Semantic Actions and 3-Address Code Generation

Introduction

We start with different syntactic categories and discuss the corresponding semantic actions and intermediate code generation if there is any.

Grammar of Simple Variable Declaration

$$
\begin{aligned}
\text{DL} &\rightarrow \text{D ; DL} \\
&\rightarrow \varepsilon \\
\text{D} &\rightarrow \text{TY VL} \\
\text{TY} &\rightarrow \text{int} \mid \text{real} \\
\text{VL} &\rightarrow \text{VL , id} \mid \text{id}
\end{aligned}
$$

Synthesized Attributes

- The variable VL may have a synthesized attribute locLst, a list of indices of the symbol table where names are inserted.
- Type and other information of these names will be updated afterward[a].
- The non-terminal TY saves the type name in its synthesized attribute TY.type.

[a] There is an alternate mechanism available if we can access the stack below the handle.

Note

- In our simple case it is `int` or `float`.
- But it can be multi-dimensional array of any base type e.g. `int a[3][4][5]` - 3-element array of 4-element array of 5-element array of integers. In fact there may be upper and lower bounds of array indices for every dimension.

Note

- In case of a defined type like structure or disjoint sum there is a list of fields and the type of each one of them may be built-in or defined.
- A defined type name can be saved with all its field information and sizes.
- A variable of a defined type may have a link to the corresponding type entry in type table.

Note

- In case of a procedure or function name we need to save the number of parameters and their types. Also the type of the value it returns.
- If the whole type information is available, its size etc. can be calculated and stored.
- In our simple case we may store the size and the offset of the memory location from a base address of activation record.

Important Functions

- searchInsert(symTab, lexme, err): it searches the current symbol table with the second parameter[a].
- In a normal situation there should not be any entry of the lexme. It is inserted in the table and the index is returned[b].

[a] There may be separate functions for search() and insert().

[b] If the scanner inserts lexme it in the symbol table, it will be slightly different.

Important Functions

- If the lexme is found in the table (already inserted), it is an error condition.
- The type of the identifier is still unknown.
- mkLocLst(loc): makes a list of symbol-table location specified by loc and returns the single element list.

Important Functions

- catLocLst(l1,l2): concatenates two lists of symbol-table locations and returns the concatenated list.
- updateType(symTab, l, type): updates type of the symbol-table locations from the list l using type.

Semantic Actions and Code Generation

TY → int {TY.type = INT}

TY → real {TY.type = FLOAT}

VL → id

{ temp = searchInsert(symTab, id.lexme, err)

VL.locLst = mkLocLst(temp) }

VL → VL_1 , id

{ temp = searchInsert(symTab, id.lexme, err)

VL.locLst = catLocLst(VL_1.locLst, mkLocLst(temp) }

D → TY VL { updateType(symTab, VL.locLst, TY.type }

Error

What should we do if `searchInsert()` gives an error?

Expression Grammar

- Our next consideration is the expression grammar.
- We shall consider a small portion of it without involving array etc.

Part of Expression Grammar

$$
\begin{aligned}
\mathrm{E} &\rightarrow \mathrm{E} + \mathrm{E} \\
&\rightarrow \mathrm{id} \\
&\rightarrow \mathrm{ic} \\
&\rightarrow \mathrm{fc}
\end{aligned}
$$

Where id is a simple scalar variable, ic is an integer constant and fc is a floating-point constant.

Synthesized Attributes

- An expression E has two attributes, E.loc which is an index to the symbol table, and E.type[a].
- The symbol table entry corresponding to E.loc may be a program defined variable or a compiler generated variable.

[a] Which is also available in the symbol table.

Important Functions

- searchInsert(symTab, lexme, err): is as we have already defined.
- But in this case, if the lexme corresponds to a program variable and it is not found in the symbol-table, it is an error. Necessary actions are to be taken[a].

[a] We may insert the name in the symbol table with a type UNDEF or some default type. This will stop generating error message on the same undefined variable.

Important Functions

- The function newTemp() generates a compiler defined variable name. Its type is determined by the type of the expression being evaluated.
- The function getType(symTab, loc) returns the type of the variable at the index loc of the symbol table.

Semantic Actions and Code Generation

E → id

{E.loc = searchInsert(symTab, ID.lexme, err) }

{E.type = getType(symTab, E.loc) }

E → ic

{E.loc =

searchInsert(symTab, newTemp(), err)

updateType(symTab, mkLocLst(E.loc), INT)

E.type = INT

codeGen(assIntConst, ic.val, E.loc)}

Semantic Actions and Code Generation

E → fc

{E.loc =

searchInsert(symTab, newTemp(), err)

updateType(symTab, mkLocLst(exp.loc), FLOAT)

E.type = FLOAT

codeGen(assFltConst, fc.val, E.loc)}

Semantic Actions and Code Generation

$E \rightarrow E_1 + E_2$

{if E_1.type = INT and
E_2.type = INT then
E.loc = searchInsert(symTab, newTemp(), err)
updateType(symTab, mkLocLst(E.loc), INT)
E.type = INT
codeGen(assIntPlus, E_1.loc, E_2.*loc*, E.loc)}

Semantic Actions and Code Generation

if E_1.type = FLOAT

E_2.type = FLOAT then

E.loc = searchInsert(symTab, newTemp(), err)

updateType(symTab, mkLocLst(exp.loc), FLOAT)

E.type = FLOAT

codeGen(assFltPlus, E_1.loc, E_2.loc, E.loc)

Semantic Actions and Code Generation

if E_1.type = INT

E_2.type = FLOAT then

temp = searchInsert(symTab, newTemp(),err)

updateType(symTab, mkLocLst(temp),FLOAT)

codeGen(assignIntToFlt, E_1.loc, temp)

E.loc = searchInsert(symTab, newTemp(),err)

updateType(symTab, mkLocLst(E.loc),FLOAT)

E.type = FLOAT

codeGen(assFltPlus, temp, E_2.*loc*, E.loc)

Another case is similar.

Where to Store the Code

- The question is where to store the generated 3-address codes.
- They may be saved in a global array, or
- They may be kept as another attribute of E.

Grammar for Statements

Our next considerations are statements. We start with simple assignment statement.

Grammar Simple Assignment Statement

$$\text{AS} \rightarrow \text{id} = \text{E}$$

We assume that id is a simple scalar variable.

Semantic Actions and Code Generation

AS

→ id = E

{temp = searchInsert(symTab, id.lexme, err)

if getType(symTab,temp) = UNDEF then ERROR

if (getType(symTab, temp) = INT and E.type = INT) or

(getType(temp) = FLOAT and E.type = FLOAT) then

codeGen(assign, E.loc, temp)

Semantic Actions and Code Generation

```
if (getType(symTab,temp) = INT and E.type = FLOAT) then
    codeGen(assignFltToInt, E.loc, temp)
if (getType(symTab,temp)=FLOAT and E.type=INT) then
    codeGen(assignIntToFlt, E.loc, temp) }
```

Control Flow Statements

Our next consideration is the statements that control the flow of execution. We talk about two techniques to handle jump/branch addresses. Following is a sample fragment of grammar used to control the flow of execution.

A Grammar

$$
\begin{aligned}
P &\rightarrow S \\
S &\rightarrow S\ S \\
S &\rightarrow \mathrm{id} = E \\
S &\rightarrow \texttt{if}\ (\ B\ )\ S \\
S &\rightarrow \texttt{if}\ (\ B\ )\ S\ \texttt{else}\ S \\
S &\rightarrow \texttt{while}\ (\ B\ )\ S
\end{aligned}
$$

Boolean Expressions: a Grammar

$$BE \rightarrow BE \,||\, BE$$

$$BE \rightarrow BE \,\&\&\, BE$$

$$BE \rightarrow BE$$

$$BE \rightarrow (\ BE\ )$$

$$BE \rightarrow E \,@\, E,\ E : \text{arithmetic expression}$$

$$BE \rightarrow \text{true} \mid \text{false}.$$

where $@ \in \{<, >, \leq, \geq, ==, ! =\}$.

Two usage of Boolean Expressions

- Used to control flow of execution. The boolean value implicitely determines the program point where the control reaches.
- The value is computed like an expression and assigned to a variable.

Control Flow Statements

- Boolean expressions and flow-of-control statements require branch instructions.
- The branch target is unknown when the 3-address code for branch instructions are generated.
- One solution is to pass the label of the branch target as an inherited attribute.

Attributes of S

- Each statement S has an inherited attribute, the label of the next instruction to execute (the continuation of S). Call it S.next.
- Each statement S also has a synthesized attribute holding the code sequence corresponding to S. Call it S.code.

Short-Circuit Code

- It may not be necesary to evaluate all parts of a Boolean expression.
- It depends on the definition of the language and also on the operator.
- Operators do not appare in the code. The value of the expression is indirectly represented by the position in the code.

Short-Circuit Code: an Example

- ```
  if (x < 10 || y > 2) a = 0; else a = 1;
  ```
- ```
  if x < 10 goto L1
  if y > 2 goto L1
  goto L2
  L1: a = 0
      goto L3
  L2: a = 1
  L3:
  ```

Short-Circuit Code

- Short-Circuit code is possible for operators not, and and or.
- But it is not possible for operators like xor. One needs to evaluate both the relations.

Attributes of BE

- Each Boolean expression BE has a synthesized attribute holding the code sequence corresponding to BE. Call it BE.code.
- Each Boolean expression BE has two inherited attributes. BE.true is the label of the instruction that will be executed next, if BE evaluates to true. Similar is BE.false, if BE evaluates to false.

if-statement

S $\rightarrow$ `if` (BE) S_1 `else` S_2.

- S.next is passed to S_1.next and S_2.next.
- At the end of the code sequence of S_1, there is `goto` S_1.next.

Example: if-statement

- Before generating code for the Boolean expression the labels corresponding to BE.true and BE.false are generated.
- The label BE.true is placed before S_1 and BE.false is placed before S_2.

Code for S

- $S \rightarrow$ `if` (BE) S_1 `else` S_2
- S.code = BE.code + L_1 : + S_1.code + `goto` L + L_2 : + S_2.code
- S.next = L, BE.true = L_1, BE.false = L_2.

Example: Boolean Expression

$\text{BE} \rightarrow \text{BE}_1 \,||\, \text{BE}_2$

- A new label $L.i$: is generated and placed before the code of BE_2.
- $\text{BE}_1.\text{true} = \text{BE.true}$ and $\text{BE}_1.\text{false} = L.i$.
- $\text{BE}_2.\text{true} = \text{BE.true}$ and $\text{BE}_2.\text{false} = \text{BE.false}$.

Code for S

- $S \rightarrow$ `if` $(\text{BE}_1 \mid\mid \text{BE}_2)\ S_1$ `else` S_2
- $S.\text{code} = \text{BE}_1.\text{code} + L_1\text{:} + \text{BE}_2.\text{code} + L_2\text{:} + S_1.\text{code} +$ `goto` $S.\text{next} + L_3\text{:} + S_2.\text{code}$
- $\text{BE}_1.\text{false} = L_1$, $\text{BE}_1.\text{true} = \text{BE}_2.\text{true} = L_2$, $\text{BE}_1.\text{false} = L_3$

Example: Boolean Expression

$BE \rightarrow BE_1 \;\&\&\; BE_2$

- A new label $L.i$: is generated and placed before the code of BE_2.
- $BE_1.true = L.i$ and $BE_1.false = BE.false$.
- $BE_2.true = BE.true$ and $BE_2.false = BE.false$.

Example: Boolean Expression

$BE \rightarrow \; ! \; BE_1$

- $BE_1.true = BE.false.$
- $BE_1.false = BE.true.$

Example: Relational Expression

A base case of a Boolean expression is a relational expression. BE $\rightarrow$ E_1 relOp E_2

Where relOp is == <= >= <> < >, and the generated code is

codeGen(relOpTrue, E_1.loc, E_2.loc, BE.true)
codeGen(goto BE.false).

Exercise

Write semantic actions and generate code for while-statement

S $\rightarrow$ while BE do S

Backpatching: an Alternate Approach

Backpatching in Control Flow Statements

- The target location is not yet known. We may bind the labels afterward (lazy).
- Backpatching is an alternate approach where the targets of code corresponding to branch/jump instructions are kept unfilled.
- List of indices of unfilled code is passed as a synthesized attributes.

Backpatching in Control Flow Statements

- Target holes in these 3-address codes will be filled (backpatched) when the target labels are generated.
- Production rules of boolean expression and control flow statements are modified by introducing special non-terminals, known as markers producing null strings.

Modified Grammar of Boolean Expression

$$
\begin{aligned}
\text{BE} &\rightarrow \text{BE or mR BE} \\
&\rightarrow \text{BE and mR BE} \\
&\rightarrow \text{not BE} \\
&\rightarrow \text{(BE)} \\
&\rightarrow \text{E relOP E} \\
\text{mR} &\rightarrow \varepsilon \text{ (new Marker non-terminal)}
\end{aligned}
$$

Synthesized Attributes

- The non-terminal BE has two synthesized attributes trueLst and falseLst.
- BE.trueLst is the list of 3-address codes (indices) corresponding to jumps/branches that will be taken when the expression of BE evaluates to true.

Synthesized Attributes

- Similarly BE.falseLst is the list of code indices from where jump/branches are taken when BE evaluates to false.
- The BE.trueLst will be backpatched by the index of the 3-address code where the control will be transferred when BE evaluates to true.
- Similar is the case for BE.falseLst.

Sequence Number of an Instructions

- There is a sequence number or index of every instruction If they are stored in a global array. These indices are used as labels[a].
- Following are a few useful functions for semantics actions.

[a] If the sequence of instructions is maintained as a list, then we may have a label in the list or a pointer to the target instruction.

Important Functions

- mkLst(i): makes a single element list with the code index i and returns the pointer of the list.
- catLst(l_1,l_2): two lists pointed by l_1 and l_2 are concatenated and returned as a list.

Important Functions

- fill(l, i): the unfilled targets of each jump/branch instruction whose indices are in the list l are filled/backpatched by the index i of the target instruction.
- The global variable nextInd has the sequence number(index) of next 3-address code to be generated.

Semantic Actions and Code Generation

- The non-terminal mR has a synthesized attribute nextInd, the current value of the variable nextInd.
- $\text{mR} \rightarrow \varepsilon$

  $\{\text{mR.nextInd} = \text{nextInd}\}$

An Alternative

- As an alternative the non-terminal mR has a synthesized attribute label. The reduction of mR generates a new label, attaches it to the next 3-address code and saves it in mR.label.

  $\text{mR} \rightarrow \varepsilon$

- $\{\text{mR.label} = \text{newlabel}()\}$

  $\{\text{codeGen(label, mR.label)}\}$

Semantic Actions and Code Generation

$$
\begin{aligned}
BE \rightarrow\ & exp_1\ relOP\ exp_2 \\
& \{BE.trueLst = mkLst(nextInd) \\
& BE.falseLst = mkLst(nextInd+1) \\
& codeGen(\text{`if relOP'}, exp_1.loc, \\
& \qquad exp_2.loc, \text{`goto'} \cdots) \\
& codeGen(\text{`goto'} \cdots) \\
& nextInd = nextInd+2\ \}
\end{aligned}
$$

Semantic Actions and Code Generation

$$
\begin{aligned}
\mathrm{BE} \rightarrow\ & \mathrm{BE}_1 \text{ or } \mathrm{mR}\ \mathrm{BE}_2 \\
& \{\mathrm{fill}(\mathrm{BE}_1.\mathrm{falseLst}, \mathrm{mR}.\mathrm{nextInd}) \\
& \mathrm{BE}.\mathrm{trueLst} = \mathrm{catLst}(\mathrm{BE}_1.\mathrm{trueLst}, \\
& \qquad \mathrm{BE}_2.\mathrm{trueLst}) \\
& \mathrm{BE}.\mathrm{falseLst} = \mathrm{BE}_2.\mathrm{falseLst}\ \}
\end{aligned}
$$

Semantic Actions and Code Generation

$$
\begin{aligned}
\mathrm{BE} \rightarrow\ & \mathrm{BE}_1 \text{ and } \mathrm{mR}\ \mathrm{BE}_2 \\
& \{\mathrm{fill}(\mathrm{BE}_1.\mathrm{trueLst}, \mathrm{mR}.\mathrm{nextInd}) \\
& \mathrm{BE}.\mathrm{falseLst} = \mathrm{catLst}(\mathrm{BE}_1.\mathrm{falseLst}, \\
& \qquad \mathrm{BE}_2.\mathrm{falseLst}) \\
& \mathrm{BE}.\mathrm{trueLst} = \mathrm{BE}_2.\mathrm{trueLst}\ \}
\end{aligned}
$$

Semantic Actions and Code Generation

$$\begin{aligned} BE \rightarrow\ & \text{not } BE_1 \\ & \{BE.falseLst = BE_1.trueLst \\ & BE.trueLst = BE_1.falseLst\ \} \end{aligned}$$

Semantic Actions and Code Generation

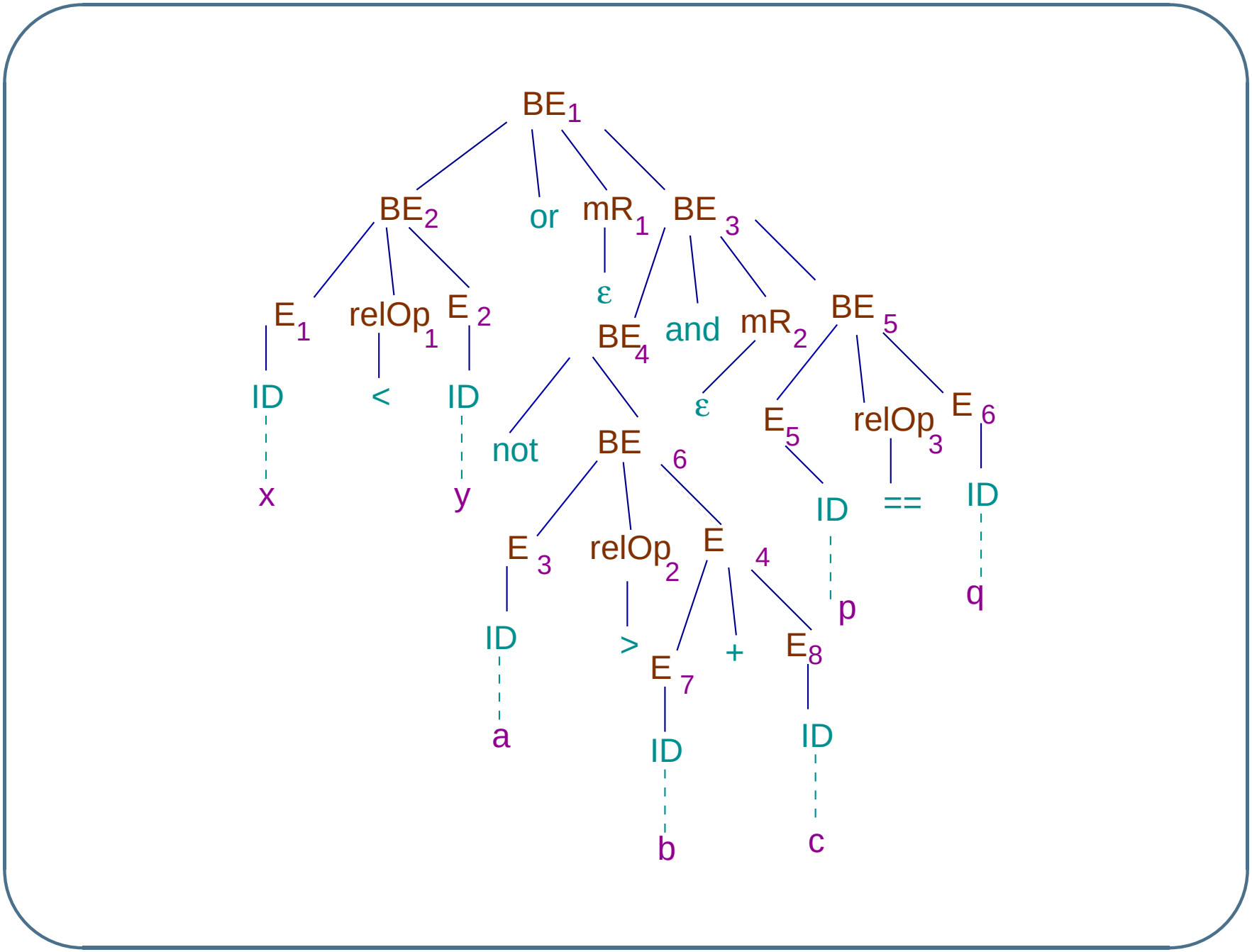
$$\begin{aligned} BE \rightarrow\ & (BE_1) \\ & \{BE.falseLst = BE_1.falseLst \\ & BE.trueLst = BE_1.trueLst \ \} \end{aligned}$$

Example

Consider the Boolean expression

```
x <= y or not a > b + c and p = q
```

Boolean Expression: Parse Tree


BE1
BE2
or
mR1
BE3
E1
relOp1
E2
ε
BE4
and
mR2
BE5
ID
<
ID
ε
E5
relOp3
E6
not
BE6
ID
==
ID
x
y
E3
relOp2
E4
p
q
ID
>
+
E8
E7
ID
ID
a
b
c

Example

- Let the next index (nextInd) of the 3-address code sequence be 100.
- The 3-address codes corresponding to BE_2 in readable form is

  100 `if x < y goto` $\cdots$

  101 `goto` $\cdots$
- BE_2.trueLst $= \{100\}$ and BE_2.falseLst $= \{101\}$ and nextInd: 102.

Example

- Next reduction is $\text{mR}_1 \rightarrow \varepsilon$. The attribute mR_1.nextInd $\leftarrow$ nextInd: 102.
- Next 3-address code is due to exp_4.

  102 `$i ← b + c`
- Then the code corresponding to BE_6 is

  103 `if a > $i goto` $\cdots$

  104 `goto` $\cdots$

Example

- BE_6.trueLst = {103} and BE_6.falseLst = {104} and nextInd: 105.
- The not operator flips the lists.
  BE_4.trueLst = {104} and BE_4.falseLst = {103}.
- Next reduction is $mR_2 \rightarrow \varepsilon$. The attribute mR_2.nextInd $\leftarrow$ nextInd: 105.

Example

- Next 3-address codes are corresponding to BE_5:

  105 `if p == q goto` $\cdots$

  106 `goto` $\cdots$

- BE_5.trueLst $= \{105\}$ and BE_5.falseLst $= \{106\}$ and nextInd: 107.
- At reduction of BE_3 the BE_4.trueLst is backpatched by mR_2.nextInd $= 105$.

Example

- The code after the first backpatching:

```
100 if x < y goto ···
101 goto ···
102 $i ← b + c
103 if a > $i goto ···
104 goto 105
105 if p == q goto ···
106 goto ···
```

Note

A peephole optimization can replace goto 105 to noOp.

Example

- $BE_3.trueLst = BE_5.trueLst$: $\{105\}$ and $BE_3.falseLst = (BE_4.falseLst \cup BE_5.falseLst)$: $\{103, 106\}$.
- At reduction of BE_1 the $BE_2.falseLst$ is backpatched by $mR_1.nextInd = 102$.
- $BE_1.trueLst = \{100, 105\}$ and $BE_1.falseLst = \{103, 106\}$.

Example

- Modified code is

```
100 if x < y goto ···
101 goto 102
102 $i ← b + c
103 if a > $i goto ···
104 goto 105
105 if p == q goto ···
106 goto ···
```

Example: Note

It is clear that codes in sequence number 101 is also useless and can be replaced no-operation (noOp)

Example

- The modified code is

```
100 if x < y goto ···
101 nop
102 $i ← b + c
103 if a > $i goto ···
104 nop
105 if p == q goto ···
106 goto ···
```

Statements and Backpatching

We use backpatching for assignment statement, sequence of statements and flow-of-control statements. So the grammar is modified with marker non-terminals[a]

[a] One should be careful about doing that as in some cases the modified grammar may cease to be LALR.

Modified Grammar of Statements

$$
\begin{aligned}
\text{SL} &\rightarrow \text{SL mR S} \mid \text{S} \\
\text{S} &\rightarrow \text{AS} \\
&\rightarrow \text{if BE mR then SL kR else SL ;} \\
&\rightarrow \text{for mR BE mR do SL ;} \\
&\rightarrow \text{nop} \\
\text{mR} &\rightarrow \varepsilon \\
\text{kR} &\rightarrow \varepsilon
\end{aligned}
$$

Synthesized Attribute of a Statement

Every statement (S and SL) has a synthesized attribute nextLst. This is the list of indices of jump and branch instructions (unfilled) within the statement that transfer control to the 3-address instruction following the statement.

Backpatching: Statement List

$$SL \rightarrow SL_1\ mR\ S$$

$$\{fill(SL_1.nextLst,\ mR.nextInd)$$

$$SL.nextLst = S.nextLst\}$$

$$SL \rightarrow S$$

$$\{SL.nextLst = S.nextLst\}$$

Backpatching: Assignment, `nop` Statement and Marker

$$
\begin{aligned}
S &\rightarrow AS \ \{S.nextLst = nil\} \\
S &\rightarrow \texttt{nop} \ \{codeGen(\text{`nop'}) \\
&\qquad nextInd = nextInd{+}1 \\
&\qquad S.nextLst = nil\} \\
kR &\rightarrow \varepsilon \ \{kR.nextInd = nextInd \\
&\qquad codeGen(\text{`goto'} \cdots) \\
&\qquad nextInd = nextInd{+}1\}
\end{aligned}
$$

Backpatching: `if`-Statement

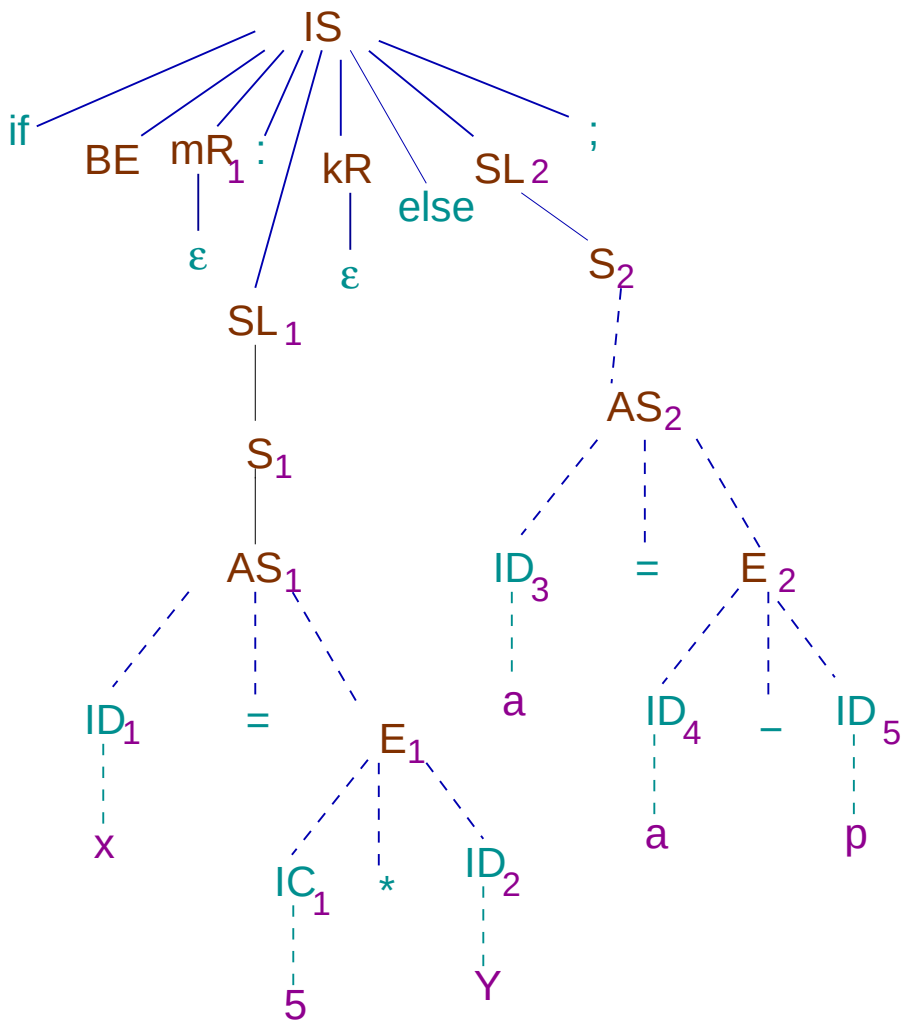
$$
\begin{aligned}
S \quad \rightarrow \quad & \text{if BE mR then } SL_1 \text{ kR else } SL_2 \ ; \\
& \{\text{fill(BE.trueLst, mR.nextInd)} \\
& \text{fill(BE.falseLst, kR.nextInd+1)} \\
& \text{temp} = \text{catLst}(SL_1.\text{nextLst}, \text{mkLst(kR.nextInd)}) \\
& S.\text{nextLst} = \text{catLst}(\text{temp}, SL_2.\text{nextLst}) \ \}
\end{aligned}
$$

Example

Consider the if-statement with the same boolean expression taken earlier as an example.

```
if x < y or not a > b + c and p == q then
   x = 5*y
else
   a = a - p
end
```

if-statement: Parse Tree



Example

We already know that the code corresponding to BE is as follows:

```
100 if x < y goto ···
101 nop
102 $i ← b + c
103 if a > $i goto ···
104 nop
105 if p == q goto ···
106 goto ···
```

BE.trueLst = {100, 105} and BE.falseLst = {103, 106} and nextInd: 107.

Example

- Next reduction is $mR_1 \rightarrow \varepsilon$. The attribute mR_1.nextInd $\leftarrow$ nextInd: 107.
- The code corresponding to SL_1 is

  107 `$(i+1) = 5 * y`

  108 `x = $(i+1)`
- The reduction of $kR_1 \rightarrow \varepsilon$ generates the code

  109 `goto` $\cdots$

  Its attribute is kR.nextLst $= \{109\}$

Example

- The code corresponding to SL_2 is

```
110 $(i+2) = a + p
111 a = $(i+2)
```

Example

The sequence of code and synthesized data at this point of compilation are

```
100 if x < y goto ···
101 nop
102 $i ← b + c
103 if a > $i goto ···
104 nop
105 if p == q goto ···
106 goto ···
107 $(i+1) = 5 * y
108 x = $(i+1)
109 goto ···
110 $(i+2) = a + p
111 a = $(i+2)
```

Example

- BE.trueLst $= \{100, 105\}$ and BE.falseLst $= \{103, 106\}$.
- mR_1.nextInd $= 107$.
- kR.nextLst $= \{109\}$
- SL_1.nextLst $= \text{SL}_2$.nextLst $=$ nil

Example

During the reduction to IS following actions are taken.

- Backpatch BE.trueLst with mR_1.nextInd.
- Backpatch BE.falseLst with kR.nextInd.
- ifStmt.nextLst = mkLst(kR.nextLst+1) as SL_1.nextLst = SL_2.nextLst = nil.

Example

Final sequence of code is

```
100 if x < y goto 107
101 nop
102 $i ← b + c
103 if a > $i goto 110
104 nop
105 if p == q goto 107
106 goto 110
107 $(i+1) = 5 * y
108 x = $(i+1)
109 goto ···
110 $(i+2) = a + p
111 a = $(i+2)
```

Backpatching: `for/while` Statement

Note that our `for` is nothing but `while`.

$S \rightarrow$ for mR_1 BE mR_2 do SL end

{fill(SL.nextLst, mR_1.nextInd)

fill(BE.trueLst, mR_2.nextInd)

S.nextLst = BE.falseLst

codeGen('goto', mR_1.nextInd) }

exitLoop Statement

- Our exit is similar to break is C language.
- We only consider necessary semantic actions and translation of exit in the context of a while-statement.
- We define an exit-list (extLst=Nil) after entering a while-loop.

exitLoop Statement

- At every exit, an unfilled '`goto -`' code is generated and its index is inserted in the exit-list.
- During the final reduction of the S → `while` ⋯, the exit-list is merged with the S.nextLst.

Modified Grammar of `while`

Grammar After First Modification

$$S \rightarrow \text{while mR BE mR : SL end}$$

$$mR \rightarrow \varepsilon$$

Grammar After Second Modification

$$S \rightarrow \text{while eR BE mR : SL end}$$

$$mR \rightarrow \varepsilon$$

$$eR \rightarrow \varepsilon$$

Semantic Actions for eR

$$\text{eR} \rightarrow \varepsilon$$

$$\{\ \text{eR.nextInd} = \text{nextInd}$$

$$\text{extLst} = \text{Nil}\}$$

Semantic Actions for eR

S → EXITLOOP

{ extLst = catLst(extLst, mkLst(nextInd))

codeGen('goto', -)

nextInd = nextInd + 1}

Backpatching Modified: `while` Statement

S → while eR BE mR : SL end

{fill(SL.nextLst, eR.nextInd)

fill(BE.trueLst, mR.nextInd)

S.nextLst = catLst(BE.falseLst,

extLst)

codeGen('goto', eR.nextInd) }

Note

- The exit-list can be maintained as a special label (say exit) in the symbol table.
- Nesting of loop will complicate the situation. In that case we use a stack to push exit-list headers of outer loops.

Note

- If a loop creates a local environment, the outer symbol-tables are pushed in a stack. If the exit-list is maintained on a symbol-table, it will automatically be stacked.