called
0	(*,103,102,500)	#mult
1	(+,500,101,501)	#add
2	(=,501,100,)	#assign

Example 2 (Code of Expressions)

- Suppose we only had the right hand side expression

Input:
b + c * d
$id_2 + id_3 * id_4$



no	lexeme	address
1	a	100
2	b	101
3	C	102
4	d	103
Symbol Table		

Example 2 (Semantic Stack and Program Block)

- A temporary memory address will remain in SS:

i	PB[i]	Semantic Actions
0	(*,103, 102, 500)	#mult
1	(+, 500, 101, 501)	#add
2		

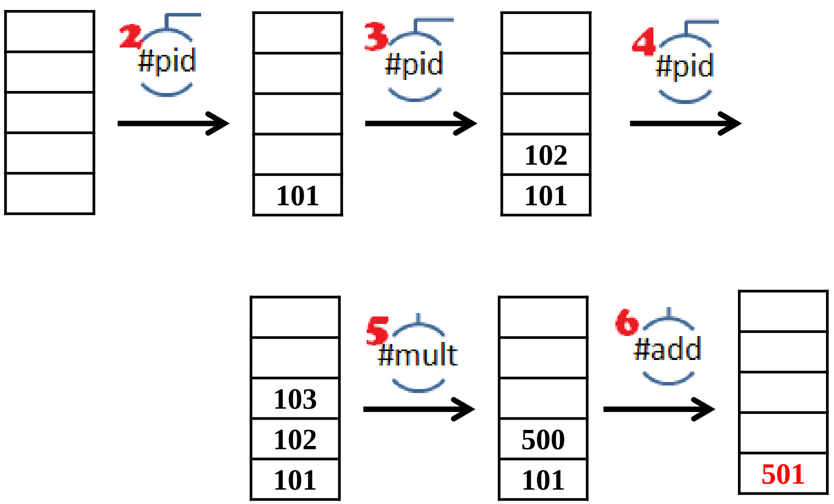
Input:

b + c * d

id_2 + id_3 * id_4

no	lexeme	address
1	a	100
2	b	101
3	C	102
4	d	103

Symbol Table



Control statements (while)

$S \rightarrow$ **while** E **do** S **end**

Input Example:

```
while    (b+c*d)    do
         a := a+1
end
```

- We need to find the appropriate place for inserting Semantic Actions symbols.

Control statements (while)

S → while E do S end

501

Input Example:

i	PB[i]	Semantic Actions
0	(*,103, 102, 500)	#mult
1	(+, 500, 101, 501)	#add
2		

```
while    (b+c*d)    do
         a := a+1
end
```

- After parsing E, the code for while's condition has been generated; and the allocated temporary memory address will be on top of the Semantic Stack.

Control statements (while)

S → while E do S end

501

Input Example:

i	PB[i]	Semantic Actions
0	(*,103, 102, 500)	#mult
1	(+, 500, 101, 501)	#add
2		

```
while    (b+c*d)    do
      a := a+1
end
```

- This is done by the semantic routines of non-terminal E

Control statements (while)

S → **while** E **do** S **end**

Input Example:

- We need a conditional jump, based on the result of while's condition, to outside of the loop.

```
while   (b+c*d) •  do
      a := a+1
end
•
```

Conditional jump

- But, we haven't yet compiled the loop body and thus don't know where the end of loop is!

Control statements (while)

#save

$S \rightarrow$ while E do S end

Input Example:

- Solution: BACKPATCHING!

#save: begin

push(i)

$i \leftarrow i + 1$

end

while (b+c*d) • do

a := a+1

end

•

Conditional jump

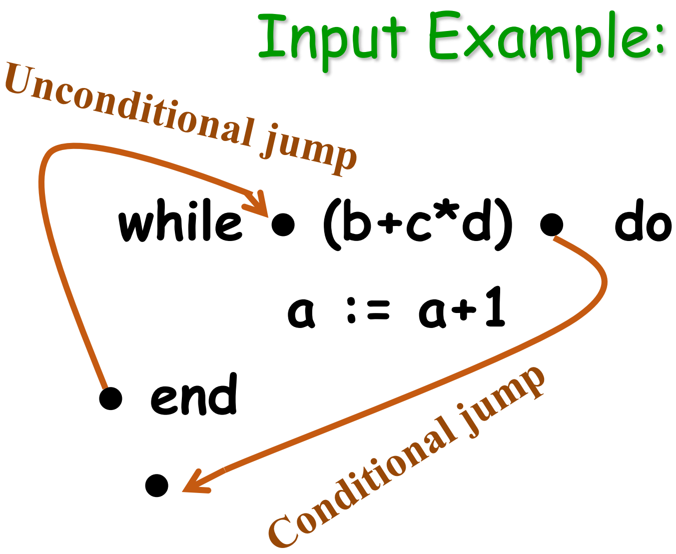
- Reserve a place in the program block for the jump, and generate the code at the end of the loop

Control statements (while)

#label

S → while E do #save S end

- We also need an unconditional jump back to while condition.
- Target of the unconditional jump must be saved in SS
- This is done by a semantic action; let's call it #label




```
#label: begin
            push(i)
        end
```

Control statements (while)

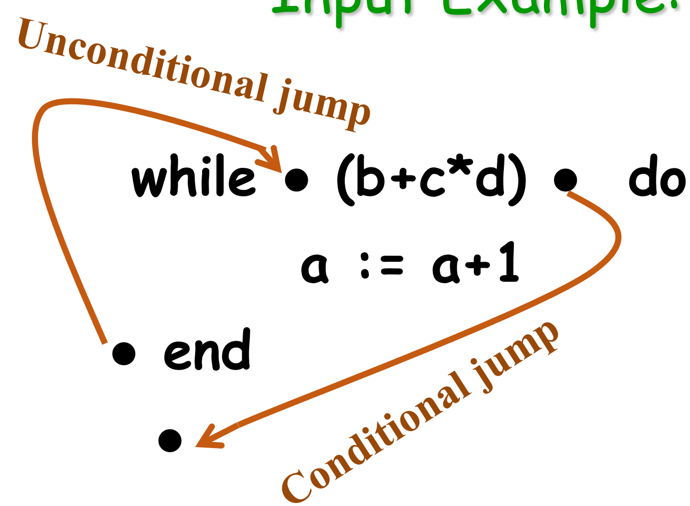
#while

S → while #label E do #save S end

At the end of while, the destination of conditional jump is known. So, the place saved by #save can be filled by #while.

An unconditional jump to the start of expression (saved by #label) is generated, too.

Input Example:



#while: begin

PB[ss(top)] ← (jpf, ss(top-1), i+1,);

PB[i] ← (jp, ss(top-2), ,);

i ← i + 1; Pop(3)

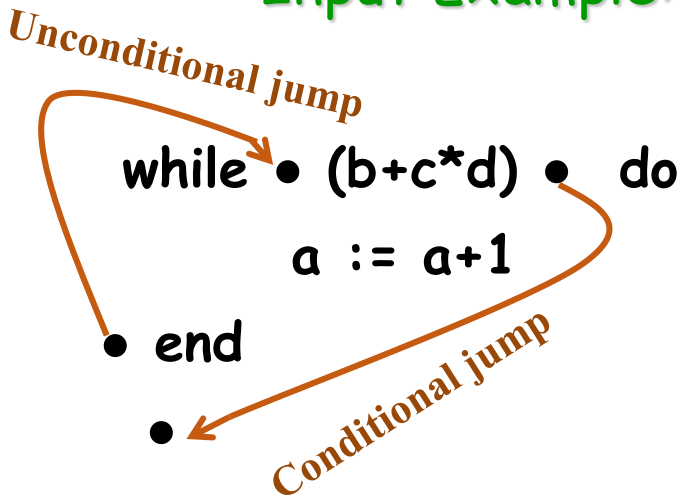
end

Control statements (while)

S → while #label E do #save S #while end

i	PB[i]	Semantic Actions
0		
1		
2		
3		
4		
5		
6		

Input Example:

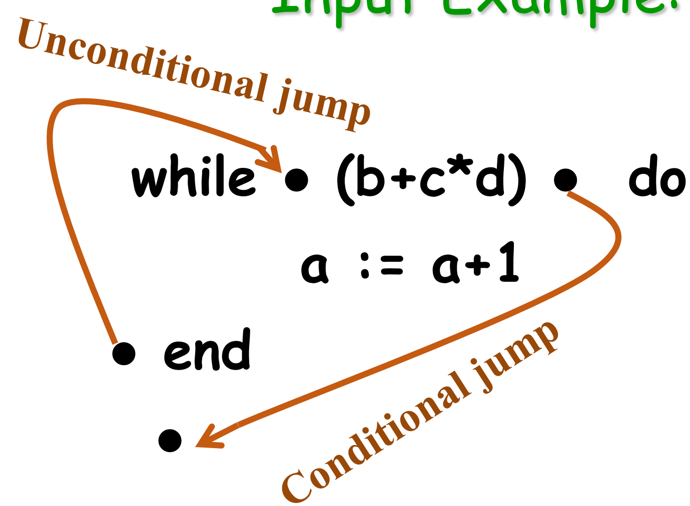


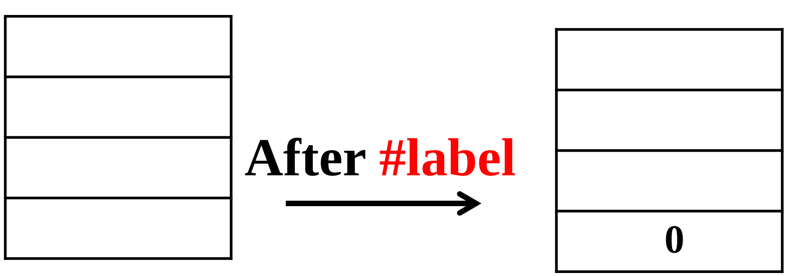
Control statements (while)

S → while #label E do #save S #while end

i	PB[i]	Semantic Actions
0		
1		
2		
3		
4		
5		
6		

Input Example:



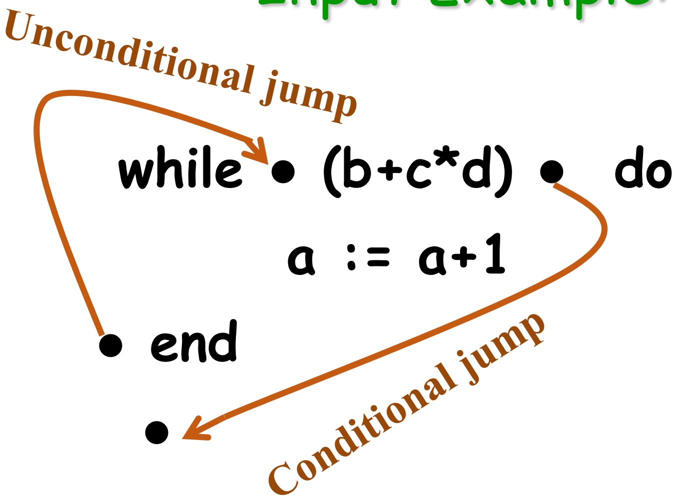


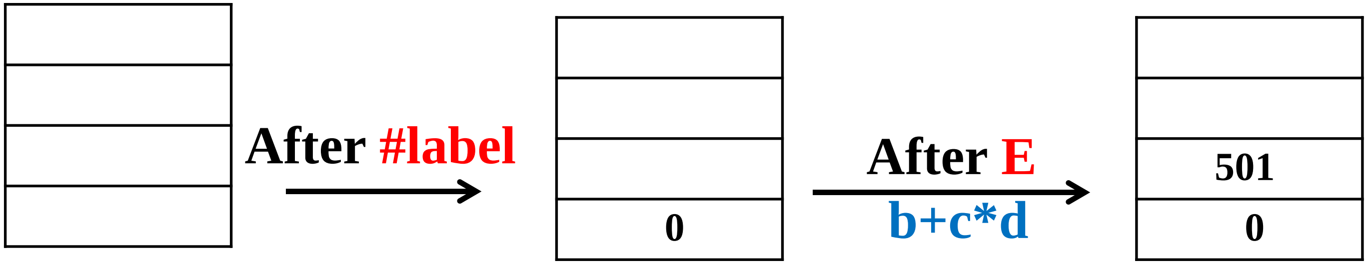
Control statements (while)

S → while #label E do #save S #while end

i	PB[i]	Semantic Actions
0	(*,103, 102, 500)	by E
1	(+, 500, 101, 501)	by E
2		
3		
4		
5		
6		

Input Example:



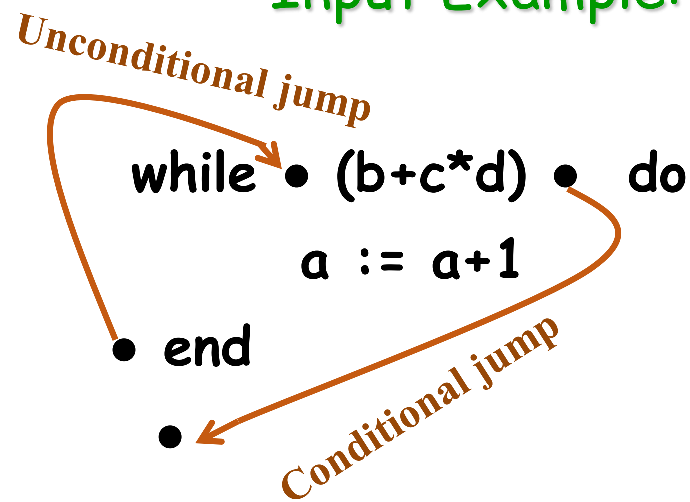


Control statements (while)

S → while #label E do #save S #while end

i	PB[i]	Semantic Actions
0	(*,103, 102, 500)	by E
1	(+, 500, 101, 501)	by E
2		
3		
4		
5		
6		

Input Example:



501
0

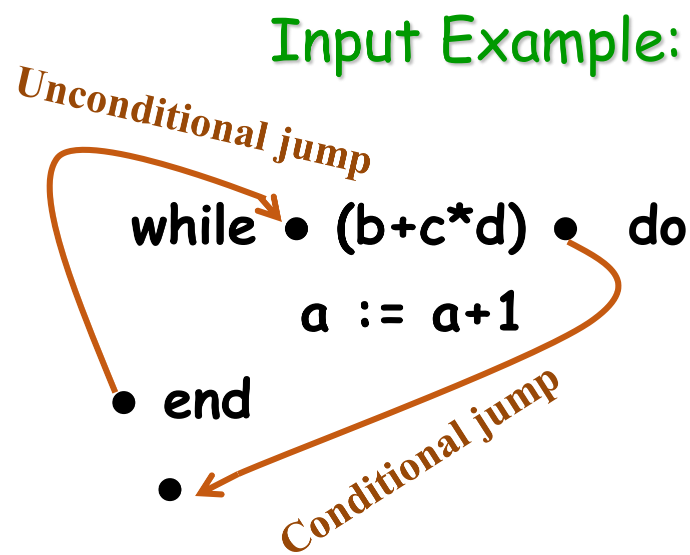
After #save →

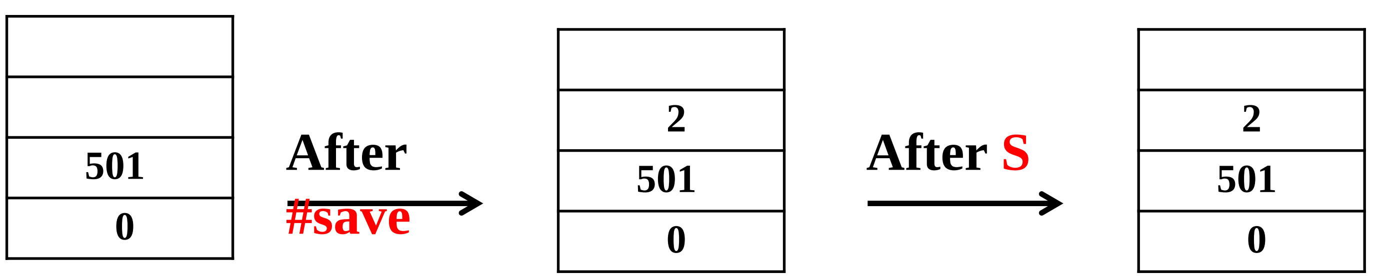
2
501
0

Control statements (while)

S → while #label E do #save S #while end

i	PB[i]	Semantic Actions
0	(*,103, 102, 500)	by E
1	(+, 500, 101, 501)	by E
2		
3	(+, 100, #1, 503)	by S
4	(=, 503, 100,)	by S
5		
6		



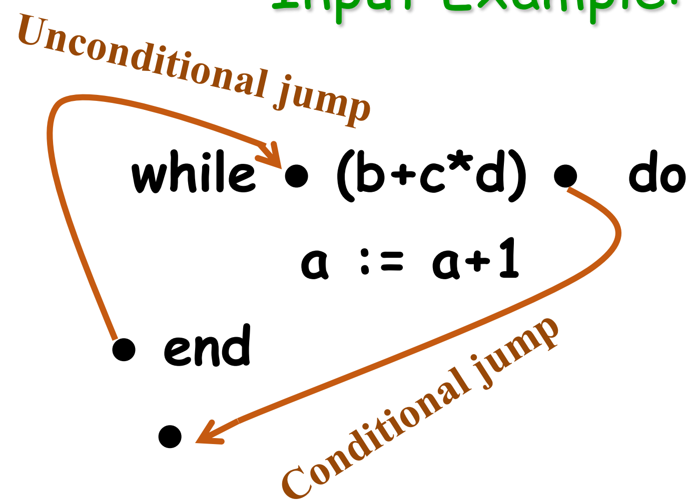


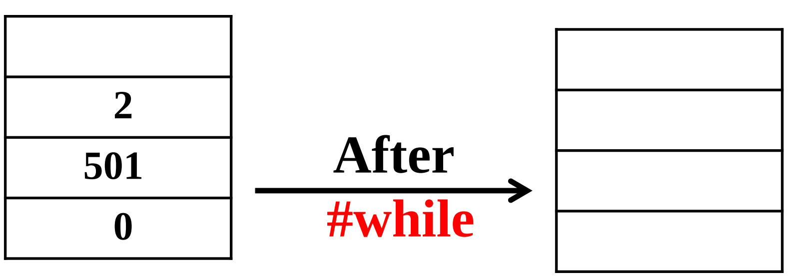
Control statements (while)

S → while #label E do #save S #while end

i	PB[i]	Semantic Actions
0	(*,103, 102, 500)	by E
1	(+, 500, 101, 501)	by E
2	(jpf, 501, 6,)	#while
3	(+, 100, #1, 503)	by S
4	(=, 503, 100,)	by S
5	(jp, 0, ,)	#while
6		

Input Example:





Conditional Statements (if-then-else)

$S \rightarrow$ **if** E **then** $S\ S'$
$S' \rightarrow$ **else** S
$S' \rightarrow \varepsilon$

Input Example:

If (a+b) then a := a+1

else b:= b+1

Conditional Statements (if-then-else)

$S \rightarrow$ if E #save then S S'
$S' \rightarrow$ else S
$S' \rightarrow \varepsilon$

Input Example:

If (a+b) then a := a+1

else b:= b+1

conditional jump **Unconditional jump**

Conditional jump: A place for jump should be saved by #save and to be later filled (by back patching).

#save: begin

push(i)

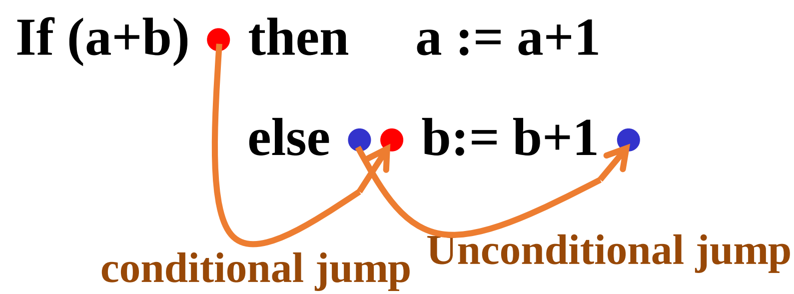
i ← i + 1

end

Conditional Statements (if-then-else)

S $\rightarrow$ if E #save then S S'
S' $\rightarrow$ else #jpf_save S
S' $\rightarrow \varepsilon$

Input Example:



When compiler reaches to else, the conditional jump can be generated by #jpf_save.

unconditional jump: A place for jump should be saved by #jpf_save and to be later filled (by *back patching*).

#jpf_save: begin

PB[ss(top)] ← (jpf,ss(top-1), i+1,)

Pop(2), push(i), i ← i + 1;

end

Conditional Statements (if-then-else)

S → if E #save then S S'
S' → else #jpf_save S #jp
S' → ε

When compiler is at the end of else statement, the unconditional jump can be generated by #jp.

Input Example:

If (a+b) then a := a+1

else b:= b+1

conditional jump Unconditional jump

```
#jp: begin
        PB[ss(top)] ← (jp, i,  ,  )
        Pop(1)
end
```

Conditional Statements (if-then-else)

$S \rightarrow$ **if** E **#save** **then** S S'
S' $\rightarrow$ **else** **#jpf_save** S **#jp**
S' $\rightarrow$ **#jpf**

Input Example:

If (a+b) • then a := a+1 •

conditional jump

If there isn't an else statement
(S' $\rightarrow \varepsilon$, is used),

only a conditional jump is generated by
#jpf.

Compare with
#jpf_save

#jpf: begin

PB[ss(top)] ← (jpf, ss(top-1) ,i ,)

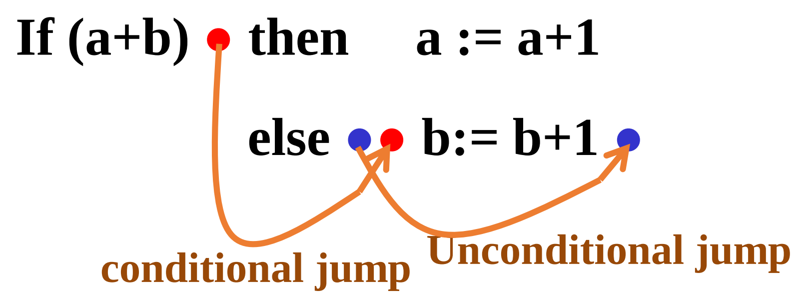
Pop(2)

end

Conditional Statements (if-then-else)

S → if E #save then S S'
S' → else #jpf_save S #jp
S' → #jpf

Input Example:



Program Block:

i	PB[i]	Semantic Actions
0	(+, a, b, t1)	#add
1	(jpf, t1, ?=5,)	#save
2	(+, a, #1, t2)	#add
3	(:=, t2, a,)	#assign
4	(jp, ?=7, ,)	#jpf_save
5	(+, b, #1, t3)	#add
6	(:=, t3, b,)	#assign
7		#jp

Goto Statements

$S \rightarrow$ goto id;
$S \rightarrow$ id: S

Difficulty: Unknown number forward goto statements.
Semantic Stack is not enough!

- Implemented by a linked list.
- Each node of linked list has:
  - Address of *goto* (in PB)
  - Label name
  - Label address (in PB)
  - Pointer to next node

Address of *goto*	**label**	**Address of label**	• →

Goto Statements (Cont.)

Example:

```
0  goto L4
1  L1: Statement1
2          goto L1;
3          goto L2;
4          goto L3;
5  L3: Statement 2
6          goto L1;
7          goto L3;
8          goto L2;
9  L2: Statement 3
10 L4: Statement 4
```

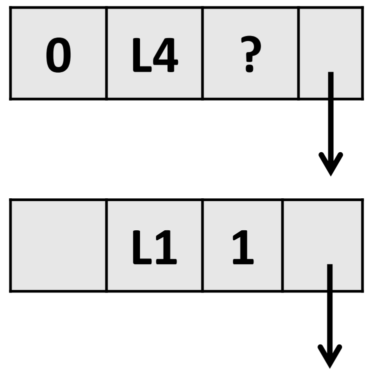
0	L4	?	

i	PB[i]
0	(jp, ?, ,)
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Goto Statements (Cont.)

Example:

```
0   goto L4
1   L1: Statement1
2         goto L1;
3         goto L2;
4         goto L3;
5   L3: Statement 2
6         goto L1;
7         goto L3;
8         goto L2;
9   L2: Statement 3
10  L4: Statement 4
```

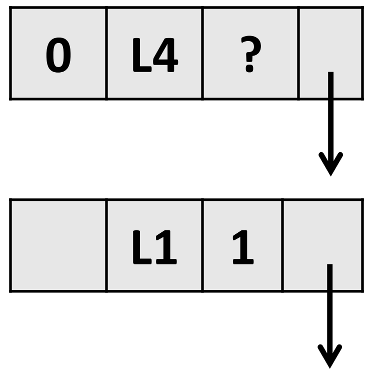


i	PB[i]
0	(jp, ?, ,)
1	statement 1
2	
3	
4	
5	
6	
7	
8	
9	
10	

Goto Statements (Cont.)

Example:

0 goto L4
1 L1: Statement1
2   goto L1;
3   goto L2;
4   goto L3;
5 L3: Statement 2
6   goto L1;
7   goto L3;
8   goto L2;
9 L2: Statement 3
10 L4: Statement 4

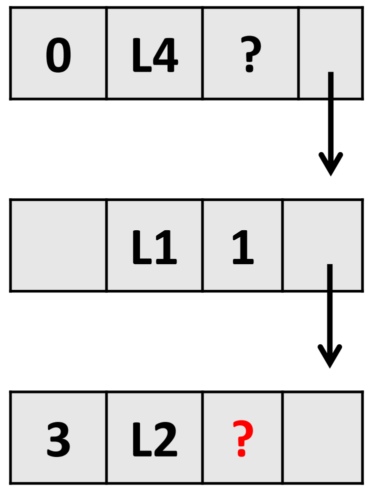


i	PB[i]
0	(jp, ?, ,)
1	statement 1
2	(jp, 1, ,)
3	
4	
5	
6	
7	
8	
9	
10	

Goto Statements (Cont.)

Example:

```
0   goto L4
1   L1: Statement1
2           goto L1;
3           goto L2;
4           goto L3;
5   L3: Statement 2
6           goto L1;
7           goto L3;
8           goto L2;
9   L2: Statement 3
10  L4: Statement 4
```

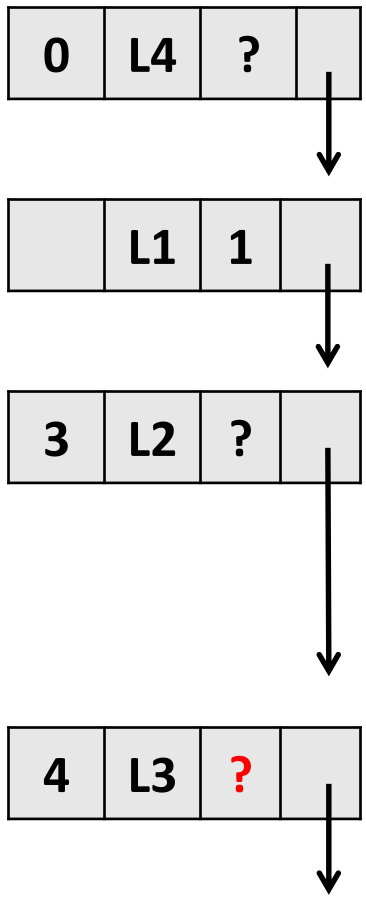


i	PB[i]
0	(jp, ?, ,)
1	statement 1
2	(jp, 1, ,)
3	(jp, ?, ,)
4	
5	
6	
7	
8	
9	
10	

Goto Statements (Cont.)

Example:

```
0   goto L4
1   L1: Statement1
2         goto L1;
3         goto L2;
4         goto L3;
5   L3: Statement 2
6         goto L1;
7         goto L3;
8         goto L2;
9   L2: Statement 3
10  L4: Statement 4
```

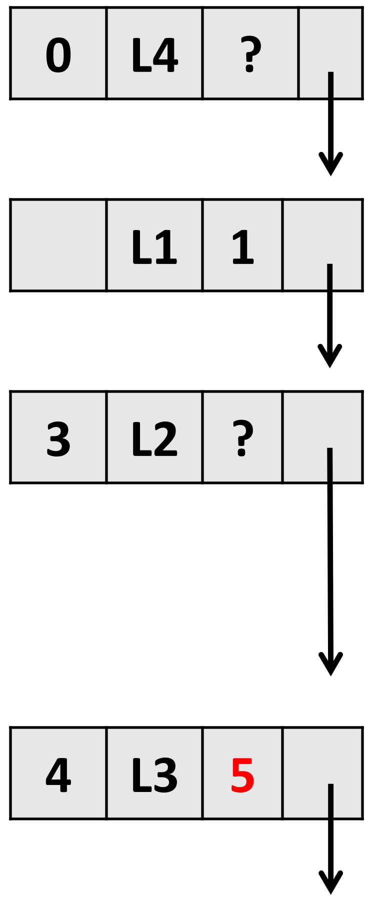


i	PB[i]
0	(jp, ?, ,)
1	statement 1
2	(jp, 1, ,)
3	(jp, ?, ,)
4	(jp, ?, ,)
5	
6	
7	
8	
9	
10	

Goto Statements (Cont.)

Example:

```
0   goto L4
1   L1: Statement1
2         goto L1;
3         goto L2;
4         goto L3;
5   L3: Statement 2
6         goto L1;
7         goto L3;
8         goto L2;
9   L2: Statement 3
10  L4: Statement 4
```

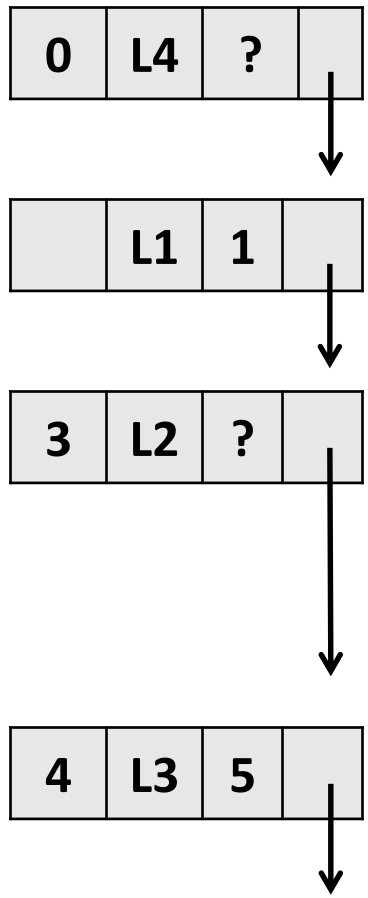


i	PB[i]
0	(jp, ?, ,)
1	statement 1
2	(jp, 1, ,)
3	(jp, ?, ,)
4	(jp, 5, ,)
5	statement 2
6	
7	
8	
9	
10	

Goto Statements (Cont.)

Example:

```
0   goto L4
1   L1: Statement1
2         goto L1;
3         goto L2;
4         goto L3;
5   L3: Statement 2
6         goto L1;
7         goto L3;
8         goto L2;
9   L2: Statement 3
10  L4: Statement 4
```

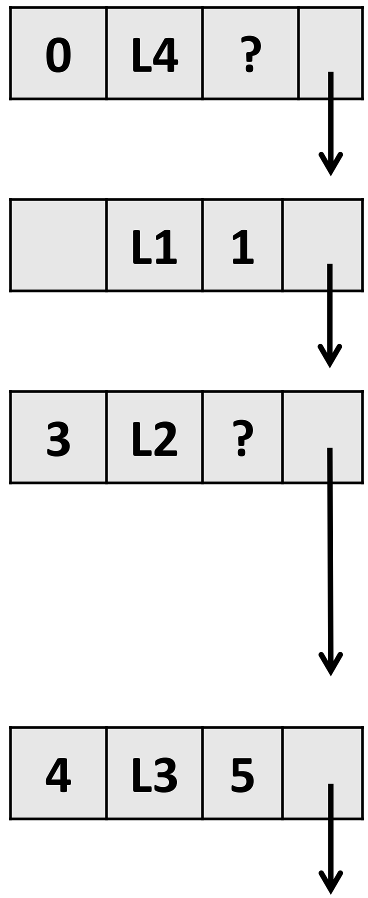


i	PB[i]
0	(jp, ?, ,)
1	statement 1
2	(jp, 1, ,)
3	(jp, ?, ,)
4	(jp, 5, ,)
5	statement 2
6	(jp, 1, ,)
7	
8	
9	
10	

Goto Statements (Cont.)

Example:

```
0   goto L4
1   L1: Statement1
2         goto L1;
3         goto L2;
4         goto L3;
5   L3: Statement 2
6         goto L1;
7         goto L3;
8         goto L2;
9   L2: Statement 3
10  L4: Statement 4
```

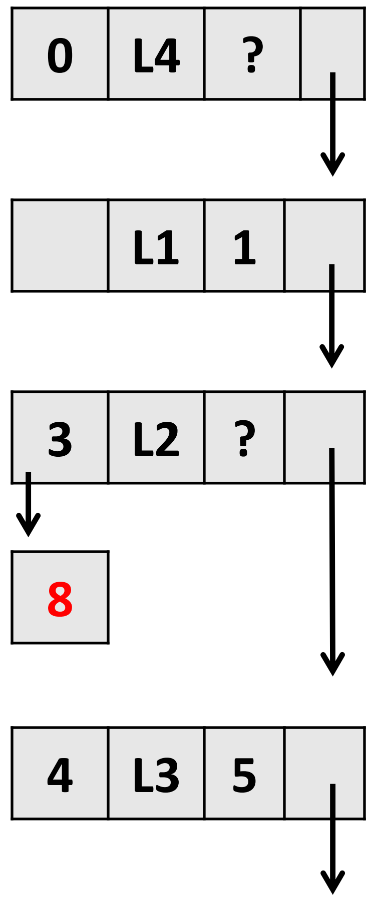


i	PB[i]
0	(jp, ?, ,)
1	statement 1
2	(jp, 1, ,)
3	(jp, ?, ,)
4	(jp, 5, ,)
5	statement 2
6	(jp, 1, ,)
7	(jp, 5, ,)
8	
9	
10	

Goto Statements (Cont.)

Example:

0 goto L4
1 L1: Statement1
2 goto L1;
3 goto L2;
4 goto L3;
5 L3: Statement 2
6 goto L1;
7 goto L3;
8 goto L2;
9 L2: Statement 3
10 L4: Statement 4

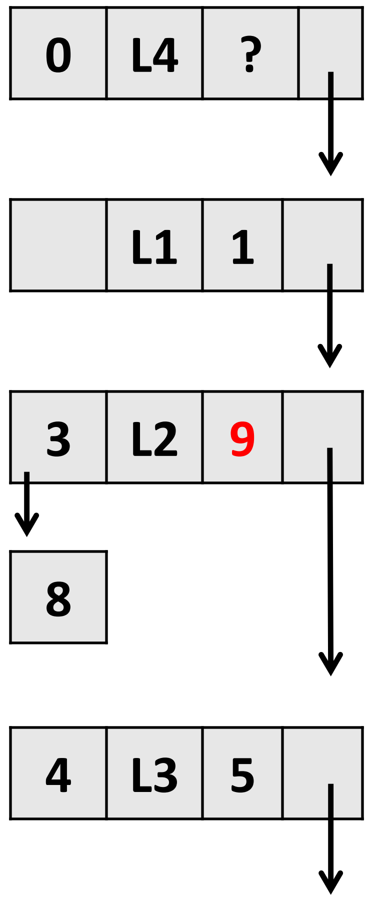


i	PB[i]
0	(jp, ?, ,)
1	statement 1
2	(jp, 1, ,)
3	(jp, ?, ,)
4	(jp, 5, ,)
5	statement 2
6	(jp, 1, ,)
7	(jp, 5, ,)
8	(jp, ?, ,)
9	
10	

Goto Statements (Cont.)

Example:

```
0   goto L4
1   L1: Statement1
2           goto L1;
3           goto L2;
4           goto L3;
5   L3: Statement 2
6           goto L1;
7           goto L3;
8           goto L2;
9   L2: Statement 3
10  L4: Statement 4
```

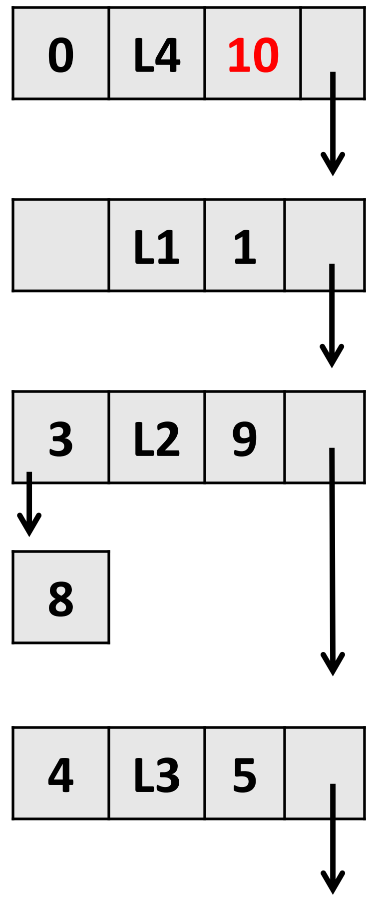


i	PB[i]
0	(jp, ?, ,)
1	statement 1
2	(jp, 1, ,)
3	(jp, 9, ,)
4	(jp, 5, ,)
5	statement 2
6	(jp, 1, ,)
7	(jp, 5, ,)
8	(jp, 9, ,)
9	statement 3
10	

Goto Statements (Cont.)

Example:

```
0   goto L4
1   L1: Statement1
2          goto L1;
3          goto L2;
4          goto L3;
5   L3: Statement 2
6          goto L1;
7          goto L3;
8          goto L2;
9   L2: Statement 3
10  L4: Statement 4
```



i	PB[i]
0	(jp, 10, ,)
1	statement 1
2	(jp, 1, ,)
3	(jp, 9, ,)
4	(jp, 5, ,)
5	statement 2
6	(jp, 1, ,)
7	(jp, 5, ,)
8	(jp, 9, ,)
9	statement 3
10	statement 4

Bottom-Up Code Generation

1. $S \rightarrow X$ id := E
2. $E \rightarrow T E'$
3. $E' \rightarrow \varepsilon$
4. $E' \rightarrow + T Y E'$
5. $T \rightarrow F T'$
6. $T' \rightarrow \varepsilon$
7. $T' \rightarrow * F Z T'$
8. $F \rightarrow (E)$
9. $F \rightarrow X$ id

$$\left.\begin{array}{ll} 10. & X \rightarrow \varepsilon \\ 11. & Y \rightarrow \varepsilon \\ 12. & Z \rightarrow \varepsilon \end{array}\right\} \text{new rules}$$

- *Intermediate code generator is called by the parser, after each reduction*
- Thus, only action symbols that are at the end of rules, are at a suitable positions
- Others action symbols must be replaced by new non-terminals producing only empty strings

Bottom-Up Code Generation

1. $S \rightarrow X$ id := E
2. $E \rightarrow T$ E'
3. E' $\rightarrow \varepsilon$
4. E' $\rightarrow$ + T Y E'
5. $T \rightarrow F$ T'
6. T' $\rightarrow \varepsilon$
7. T' $\rightarrow$ * F Z T'
8. $F \rightarrow$ (E)
9. $F \rightarrow X$ id
10. $X \rightarrow \varepsilon$
11. $Y \rightarrow \varepsilon$
12. $Z \rightarrow \varepsilon$

new rules

Instead of action symbols, rule numbers are passed by the parser to the code generator

```
Proc codegen(Action)
    case (Action) of
        10 : begin
                p ← findaddr(input);
                push(p)
        end
        11|12 : begin
                t ← gettemp
                PB[i] ← (+ | *, ss(top), ss(top-1), t);
                i ← i + 1; pop(2); push(t)
        end
      1: begin
        PB[i] ← (:=, ss(top), ss(top-1),);
        i ← i + 1; pop(2)
      end
    end
End codegen
```

Question?

$S \rightarrow$ **repeat** S **until** E **end**

Input Example:

- The above grammar defines repeat-until loops, where the loop body is executed at least once; we exit loop when its condition is true.
- Add the required action symbols and write the required semantic routines for such loops. Generate three address codes of the given example.

repeat

a := a-1

b := b+1

until (a-b) end

Question?

- The following grammar defines syntax of **for** loops.
- Add the required action symbols and write the required semantic routines for such loops. Generate three address codes of the given example.

Input Example:

```
for j := b+c  to  a*b   by  c*a  do
        d := d+j
end
```

$S \rightarrow$ **for id :=** E_1 **to** E_2 A **do** S **end**
$A \rightarrow$ **by** E_3
$A \rightarrow \varepsilon$

b+c : loop variable (j) initial value
a*b : loop variable (j) limit (constant)
c*d : loop variable (j) step (constant)

Question?

- The following grammar defines syntax of case statements, where at most one of case statements is to be executed.
- Can we generate intermediate code for these statements by just using a sematic stack to store the addresses that are required for back-patching?

Example:

```
case (c * d) of
        a: a := a + 1;
        b: b := b + 2;
        c: c := c + 3;
        otherwise: e := c*d
end
```

S → **case** E **of** L **end**
L → **id:** S B
B → ε | **otherwise** S | **; is :** S B