Intermediate Code Generation

Lecture 9
Recommended Reading


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

- Parser
- Lexical Analyzer
- Semantic Analyzer
- Intermediate Code Generator
- ...

Diagram: A flowchart showing the relationships between the components of the Intermediate Code Generation process.
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 \( x := y \text{ op } z \) here represented as \((\text{op}, y, z, x)\)

• No built-up arithmetic expressions are allowed.

• As a result, \( x := y + z \ast w \) should be represented as:
  \[
  t_1 := z \ast w \\
  t_2 := y + t_1 \\
  x := t_2
  \]

• Three-address code is useful: related to machine-language/ simple/ optimizable.
<table>
<thead>
<tr>
<th>Type of Statement</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := y \ op \ z$</td>
<td>$(\text{op}, y, z, x)$</td>
</tr>
<tr>
<td>$x := \text{op} \ z$</td>
<td>$(\text{op}, z, x, \ )$</td>
</tr>
<tr>
<td>$x := z$</td>
<td>$(:=, z, x, \ )$</td>
</tr>
<tr>
<td><code>goto L</code></td>
<td><code>(\text{jp}, L, ,)</code></td>
</tr>
<tr>
<td><code>if x \ \text{relop} y \ goto L</code></td>
<td><code>(\text{relop}, x, y, L), \ or</code></td>
</tr>
<tr>
<td></td>
<td><code>(\text{jpf}, A1, A2, ),</code></td>
</tr>
<tr>
<td></td>
<td><code>(\text{jpt}, A1, A2, ), \ etc.$</code></td>
</tr>
</tbody>
</table>
Different Addressing Modes

(+, 100, 101, 102)

location 102 ← content of 100 + content of 101

(+, #100, 101, 102)

location 102 ← constant 100 + content of 101

(+, @100, 101, 102)

location 102 ← content of content of 100 + content of 101
Definitions

- **Action Symbols** (e.g., #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

- Program Block
- Data Block
- Runtime Memory
- Code
- Data
- Temporaries
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. $S \rightarrow \#\text{pid} \ id := E \ #\text{assign}$
2. $E \rightarrow T \ E'$
3. $E' \rightarrow \epsilon$
4. $E' \rightarrow + T \ #\text{add} \ E'$
5. $T \rightarrow F \ T'$
6. $T' \rightarrow \epsilon$
7. $T' \rightarrow * F \ #\text{mult} \ T'$
8. $F \rightarrow ( E )$
9. $F \rightarrow \#\text{pid} \ id$

e.g., input: $a := b + c \ast d$
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

\[
S \rightarrow \#pid \text{ id} := E \#assign \\
E \rightarrow T \ E' \\
E' \rightarrow \varepsilon \mid + \ T \ #add \ E' \\
T \rightarrow F \ T' \\
T' \rightarrow \varepsilon \mid * \ F \ #mult \ T' \\
F \rightarrow ( \ E ) \\
F \rightarrow \#pid \ id
\]

Parse Table

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>INPUT SYMBOL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>E'</td>
<td>+ T E'</td>
</tr>
<tr>
<td>T</td>
<td>F T'</td>
</tr>
<tr>
<td>T'</td>
<td>\varepsilon</td>
</tr>
<tr>
<td>F</td>
<td>id</td>
</tr>
<tr>
<td>S</td>
<td>id := E</td>
</tr>
</tbody>
</table>
### Example (Cont.)

<table>
<thead>
<tr>
<th>Parse Stack</th>
<th>Input</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>S $</td>
<td>id1 := id2 + id3 * id4 $</td>
<td>Pop</td>
</tr>
<tr>
<td>#pid id := E #assign $</td>
<td>id1 := id2 + id3 * id4 $</td>
<td>Call codegen(#pid), pop</td>
</tr>
<tr>
<td>id := E #assign $</td>
<td>id1 := id2 + id3 * id4 $</td>
<td>2 Matching, 2 pop</td>
</tr>
<tr>
<td>T E' #assign $</td>
<td>id2 + id3 * id4 $</td>
<td>Pop</td>
</tr>
<tr>
<td>F T' E' #assign $</td>
<td>id2 + id3 * id4 $</td>
<td>pop</td>
</tr>
<tr>
<td>#pid id T' E' #assign $</td>
<td>id2 + id3 * id4 $</td>
<td>Call codegen(#pid), pop</td>
</tr>
<tr>
<td>id T' E' #assign $</td>
<td>id2 + id3 * id4 $</td>
<td>Matching, pop</td>
</tr>
<tr>
<td>E' #assign $</td>
<td>+ id3 * id4 $</td>
<td>pop</td>
</tr>
<tr>
<td>+ T #add E' #assign $</td>
<td>+ id3 * id4 $</td>
<td>Matching, pop</td>
</tr>
<tr>
<td>F T' #add E' #assign $</td>
<td>id3 * id4 $</td>
<td>Pop</td>
</tr>
<tr>
<td>#pid id T' #add E' #assign $</td>
<td>id3 * id4 $</td>
<td>Call codegen(#pid), pop</td>
</tr>
<tr>
<td>id T' #add E' #assign $</td>
<td>id3 * id4 $</td>
<td>Matching, 2 pop</td>
</tr>
<tr>
<td>* F #mult T' #add E' #assign $</td>
<td>* id4 $</td>
<td>Matching, pop</td>
</tr>
<tr>
<td>#pid id #mult T' #add E' #assign $</td>
<td>id4 $</td>
<td>Call codegen(#pid), pop</td>
</tr>
<tr>
<td>id #mult T' #add E' #assign $</td>
<td>id4 $</td>
<td>Matching, pop</td>
</tr>
<tr>
<td>#mult T' #add E' #assign $</td>
<td>id4 $</td>
<td>Call codegen(#mult), pop</td>
</tr>
<tr>
<td>T' #add E' #assign $</td>
<td>id4 $</td>
<td>pop</td>
</tr>
<tr>
<td>#add E' #assign $</td>
<td>id4 $</td>
<td>Call codegen(#add), pop</td>
</tr>
<tr>
<td>E' #assign $</td>
<td>id4 $</td>
<td>pop</td>
</tr>
<tr>
<td>#assign $</td>
<td>id4 $</td>
<td>Call codegen(#assign), pop</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>Finish!!</td>
</tr>
</tbody>
</table>
Example (Cont.)

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

Input:
\[ a := b + c * d \]
\[ \text{id}_1 := \text{id}_2 + \text{id}_3 * \text{id}_4 \]
Example (Cont.)

• Semantic Stack (SS) status during code generation:

• Program Block (PB):

<table>
<thead>
<tr>
<th>i</th>
<th>PB[i]</th>
<th>Semantic Action called</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(*,103,102,500)</td>
<td>#mult</td>
</tr>
<tr>
<td>1</td>
<td>(+,500,101,501)</td>
<td>#add</td>
</tr>
<tr>
<td>2</td>
<td>(=,501,100, )</td>
<td>#assign</td>
</tr>
</tbody>
</table>
Example 2

- Suppose we only had the right hand side expression

\[
\text{Input: } \quad b + c \ast d \\
\text{id}_2 + \text{id}_3 \ast \text{id}_4
\]
Example 2 (Cont.)

- A temporary memory address will remain in SS:

<table>
<thead>
<tr>
<th>i</th>
<th>PB[i]</th>
<th>Semantic Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(*, 103, 102, 500)</td>
<td>#mult</td>
</tr>
<tr>
<td>1</td>
<td>(+, 500, 101, 501)</td>
<td>#add</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input:
\[ b + c \times d \]
\[ id_2 + id_3 \times id_4 \]

<table>
<thead>
<tr>
<th>no</th>
<th>lexeme</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>102</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>103</td>
</tr>
</tbody>
</table>

Symbol Table
Control statements (while)

\[ S \rightarrow \text{while } E \text{ do } S \text{ end} \]

Input Example:

\[
\begin{align*}
\text{while } (b+c*d) \quad & \text{do} \\
\quad & a := a+1 \\
\text{end}
\end{align*}
\]

- We need to find the appropriate place for inserting Semantic Actions symbols.
Control statements (while)

$\text{while } E \text{ do } S \text{ end}$

Input Example:

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

<table>
<thead>
<tr>
<th>i</th>
<th>PB[i]</th>
<th>Semantic Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(*, 103, 102, 500)</td>
<td>#mult</td>
</tr>
<tr>
<td>1</td>
<td>(+, 500, 101, 501)</td>
<td>#add</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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)

\[ S \rightarrow \text{while } E \text{ do } S \text{ end} \]

Input Example:

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

- This is done by the semantic routines of non-terminal \( E \)
Control statements (while)

\[ S \rightarrow \text{while } E \text{ do } S \text{ end} \]

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

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

Input Example:

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

Conditional jump
Control statements (while)

\[ S \rightarrow \text{while } E \text{ do } S \text{ end} \]

- Solution: BACKPATCHING!

  \#save: begin
  \hspace{1cm} \text{push(i)}
  \hspace{1cm} i \leftarrow i + 1
  \hspace{1cm} \text{end}

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

Input Example:

\[
\text{while } (b+c*d) \bullet \text{ do } \hspace{1cm} a := a+1 \hspace{1cm} \text{end}
\]

Conditional jump
Control statements (while)

#label

\[ S \rightarrow \text{while } E \text{ do } \#\text{save } S \text{ end} \]

- 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

\[ \text{push}(i) \]

end

Input Example:

while (b+c*d) do

\[ a := a+1 \]

end
Control statements (while)

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)
end

Input Example:

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

Unconditional jump
Conditional jump
Control statements (while)

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

<table>
<thead>
<tr>
<th>i</th>
<th>PB[i]</th>
<th>Semantic Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input Example:

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

\[ S \rightarrow \text{while} \ #\text{label} \ E \ \text{do} \ #\text{save} \ S \ #\text{while} \ \text{end} \]

<table>
<thead>
<tr>
<th>i</th>
<th>PB[i]</th>
<th>Semantic Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input Example:

\[
\text{while} \ (b+c*d) \ \text{do} \\
a := a+1 \\
\text{end}
\]
Control statements (while)

\[ S \rightarrow \text{while} \#\text{label} E \text{ do } \#\text{save} S \#\text{while} \text{ end} \]

<table>
<thead>
<tr>
<th>i</th>
<th>PB[i]</th>
<th>Semantic Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(*, 103, 102, 500)</td>
<td>by E</td>
</tr>
<tr>
<td>1</td>
<td>(+, 500, 101, 501)</td>
<td>by E</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input Example:

\[ \text{while } \bullet (b+c*d) \bullet \text{ do} \]
\[ a := a+1 \]
\[ \bullet \text{ end} \]

Unconditional jump

Conditional jump
Control statements (while)

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

<table>
<thead>
<tr>
<th>i</th>
<th>PB[i]</th>
<th>Semantic Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(*, 103, 102, 500)</td>
<td>by E</td>
</tr>
<tr>
<td>1</td>
<td>(+, 500, 101, 501)</td>
<td>by E</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input Example:

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

\[ S \rightarrow \text{while} \#\text{label} E \text{ do } \#\text{save} S \text{ endwhile} \]

<table>
<thead>
<tr>
<th>i</th>
<th>PB[i]</th>
<th>Semantic Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(*, 103, 102, 500)</td>
<td>by E</td>
</tr>
<tr>
<td>1</td>
<td>(+, 500, 101, 501)</td>
<td>by E</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(+, 100, #1, 503)</td>
<td>by S</td>
</tr>
<tr>
<td>4</td>
<td>(=, 503, 100, )</td>
<td>by S</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

After \#save
```
<table>
<thead>
<tr>
<th>501</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```

After S
```
<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>501</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```

Unconditional jump

Conditional jump
## Control statements (while)

### Syntax:

\[ S \rightarrow \text{while} \#label\ E\ \text{do}\ \#\text{save}\ S\ \#\text{while}\ \text{end} \]

### Example Input:

```
while \bullet (b+c*d) \bullet \text{do}
\begin{align*}
\text{a} &:= \text{a+1} \\
\text{end}
\end{align*}
```

### Semantic Actions:

<table>
<thead>
<tr>
<th>i</th>
<th>PB[i]</th>
<th>Semantic Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(*, 103, 102, 500)</td>
<td>by E</td>
</tr>
<tr>
<td>1</td>
<td>(+, 500, 101, 501)</td>
<td>by E</td>
</tr>
<tr>
<td>2</td>
<td>(jpf, 501, 6, )</td>
<td>#while</td>
</tr>
<tr>
<td>3</td>
<td>(+, 100, #1, 503)</td>
<td>by S</td>
</tr>
<tr>
<td>4</td>
<td>(=, 503, 100, )</td>
<td>by S</td>
</tr>
<tr>
<td>5</td>
<td>(jp, 0, , )</td>
<td>#while</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Diagram:

- **Unconditional Jump:** \( \text{jpf} \)
- **Conditional Jump:** \( \text{jp} \)
Conditional Statements (if-then-else)

\[ S \rightarrow \text{if } E \text{ then } S \ S' \]
\[ S' \rightarrow \text{else } S \]
\[ S' \rightarrow \varepsilon \]

Input Example:

If \((a+b)\) then \(a := a+1\)
else \(b := b+1\)
Conditional Statements (if-then-else)

\[ S \rightarrow \text{if } E \ #save \ \text{then } S \ S' \]
\[ S' \rightarrow \text{else } S \]
\[ S' \rightarrow \varepsilon \]

**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
```

**Input Example**:

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

else \(b := b+1\)

**Conditional jump** vs **Unconditional jump**
Conditional Statements (if-then-else)

\[
S \rightarrow \text{if } E \#\text{save then } S \ S' \\
S' \rightarrow \text{else } \#\text{jpf\_save } S \\
S' \rightarrow \varepsilon
\]

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

\[
\text{PB[ss(top)] } \leftarrow (\text{jpf,ss(top-1), i+1, }) \\
\text{Pop(2), push(i), i } \leftarrow i + 1;
\]

end
Conditional Statements (if-then-else)

\[
\begin{align*}
S & \rightarrow \text{if } E \#\text{save then } S \ S' \\
S' & \rightarrow \text{else } \#\text{jpf\_save } S \ #\text{jp} \\
S' & \rightarrow \varepsilon
\end{align*}
\]

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

\#\text{jp}: begin

PB[ss(top)] \leftarrow (jp, i, , )

Pop(1)

dend

Input Example:

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

else \(b := b+1\)
Conditional Statements (if-then-else)

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

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

Input Example:
If (a+b) ● then a := a+1 ●

Compare with #jpf_save

conditional jump
Conditional Statements (if-then-else)

\[
S \rightarrow \text{if } E \ #\text{save} \text{ then } S \ S' \\
S' \rightarrow \text{else } \#\text{jpf\_save} \ S \ \#\text{jp} \\
S' \rightarrow \#\text{jpf}
\]

Program Block:

<table>
<thead>
<tr>
<th>i</th>
<th>PB[i]</th>
<th>Semantic Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(+, a, b, t1)</td>
<td>#add</td>
</tr>
<tr>
<td>1</td>
<td>(jpf, t1, ?=5, )</td>
<td>#save</td>
</tr>
<tr>
<td>2</td>
<td>(+, a, #1, t2)</td>
<td>#add</td>
</tr>
<tr>
<td>3</td>
<td>(:=, t2, a, )</td>
<td>#assign</td>
</tr>
<tr>
<td>4</td>
<td>(jp, ?=7, , )</td>
<td>#jpf_save</td>
</tr>
<tr>
<td>5</td>
<td>(+, b, #1, t3)</td>
<td>#add</td>
</tr>
<tr>
<td>6</td>
<td>(:=, t3, b, )</td>
<td>#assign</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>#jp</td>
</tr>
</tbody>
</table>

Input Example:

If \((a+b)\) then \(a := a+1\)  
else \(b := b+1\)
Goto Statements

\[ S \rightarrow \text{goto id;} \]
\[ S \rightarrow \text{id: } S \]

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
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
Goto Statements (Cont.)

Example:

0  goto L4
1  L1: Statement 1
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

<table>
<thead>
<tr>
<th>i</th>
<th>PB[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(jp, ?, , )</td>
</tr>
<tr>
<td>1</td>
<td>statement 1</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
**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

<table>
<thead>
<tr>
<th>i</th>
<th>PB[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(jp, ?, , )</td>
</tr>
<tr>
<td>1</td>
<td>statement 1</td>
</tr>
<tr>
<td>2</td>
<td>(jp, 1, , )</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
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

```
<table>
<thead>
<tr>
<th>i</th>
<th>PB[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(jp, ?, , )</td>
</tr>
<tr>
<td>1</td>
<td>statement 1</td>
</tr>
<tr>
<td>2</td>
<td>(jp, 1, , )</td>
</tr>
<tr>
<td>3</td>
<td>(jp, ?, , )</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
```
Goto Statements (Cont.)

Example:

0  goto L4
1  L1: Statement 1
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

The diagram illustrates the goto statements and the corresponding PB[i] values:

<table>
<thead>
<tr>
<th>i</th>
<th>PB[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(jp, ?, , )</td>
</tr>
<tr>
<td>1</td>
<td>statement 1</td>
</tr>
<tr>
<td>2</td>
<td>(jp, 1, , )</td>
</tr>
<tr>
<td>3</td>
<td>(jp, ?, , )</td>
</tr>
<tr>
<td>4</td>
<td>(jp, ?, , )</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
**Example:**

0  goto L4
1  L1: Statement 1
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

**Goto Statements (Cont.)**

<table>
<thead>
<tr>
<th>i</th>
<th>PB[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( (\text{jp}, ?, , ) )</td>
</tr>
<tr>
<td>1</td>
<td>statement 1</td>
</tr>
<tr>
<td>2</td>
<td>( (\text{jp}, 1, , ) )</td>
</tr>
<tr>
<td>3</td>
<td>( (\text{jp}, ?, , ) )</td>
</tr>
<tr>
<td>4</td>
<td>( (\text{jp}, 5, , ) )</td>
</tr>
<tr>
<td>5</td>
<td>statement 2</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
Goto Statements (Cont.)

Example:

0 goto L4
1 L1: Statement 1
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

<table>
<thead>
<tr>
<th>i</th>
<th>PB[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(jp, ?, , )</td>
</tr>
<tr>
<td>1</td>
<td>statement 1</td>
</tr>
<tr>
<td>2</td>
<td>(jp, 1, , )</td>
</tr>
<tr>
<td>3</td>
<td>(jp, ?, , )</td>
</tr>
<tr>
<td>4</td>
<td>(jp, 5, , )</td>
</tr>
<tr>
<td>5</td>
<td>statement 2</td>
</tr>
<tr>
<td>6</td>
<td>(jp, 1, , )</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>i</th>
<th>PB[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(jp, ?, , )</td>
</tr>
<tr>
<td>1</td>
<td>statement1</td>
</tr>
<tr>
<td>2</td>
<td>(jp, 1, , )</td>
</tr>
<tr>
<td>3</td>
<td>(jp, ?, , )</td>
</tr>
<tr>
<td>4</td>
<td>(jp, 5, , )</td>
</tr>
<tr>
<td>5</td>
<td>statement2</td>
</tr>
<tr>
<td>6</td>
<td>(jp, 1, , )</td>
</tr>
<tr>
<td>7</td>
<td>(jp, 5, , )</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
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.)
Goto Statements (Cont.)

Example:

0  goto L4
1  L1: Statement 1
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

<table>
<thead>
<tr>
<th>i</th>
<th>PB[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(jp, ?, , )</td>
</tr>
<tr>
<td>1</td>
<td>statement 1</td>
</tr>
<tr>
<td>2</td>
<td>(jp, 1, , )</td>
</tr>
<tr>
<td>3</td>
<td>(jp, 9, , )</td>
</tr>
<tr>
<td>4</td>
<td>(jp, 5, , )</td>
</tr>
<tr>
<td>5</td>
<td>statement 2</td>
</tr>
<tr>
<td>6</td>
<td>(jp, 1, , )</td>
</tr>
<tr>
<td>7</td>
<td>(jp, 5, , )</td>
</tr>
<tr>
<td>8</td>
<td>(jp, 9, , )</td>
</tr>
<tr>
<td>9</td>
<td>statement 3</td>
</tr>
</tbody>
</table>
Goto Statements (Cont.)

Example:

0    goto L4
1    L1: Statement 1
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
Bottom-Up Code Generation

1. \( S \rightarrow X \text{id} := E \)
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. \( F \rightarrow X \text{id} \)
10. \( X \rightarrow \varepsilon \)
11. \( Y \rightarrow \varepsilon \)
12. \( Z \rightarrow \varepsilon \)

- 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

new rules
Bottom-Up Code Generation

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
  end

End codegen
```
Question?

\[ S \to \text{repeat } S \text{ until } E \text{ end} \]

- 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.

Input 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.

\[
S \rightarrow \text{for id := } E_1 \text{ to } E_2 \text{ A do } S \text{ end}
\]
\[
A \rightarrow \text{by } E_3
\]
\[
A \rightarrow \varepsilon
\]

\[
b+c : \text{loop variable (}j\text{) initial value}
\]
\[
a*b : \text{loop variable (}j\text{) limit (constant)}
\]
\[
c*d : \text{loop variable (}j\text{) step (constant)}
\]

Input Example:

\[
\text{for } j := b+c \text{ to } a*b \text{ by } c*a \text{ do}
\]
\[
d := d+j
\]
end
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 semantic 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
```

**Grammar:**

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