Intermediate Code Generation

- 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.
- Intermediate Code Generation can be performed in a top-down or bottom-up fashion (depending on the parsing method that the compiler employs)

Three Address Code

- Statements of general form $x:=y \text{ op } z$
  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) : put the result of adding contents of 100 and 101 into 102
  - (+, #100, 101, 102) : put the result of adding constant 100 and content of 101 into 102
  - (+, @100, 101, 102) : put the result of adding content of content of 100 and content of 101 into 102

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 intermediate code generator to store the required information
- Program Block (here referred to by “PB”): part of run time memory to be filled by the generated code

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. **a := b + c * d**

Code Generator

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

Example

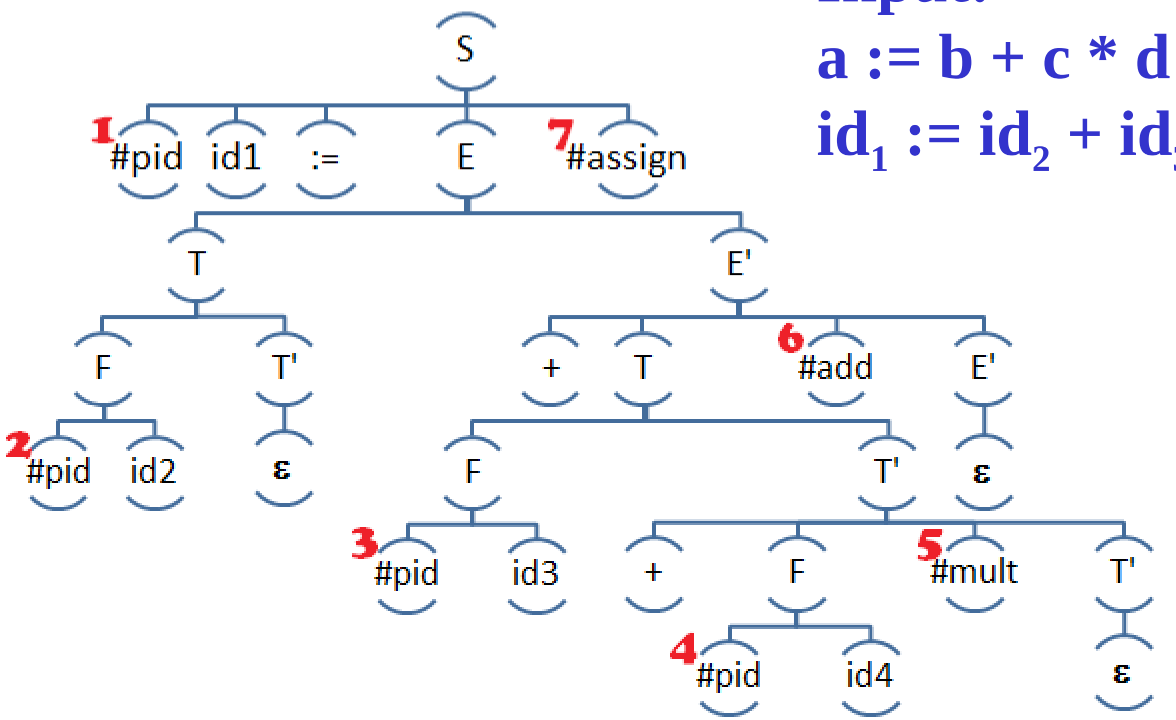
S → #pid id := E #assign
E → T E’
E’ → ε | + T #add E’
T → F T’
T’ → ε | * F #mult T’
F → (E)
F → #pid id

Parse Table

Non-terminal	INPUT SYMBOL						
	id	+	*	(	)	$	:=
E	E→TE’			E→TE’			
E’		E’→+TE’			E’→∈	E’→∈	
T	T→FT’			T→FT’			
T’		T’→∈	T’→*FT’		T’→∈	T’→∈	
F	F→id			F→(E)			
S	S → id := E						

Example (cont.)

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



Input:

a := b + c * d

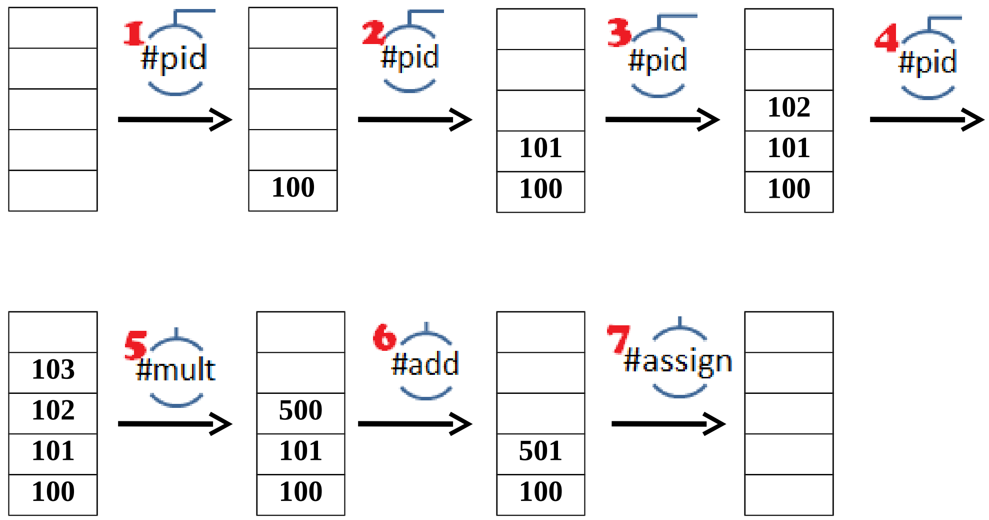
id_1 := id_2 + id_3 * id_4

Example – Parse Trace (cont.)

Parse Stack	Input	Operations
S $	id1 := id2 + id3 * id4 $	Pop
#pid id := E #assign $	id1 := id2 + id3 * id4 $	Call codegen(#pid), pop
id := E #assign $	id1 := id2 + id3 * id4 $	2 Matching ,2 pop
T E’ #assign $	id2 + id3 * id4 $	Pop
F T’ E’ #assign $	id2 + id3 * id4 $	pop
#pid id T’ E’ #assign $	id2 + id3 * id4 $	Call codegen(#pid), pop
id T’ E’ #assign $	id2 + id3 * id4 $	Matching , pop
E’ #assign $	+ id3 * id4 $	pop
+ T #add E’ #assign $	+ id3 * id4 $	Matching , pop
F T’ #add E’ #assign $	id3 * id4 $	Pop
#pid id T’ #add E’ #assign $	id3 * id4 $	Call codegen(#pid), pop
id T’ #add E’ #assign $	id3 * id4 $	Matching, 2 pop
* F #mult T’ #add E’ #assign $	* id4 $	Matching, pop
#pid id #mult T’ #add E’ #assign $	id4 $	Call codegen(#pid), pop
id #mult T’ #add E’ #assign $	id4 $	Matching, pop
#mult T’ #add E’ #assign $	$	Call codegen(#mult), pop
T’ #add E’ #assign $	$	pop
#add E’ #assign $	$	Call codegen(#add), pop
E’ #assign $	$	pop
#assign $	$	Call codegen(#assign), pop
$	$	Finish!!

Example (cont.)

- Semantic Stack (SS) :
  - Temporary variables range: [500, 501, …]
  - Data Block range :[100, 101, …, 499]



Example (cont.)

- Program Block (PB) :
  - Program Block range: [0, 1, …, 99]

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

Control statements (while)

10. **S → while E do S end**

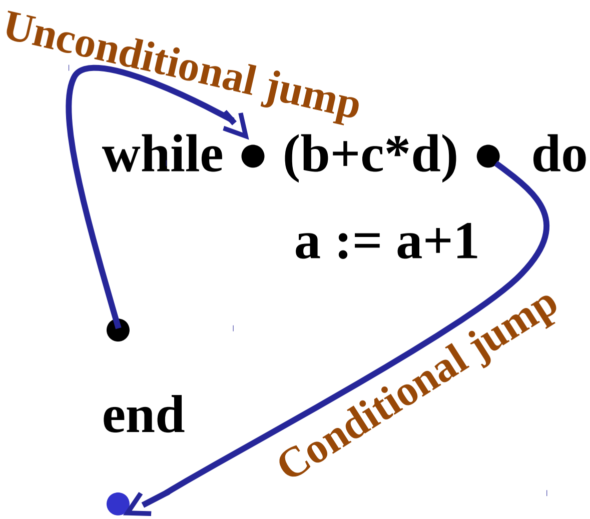
- **Semantic Actions places in grammer should be found.**

Input Example:

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

Control statements (while – cont.)

10. **S → while #label E do S end**



***Unconditional jump*:**

Destination of jump should be saved in **SS** by #label. (in order to generate the jump when compiler reaches to the end of loop)

```
#label:
begin

push(i)
end
```

Control statements (while – cont.)

10. **S → while #label E #save do S end**

Unconditional jump

while • (b+c*d) • do

a := a+1

•

end

Conditional jump

•

```
#save: begin
        push(i)
        i = i +1
end
```

Conditional jump: A place for jump should be saved by #save and later be filled (by *back patching*). That is because destination of jump is unknown when compiler has not yet seen the body of the loop.

Control statements (while – cont.)

10. **S → while #label E #save do S #while 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.

```
#while: begin
        PB[ss(top)] ← (jpf, ss(top-1), i+1,  );
        PB[i] ← (jp, ss(top-2),  ,  );
        i ← i + 1;
        Pop(3)
```

Control statements (while – cont.)

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

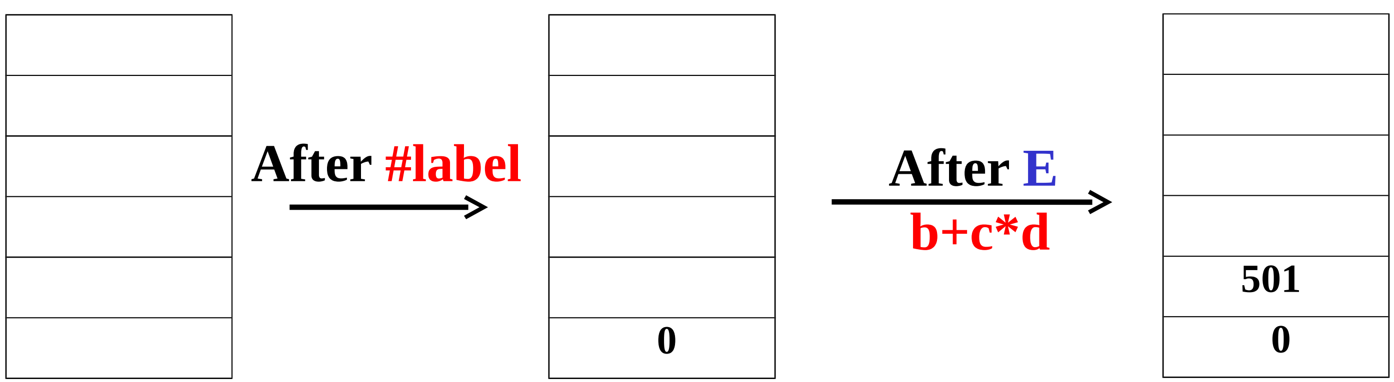
- Semantic Stack:

Input Example:

while (b+c*d) do

a := a+1

end



Control statements (while – cont.)

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

- Semantic Stack :

Input Example:

while (b+c*d) do

a := a+1

end

After #save →

2
501
0

After #while →

Control statements (while – cont.)

10. **S → while #label E #save do S #while end**

- **Program Block:**

i	PB[i]	Description
0	(*,103,102,500)	**#mult**, t1 ← c*d (by E)
1	(+,500,101,501)	**#add**, t2 ← b+t1 (by E)
2	(jpf,501, ?=6,)	**#save**, push(2) to **SS**; and leave i=2 empty.
3	(+,100,#1,503)	**#add**, t3 ← a+1 (by S)
4	(=,503,100,)	**#assign**, a ← t3 (by S)
5	(jp,0, ,)	**#while**, fill **PB[2]**; and jump to start of E.
6		

Input Example:

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

Control statements (while – cont.)

10. **S → while #label E #save do S #while end**

- **All Semantic Actions:**

#label : begin
push(i)
end
#save : begin
push(i);
i ← i + 1
end
#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 (repeat-until)

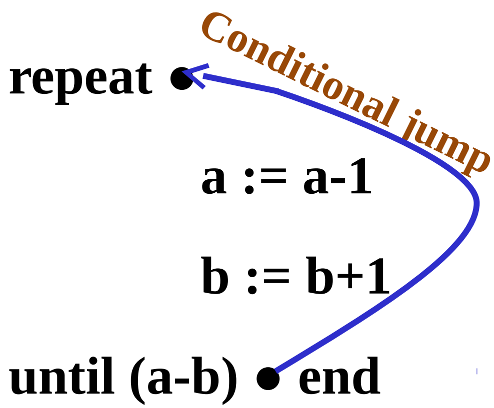
11. **S → repeat S until E end**

Input Example:

```
repeat
        a := a-1
        b := b+1
until (a - b) end
```

Control statements (repeat-until-cont.)

11. S → **repeat #label** S **until** E **end**



- ***conditional jump***: Destination of jump should be saved in **SS** by #label. (to be used when compiler reaches to the end of loop)

Control statements (repeat-until-cont.)

11. S → **repeat #label S until E #until end**

repeat •
Conditional jump
a := a-1
b := b+1
until (a-b) • end

- At the end of **repeat-until**, a conditional jump to the start of loop's body (saved by #label) is generated by #until. (No need to *Back Patching*)

```
#until: begin
        PB[i] ← (jpf, ss(top-1),  );
        i ← i +1;
        pop(2)
end
```

Control statements (repeat-until-cont.)

11. **S → repeat #label S until E #until end**

- **Program Block:**

i	PB[i]	Descriptions
0	(+, a, #1, t1)	**#add**
1	(:=, t1, a,)	**#assign**
2	(-, b, #1, t2)	**#add**
3	(:=, t2, b,)	**#assign**
4	(-, a, b, t3)	**#sub**
5	(jpf, t3, 0,)	**#until**
6		

Input Example:

repeat

a := a-1

b := b+1

until (a - b) end

Conditional Statements (if)

12. **S → if E then S S’**
13. **S’ → else S**
14. **S’ → ε**

Input Example:

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

else b:= b+1

Conditional Statements (if-cont.)

12. S → **if** E #save **then** S S'
13. S' → **else** S
14. S' → ε

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*).

Conditional Statements (if-cont.)

12. S → **if** E #save **then** S S'
13. S' → **else** #jpf_save S
14. S' → ε

If (a+b) • then a := a+1

else •• b:= b+1 •

conditional jump

Unconditional jump

#jpf_save: begin

PB[ss(top)] ← (jpf,ss(top-1), i+1,)

Pop(2), push(i), i ← i +1;

end

- 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*).

Conditional Statements (if-cont.)

12. S → **if** E #save **then** S S'
13. S' → **else** #jpf_save S #jp
14. S' → ε

If (a+b) • then a := a+1

else •• b:= b+1 •

conditional jump **Unconditional jump**

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

#jp: begin

PB[ss(top)] ← (jp, i, ,)

Pop(1)

Conditional Statements (if-cont.)

12. S → **if** E #save **then** S S’
13. S’ → **else** #jpf_save S #jp
14. S’ → #jpf

If (a+b) • then a := a+1 •

conditional jump

- If there isn’t an else statement (S’ → ε, is used), only a conditional jump is generated by #jpf.

```
#jpf: begin
        PB[ss(top)] ← (jpf, ss(top-1) ,i ,  )
        Pop(2)
end
```

Conditional Statements (if-cont.)

12. S → **if** E #save **then** S S'
13. S' → **else** #jpf_save S #jp
14. S' → #jpf

- **Program Block:**

Input Example:

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

else b:= b+1

i	PB[i]	Descriptions
0	(+, a, b, t1)	**#add**, push(t1) to **SS**;
1	(jpf, t1, ?=5,)	**#save**, push(1) to **SS**; and leave i=1 empty.
2	(+, a, #1, t2)	**#add**
3	(:=, t2, a,)	**#assign**
4	(jp, ?=7, ,)	**#jpf_save**, fill **PB[1]**; pop(2), push(4) to **SS** and leave i = 4 empty.
5	(+, b, #1, t3)	**#add**
6	(:=, t3, b,)	**#assign**
7		**#jp**, fill **PB[4]**.

Conditional Statements (if-cont.)

12. **S → if E #save then S S'**
13. **S' → else #jpf_save S #jp**
14. **S' → #jpf**

- **All Semantic Actions:**

#jpf_save : begin
PB[ss(top)] ← (jpf, ss(top-1), i + 1,);
pop(2); push(i); i ← i + 1
end
#jp : begin
PB[ss(top)] ← (jp, i, ,);
pop(1)
end
#jpf : begin
PB[ss(top)] ← (jpf, ss(top-1), i ,);
pop(2)
end

Control statements (for)

15. **S $\rightarrow$ for id := E_1 to E_2 STEP do S end**
16. **STEP $\rightarrow$ ε**
17. **STEP $\rightarrow$ by E_3**

Input Example:

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

b+c : loop variable (j) initial value

a*b : loop variable (j) limit (constant)

c*d : loop variable (j) step (constant)

Control statements (for-cont.)

15. $S \rightarrow$ **for #pid#pid id := E_1 #assign to E_2 STEP do S end**
16. **STEP** $\rightarrow \varepsilon$
17. **STEP** $\rightarrow$ **by** E_3

Unconditional jump

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

conditional jump

- **2** #pid put 2 copies of id's address in SS; one copy is used and poped by #assign. The second copy is later (after seeing E_2) used for comparison with limit of loop's variable.

Control statements (for-cont.)

15. S $\rightarrow$ **for #pid#pid id :=** E$_1$ **#assign to** E$_2$ **#cmp_save** STEP **do** S **end**
16. STEP $\rightarrow$ ε
17. STEP $\rightarrow$ **by** E$_3$

Unconditional jump

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

d := d+j

end

conditional jump

- After seeing E$_2$, loop's variable is compared with its limit and the result is saved in a temporary memory. In addition, a place for conditional jump is saved to be later used (by *back patching*).

```
#cmp_save: begin
        t ← gettemp
        PB[i] ← (<=, ss(top-1), ss(top) ,
t )
        i ← i+1 , pop(1), push(t), push(i),
```

Control statements (for-cont.)

15. $S \rightarrow$ **for #pid#pid id :=** E_1 **#assign to** E_2 **#cmp_save** STEP **do** S #for **end**
16. $STEP \rightarrow \varepsilon$
17. $STEP \rightarrow$ **by** E_3

Unconditional jump

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

conditional jump

- In the end of loop and by semantic routine #for:
  - Loop's variable should be increased by step,
  - An unconditional jump to the start loop is generated, and
  - The place saved by #cmp_save should be filled by a conditional jump

Control statements (for-cont.)

15. S → **for** **#pid#pid** **id :=** E_1 **#assign** **to** E_2 **#cmp_save** STEP **do** S **#for** **end**
16. STEP → ε
17. STEP → **by** E_3

Unconditional jump

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

conditional jump

#for: begin

PB[i] ← (+, ss(top), ss(top-3), ss(top-3));

i ← i+1;

PB[i] ← (jp, ss(top-1)-1, ,);

i←i+1;

PB[ss(top-1)] ← (jpf, ss(top-2), i,);

Control statements (for-cont.)

15. S → **for #pid#pid id :=** E_1 **#assign to** E_2 **#cmp_save** STEP **do** S **#for end**
16. STEP → **#step1**
17. STEP → **by** E_3

Unconditional jump

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

d := d+j

end

conditional jump

- If there is not an explicit step, (STEP → **ε** is used) , the step should be set to the default value of 1 (by #step1).

```
#step1: begin
        t← gettemp
        PB[i] ← (:=, #1,
t, )
        i← i+1, push(t)
```

Control statements (for-cont.)

15. S $\rightarrow$ for #pid#pid id := E_1 #assign to E_2 #cmp_save STEP do S #for end
16. STEP $\rightarrow$ #step1
17. STEP $\rightarrow$ by E_3

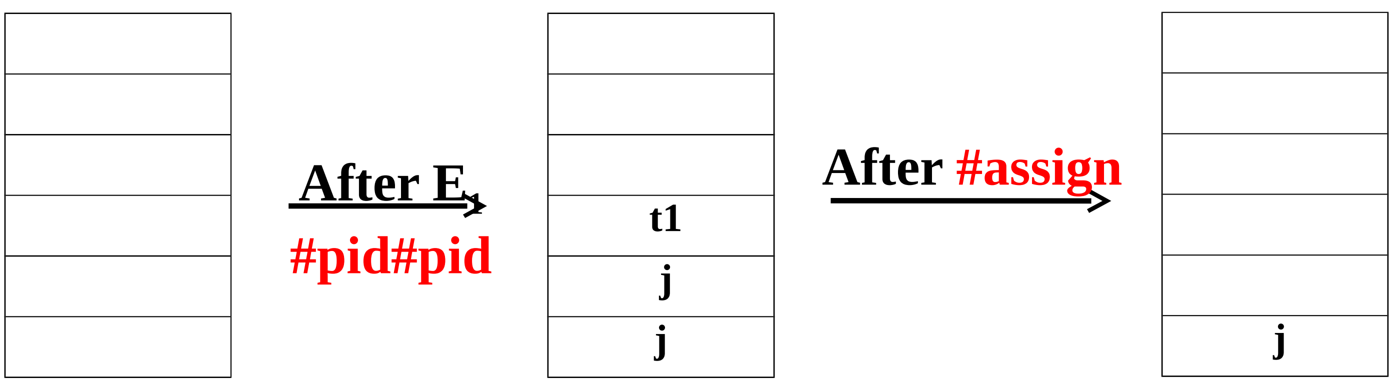
Input Example:

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

d := d+j

end

- **Semantic Stack :**



Control statements (for-cont.)

15. S $\rightarrow$ **for #pid#pid id := E_1 #assign to E_2 #cmp_save** STEP **do** S #for **end**
16. **STEP $\rightarrow$ #step1**
17. **STEP $\rightarrow$ by E_3**

Input Example:

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

Semantic Stack :

After E_2 $\longrightarrow$

t2
j

After #cmp_save $\longrightarrow$

4
t3
j

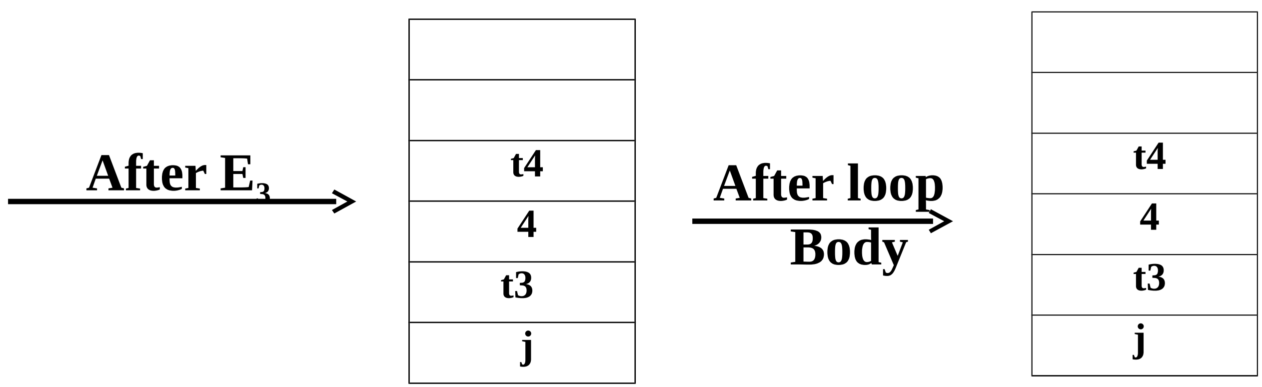
Control statements (for-cont.)

15. S $\rightarrow$ **for #pid#pid id :=** E$_1$ **#assign to** E$_2$ **#cmp_save** STEP **do** S **#for end**
16. STEP $\rightarrow$ **#step1**
17. STEP $\rightarrow$ **by** E$_3$

Input Example:

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

- **Semantic Stack :**



Control statements (for-cont.)

15. $S \rightarrow$ **for #pid#pid id :=** E_1 **#assign to** E_2 **#cmp_save** STEP **do** S **#for end**
16. STEP $\rightarrow$ **#step1**
17. STEP $\rightarrow$ **by** E_3

- **Program Block:**

i	PB[i]	Descriptions
0	(+, b, c, t1)	**#add** (by E_1)
1	(:=, t1, j,)	**#assign**: j is initialized.
2	(*, a, b, t2)	**#mult** (by E_2)
3	(<=, j, t2, t3)	**#cmp_save**: and push result, t3, to **SS**.
4	(jpf, t3, ?=10,)	**#cmp_save**, push(4) to **SS**; and leave i=4 empty.
5	(*, c, a, t4)	**#mult** (by E_3); and push(t4) to **SS**.
6	(+, d, j, t5)	**#add** (S, by body of loop)
7	(:=, t5, d,)	**#assign** (S, by body of loop)
8	(+ , t4, j, j)	**#for**: j = j + c*a (adding step)
9	(jp, 3, ,)	**#for**: unconditional jump to (4-1); and fill **PB[4]**
10		

Control statements (for-cont.)

15. **S → for #pid #pid id := E$_1$ #assign to E$_2$ #cmp_save** STEP **do** S **#for end**
16. **STEP → #step1**
17. **STEP → by E$_3$**

- **All Semantic Actions:**

```
#cmp_save : begin
t ← gettemp
PB[i] ← (<=, ss(top-1), ss(top), t);
i ← i + 1; pop(1); push(t); push(i); i ← i + 1
end
#for : begin
PB[i] ← (+, ss(top), ss(top-3), ss(top-3));
i ← i + 1;
PB[i] ← (jp, ss(top-1)-1, ,);
i ← i + 1;
PB[ss(top-1)] ← (jpf, ss(top-2), i,);
pop(4)
end
#step1 : begin
t ← gettemp
PB[i] ← (:=, #1, t, );
i ← i + 1; push(t)
end
```

goto statements

- 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

goto statements (cont.)

- Example:

goto L4
L1: Statement1
    goto L1;
    goto L2;
    goto L3;
L3: Statement 2
    goto L1;
    goto L3;
    goto L2;
L2: Statement 3
L4: Statement 4

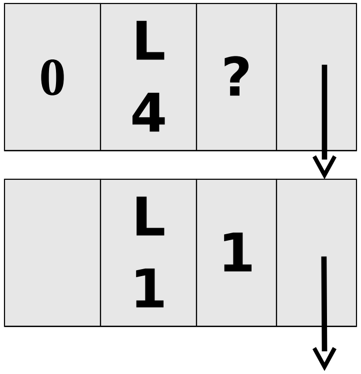
0	L4	?	

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

goto statements (cont.)

- Example:

goto L4
L1: Statement1
    goto L1;
    goto L2;
    goto L3;
L3: Statement 2
    goto L1;
    goto L3;
    goto L2;
L2: Statement 3
L4: Statement 4

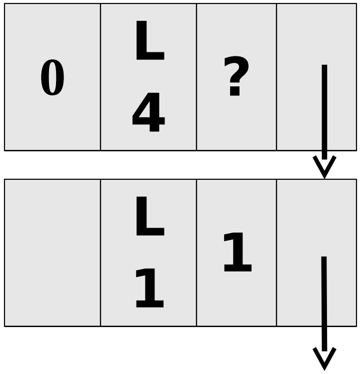


i	PB[i]
0	(jp, ?, ,)
1	statement 1
2	
3	
4	
5	
6	
7	
8	
9	
10	

goto statements (cont.)

- Example:

goto L4
L1: Statement1
    goto L1;
    goto L2;
    goto L3;
L3: Statement 2
    goto L1;
    goto L3;
    goto L2;
L2: Statement 3
L4: Statement 4

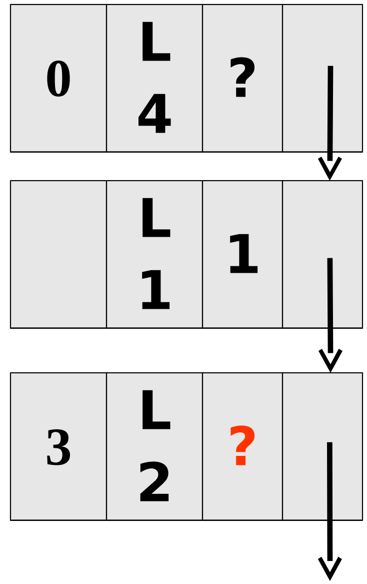


i	PB[i]
0	(jp ?, ,)
1	statement 1
2	(jp, 1, ,)
3	
4	
5	
6	
7	
8	
9	
10	

goto statements (cont.)

- Example:

goto L4
L1: Statement1
    goto L1;
    goto L2;
    goto L3;
L3: Statement 2
    goto L1;
    goto L3;
    goto L2;
L2: Statement 3
L4: Statement 4

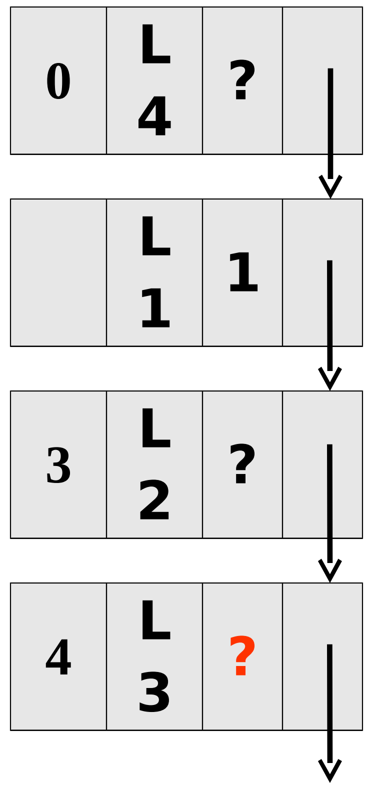


i	PB[i]
0	(jp, ?, ,)
1	statement 1
2	(jp, 1, ,)
3	(jp, ?, ,)
4	
5	
6	
7	
8	
9	
10	

goto statements (cont.)

- Example:

goto L4
L1: Statement1
    goto L1;
    goto L2;
    goto L3;
L3: Statement 2
    goto L1;
    goto L3;
    goto L2;
L2: Statement 3
L4: Statement 4

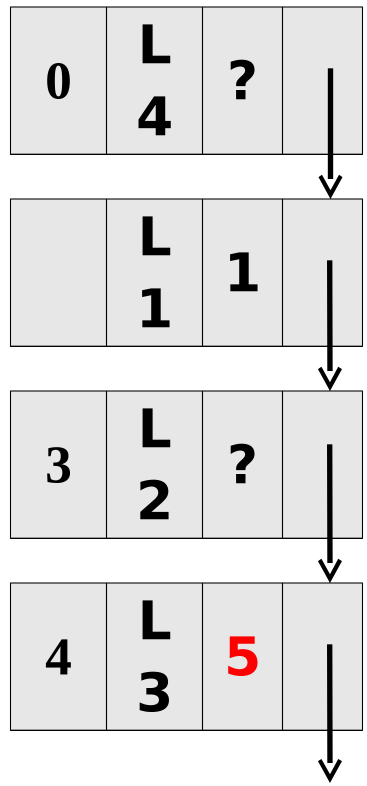


i	PB[i]
0	(jp, ?, ,)
1	statement 1
2	(jp, 1, ,)
3	(jp, ?, ,)
4	(jp, ?, ,)
5	
6	
7	
8	
9	
10	

goto statements (cont.)

- Example:

goto L4
L1: Statement1
    goto L1;
    goto L2;
    goto L3;
L3: Statement 2
    goto L1;
    goto L3;
    goto L2;
L2: Statement 3
L4: Statement 4

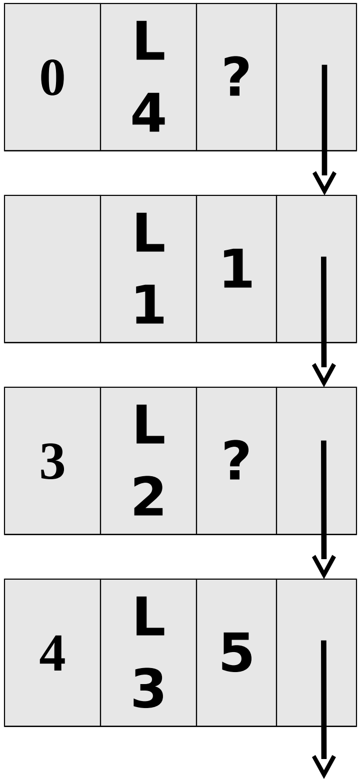


i	PB[i]
0	(jp, ?, ,)
1	statement 1
2	(jp, 1, ,)
3	(jp, ?, ,)
4	(jp, ?=5, ,)
5	statement 2
6	
7	
8	
9	
10	

goto statements (cont.)

- Example:

goto L4
L1: Statement1
    goto L1;
    goto L2;
    goto L3;
L3: Statement 2
    goto L1;
    goto L3;
    goto L2;
L2: Statement 3
L4: Statement 4

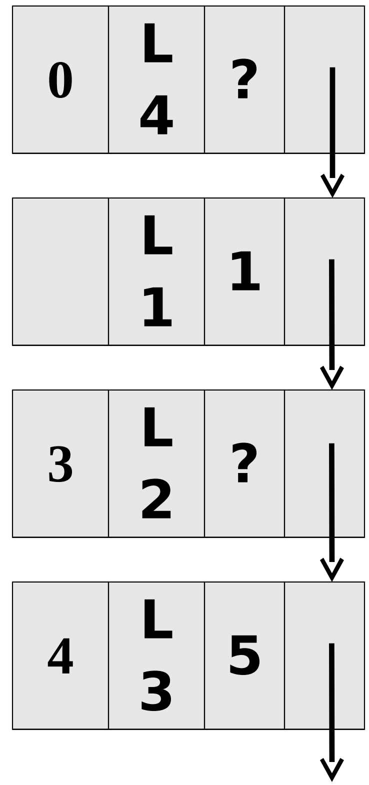


i	PB[i]
0	(jp, ?, ,)
1	statement 1
2	(jp, 1, ,)
3	(jp, ?, ,)
4	(jp, ?=5, ,)
5	statement 2
6	(jp, 1, ,)
7	
8	
9	
10	

goto statements (cont.)

- Example:

goto L4
L1: Statement1
    goto L1;
    goto L2;
    goto L3;
L3: Statement 2
    goto L1;
    goto L3;
    goto L2;
L2: Statement 3
L4: Statement 4

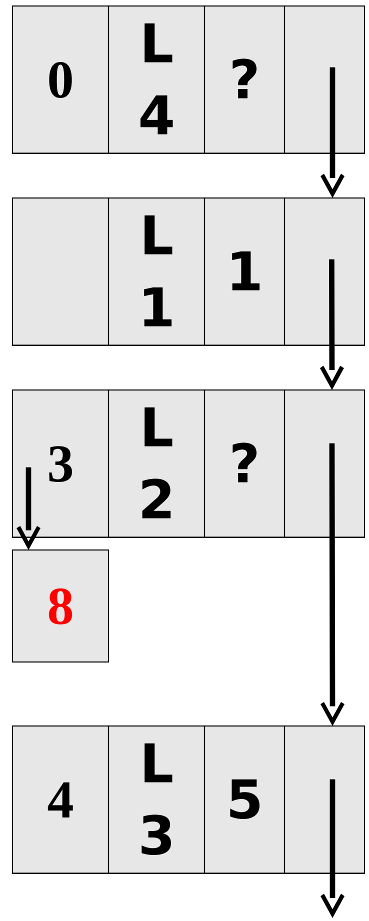


i	PB[i]
0	(jp, ?, ,)
1	statement 1
2	(jp, 1, ,)
3	(jp, ?, ,)
4	(jp, ?=5, ,)
5	statement 2
6	(jp, 1, ,)
7	(jp, 5, ,)
8	
9	
10	

goto statements (cont.)

- Example:

goto L4
L1: Statement1
    goto L1;
    goto L2;
    goto L3;
L3: Statement 2
    goto L1;
    goto L3;
    goto L2;
L2: Statement 3
L4: Statement 4

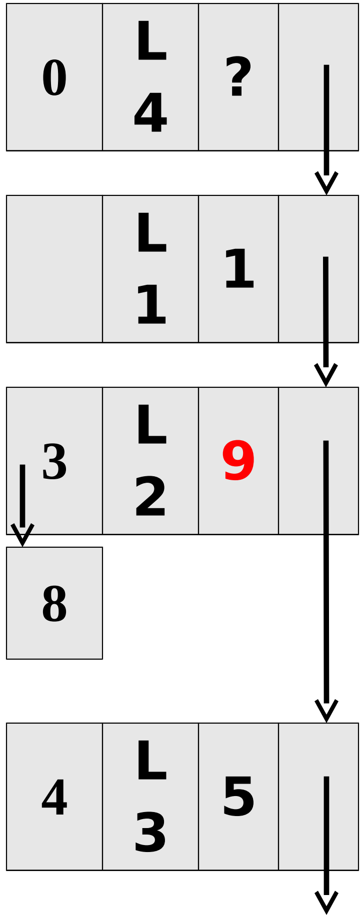


i	PB[i]
0	(jp, ?, ,)
1	statement 1
2	(jp, 1, ,)
3	(jp, ?, ,)
4	(jp, ?=5, ,)
5	statement 2
6	(jp, 1, ,)
7	(jp, 5, ,)
8	(jp, ?, ,)
9	
10	

goto statements (cont.)

- Example:

goto L4
L1: Statement1
    goto L1;
    goto L2;
    goto L3;
L3: Statement 2
    goto L1;
    goto L3;
    goto L2;
L2: Statement 3
L4: Statement 4

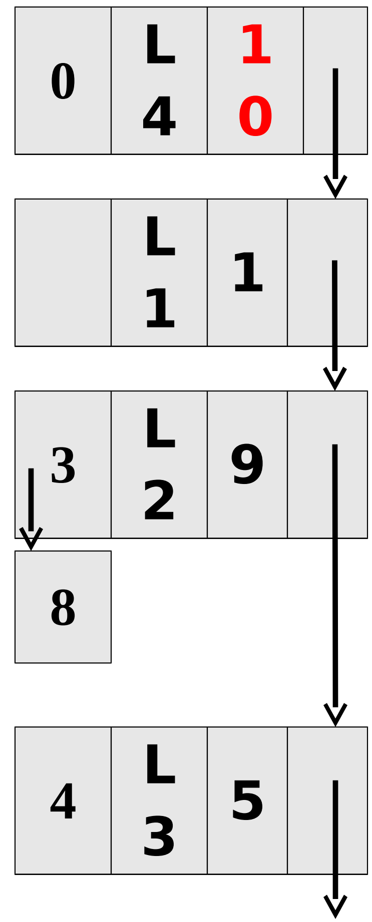


i	PB[i]
0	(jp, ?, ,)
1	statement 1
2	(jp, 1, ,)
3	(jp, ?=9, ,)
4	(jp, ?=5, ,)
5	statement 2
6	(jp, 1, ,)
7	(jp, 5, ,)
8	(jp, ?=9, ,)
9	statement 3
10	

goto statements (cont.)

- **Example:**

goto L4
L1: Statement1
    goto L1;
    goto L2;
    goto L3;
L3: Statement 2
    goto L1;
    goto L3;
    goto L2;
L2: Statement 3
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