Semantic Actions and 3-Address Code Generation

Introduction

We start with different constructs of the given grammar (laboratory assignment-5) and discuss semantic actions and intermediate code generation. First we consider simple variable declaration.

Grammar of Simple Variable Declaration

$$
\begin{aligned}
\text{decl} &\rightarrow \text{def typeList end} \\
\text{typeList} &\rightarrow \text{typeList ; varList : type} \\
&\rightarrow \text{varList : type} \\
\text{varList} &\rightarrow \text{var , varList} \\
&\rightarrow \text{var} \\
\text{type} &\rightarrow \text{INT} \mid \text{FLOAT} \\
\text{var} &\rightarrow \text{ID}
\end{aligned}
$$

Synthesized Attributes

- Both var and varList have synthesized attribute locLst, a list of locations of the symbol table where identifiers are inserted and their type and other information are to be updated.
- The non-terminal type remembers the type of the list of variables in its synthesized attribute type.type.

Note

- In our simple case it is just `int` and `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, or it can be a structure with different types of fields.
- If the whole type information is available, its size etc. can be calculated.

Important Functions

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

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(l, type): updates type of the symbol-table locations from the list l using type.

Semantic Actions and Code Generation

$$
\begin{aligned}
\text{var} &\rightarrow \text{ID} \\
&\quad \{\text{temp} = \text{searchInsert}(\text{symTab}, \text{ID.lexme}, \text{err}) \\
&\quad \text{var.locLst} = \text{mkLocLst}(\text{temp})\} \\
\text{type} &\rightarrow \text{INT } \{\text{type.type} = \text{INT}\} \\
\text{type} &\rightarrow \text{FLOAT } \{\text{type.type} = \text{FLOAT}\}
\end{aligned}
$$

Semantic Actions and Code Generation

$$
\begin{aligned}
\text{varList} &\rightarrow \text{var , varList}_1 \\
&\quad \{\text{varList.locLst} = \text{catLocLst(var.locLst,} \\
&\qquad \text{varList}_1\text{.locLst)}\} \\
\text{varList} &\rightarrow \text{var} \\
&\quad \{\text{varList.locLst} = \text{var.locLst}\} \\
\text{typeList} &\rightarrow \text{varList : type} \\
&\quad \{\text{updateType(varLst.locLst, type.type)}\}
\end{aligned}
$$

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

$$
\begin{aligned}
\text{exp} &\rightarrow \text{exp} + \text{exp} \\
&\rightarrow \text{ID} \\
&\rightarrow \text{IC} \\
&\rightarrow \text{FC}
\end{aligned}
$$

We assume that ID is a simple scalar variable, IC is an integer constant and FC is a floating-point constant.

Synthesized Attributes

- An expression exp has the attribute exp.loc which is an index to the symbol table.
- The symbol table entry corresponding to exp.loc may be a program defined variable or a compiler generated variable.

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.

Important Functions

- The function newTemp() generates a compiler defined variable name. Its value is determined by the type of the expression being evaluated.

Semantic Actions and Code Generation

exp → ID

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

exp → IC

{exp.loc =

searchInsert(symTab, newTemp(), err)

updateType(mkLocLst(exp.loc), INT)}

Semantic Actions and Code Generation

exp → FC

{exp.loc =

searchInsert(symTab, newTemp(), FLOAT, err)

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

Semantic Actions and Code Generation

$$\text{exp} \rightarrow \text{exp}_1 + \text{exp}_2$$

{if type(exp_1.loc) = INT and
type(exp_2.loc) = INT then
exp.loc = searchInsert(symTab, newTemp(), err)
updateType(mkLocLst(exp.loc), INT)
codeGen(assIntPlus, exp_1.loc, exp_2.*loc*, exp.loc)

Semantic Actions and Code Generation

if type(symTab,exp_1.loc) = FLOAT

type(symTab, exp_2.loc) = FLOAT then

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

updateType(mkLocLst(exp.loc), FLOAT)

codeGen(assFltPlus, exp_1.loc, exp_2.*loc*, exp.loc)

Semantic Actions and Code Generation

if type(symTab,exp_1.loc) = INT

type(symTab, exp_2.loc) = FLOAT then

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

updateType(mkLocLst(temp),FLOAT)

codeGen(assignIntToFlt, exp_1.loc, temp)

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

updateType(mkLocLst(exp.loc),FLOAT)

codeGen(assFltPlus, temp, exp_2.*loc*, exp.loc)

Semantic Actions and Code Generation

if type(symTab,exp_1.loc) = FLOAT

type(symTab, exp_2.loc) = INT then

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

updateType(mkLocLst(temp),FLOAT)

codeGen(assignIntToFlt, exp_2.loc, temp)

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

updateType(mkLocLst(exp.loc),FLOAT)

codeGen(assFltPlus, exp_1.*loc*, temp, exp.loc) }

Grammar for Statements

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

Grammar Simple Assignment Statement

$$\text{assignmentStmt} \rightarrow \text{ID} := \text{exp}$$

We assume that ID is a simple scalar variable.

Semantic Actions and Code Generation

assignmentStmt

→ ID := exp

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

if type(temp) = NOTDEF then

ERROR

if (type(temp) = INT and type(exp.loc) = INT) or

(type(temp) = FLOAT and type(exp.loc) = FLOAT)

then

codeGen(assign, exp.loc, temp)

Semantic Actions and Code Generation

if (type(temp) = INT and type(exp.loc) = FLOAT) then

codeGen(assignFltToInt, exp.loc, temp)

if (type(temp) = FLOAT and type(exp.loc) = INT) then

codeGen(assignIntToFlt, exp.loc, temp) }

Flow-of-Control Statements

Our next consideration is flow-of-control statements. Here we use a technique known as backpatching to fill the jump/branch addresses.

Backpatching in Flow-of-Control Statements

- Boolean expressions and flow-of-control statements require branch instructions.
- Often 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 inherited attribute.

Backpatching in Flow-of-Control Statements

- As the target instruction has not yet been generated, it is necessary to bind the label afterward.
- Backpatching is an alternate approach where the targets of codes corresponding to branch/jump instructions are kept unfilled.
- List of these unfilled codes are passed as synthesized attributes.

Backpatching in Flow-of-Control Statements

- Holes in these 3-address codes will be filled (backpatched) when the target label is generated.
- This does not require a second pass of associating labels to the targets.
- We modify our grammar of Boolean expression and flow-of-control statements as follows.

Modified Grammar of Boolean Expression

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

Synthesized Attributes

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

Synthesized Attributes

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

Sequence Number of an Instructions

- There is a sequence number or index of every instruction. These indices are used as labels.
- Following are a few useful functions for semantics actions.

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 indexed by the elements of 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

$$
\begin{aligned}
\text{bExp} \rightarrow\ & \text{exp}_1\ \text{relOP}\ \text{exp}_2 \\
& \{\text{bExp.trueLst} = \text{mkLst}(\text{nextInd}) \\
& \text{bExp.falseLst} = \text{mkLst}(\text{nextInd+1}) \\
& \text{codeGen}(\text{`if relOP'}, \text{exp}_1\text{.loc}, \\
& \qquad \text{exp}_2\text{.loc}, \text{`goto'} \cdots) \\
& \text{codeGen}(\text{`goto'} \cdots) \\
& \text{nextInd} = \text{nextInd+2}\ \}
\end{aligned}
$$

Semantic Actions and Code Generation

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

Semantic Actions and Code Generation

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

Semantic Actions and Code Generation

$$\begin{aligned} \text{bExp} \rightarrow\ & \text{not bExp}_1 \\ & \{\text{bExp.falseLst} = \text{bExp}_1\text{.trueLst} \\ & \text{bExp.trueLst} = \text{bExp}_1\text{.falseLst}\ \} \end{aligned}$$

Semantic Actions and Code Generation

$$\text{bExp} \rightarrow (\ \text{bExp}_1\ )$$

$$\{\text{bExp.falseLst} = \text{bExp}_1\text{.falseLst}$$

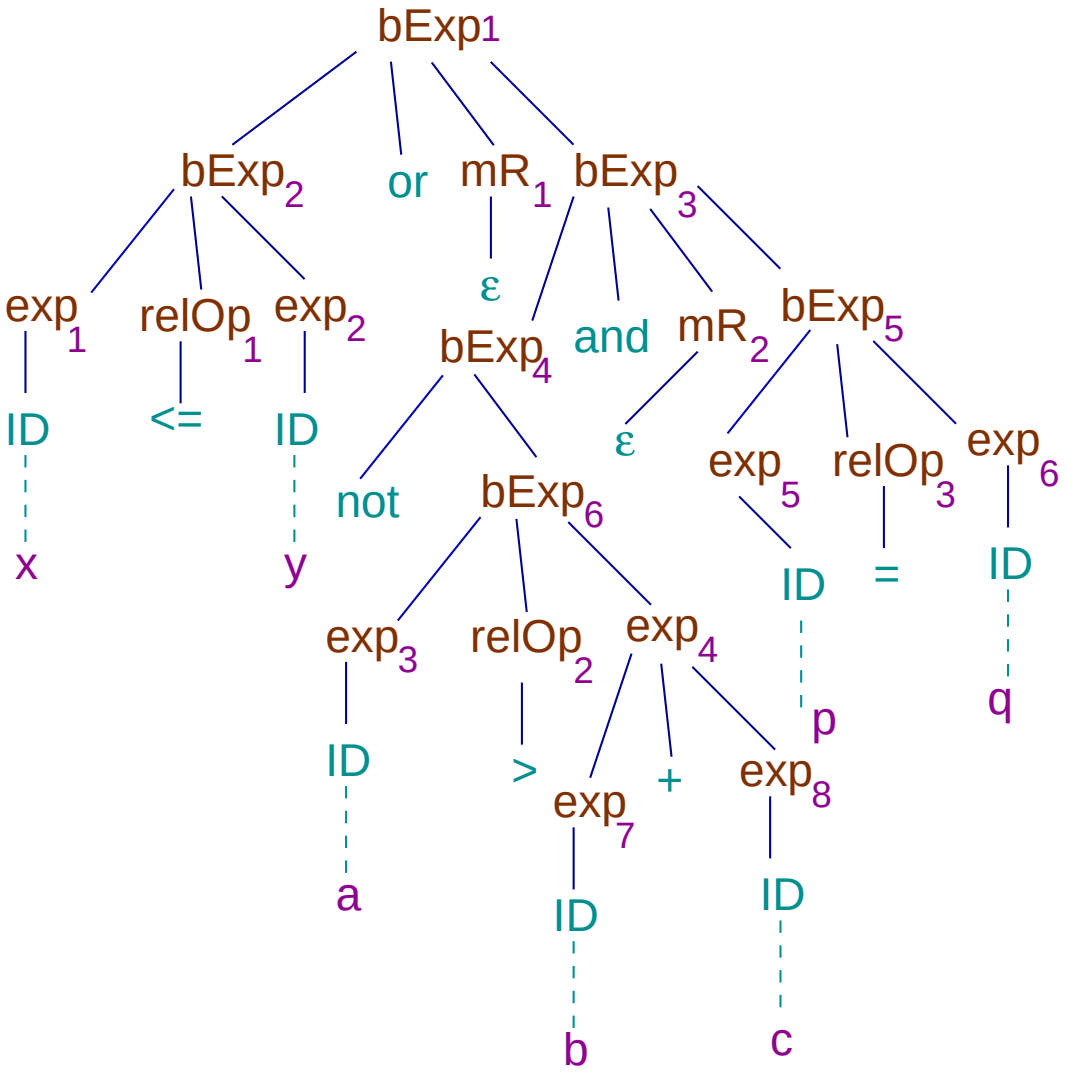
$$\text{bExp.trueLst} = \text{bExp}_1\text{.trueLst}\ \}$$

Example

Consider the Boolean expression

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

Boolean Expression: Parse Tree



Example

- Let the next index of the 3-address code (nextInd) sequence be 100.
- The 3-address codes corresponding to $bExp_2$ in readable form is

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

  101 `goto` $\cdots$
- $bExp_2$.TrueLst = {100} and $bExp_2$.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 bExp_6 is

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

  104 `goto` $\cdots$

Example

- $\text{bExp}_6.\text{TrueLst} = \{103\}$ and $\text{bExp}_6.\text{FalseLst} = \{104\}$ and nextInd: 105.
- The not operator flips the lists.
  $\text{bExp}_4.\text{TrueLst} = \{104\}$ and $\text{bExp}_4.\text{FalseLst} = \{103\}$.
- Next reduction is $\text{mR}_2 \rightarrow \varepsilon$. The attribute $\text{mR}_2.\text{nextInd} \leftarrow$ nextInd: 105.

Example

- Next 3-address codes are corresponding to bExp_5:

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

  106 `goto` $\cdots$

- bExp_5.TrueLst $= \{105\}$ and bExp_5.FalseLst $= \{106\}$ and nextInd: 107.

- At reduction of bExp_3 the bExp_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 ···
```

Example

- $\text{bExp}_3.\text{TrueLst} = \text{bExp}_5.\text{TrueLst}$: $\{105\}$ and $\text{bExp}_3.\text{FalseLst} = (\text{bExp}_4.\text{FalseLst} \cup \text{bExp}_5.\text{FalseLst})$: $\{103, 106\}$.
- At reduction of bExp_1 the $\text{bExp}_2.\text{falseLst}$ is backpatched by $\text{mR}_1.\text{nextInd} = 102$.
- $\text{bExp}_1.\text{TrueLst} = \{100, 105\}$ and $\text{bExp}_1.\text{FalseLst} = \{103, 106\}$.

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 numbers 101 and 104 are useless. We replace them by no-operations (nop)

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

$$
\begin{aligned}
\text{stmtList} &\rightarrow \text{stmtList mR ; stmt} \mid \text{stmt} \\
\text{stmt} &\rightarrow \text{assignmentStmt} \\
&\rightarrow \text{if bExp mR : stmtList kR elsePart end} \\
&\rightarrow \text{while mR bExp mR : stmtList end} \\
\text{elsePart} &\rightarrow \text{else mR stmtList} \mid \varepsilon \\
\text{mR} &\rightarrow \varepsilon \\
\text{kR} &\rightarrow \varepsilon
\end{aligned}
$$

Synthesized Attribute of a Statement

Every statement (stmt and stmtList) 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, the next-statement or continuation.

Backpatching: Statement List

$$\begin{aligned}
\text{stmtList} &\rightarrow \text{stmtList}_1\ \text{mR}\ ;\ \text{stmt} \\
&\quad \{\text{fill}(\text{stmt}_1.\text{nextLst},\ \text{mR.nextInd}) \\
&\quad \text{stmtList.nextLst} = \text{stmt.nextLst}\} \\
\text{stmtList} &\rightarrow \text{stmt} \\
&\quad \{\text{stmtList.nextLst} = \text{stmt.nextLst}\}
\end{aligned}$$

Backpatching: Assignment Statement and Marker

$$
\begin{aligned}
\text{stmt} &\rightarrow \text{assignmentStmt} \\
&\quad \{\text{stmt.nextLst} = \text{nil}\} \\
\text{kR} &\rightarrow \varepsilon \\
&\quad \{\text{kR.nextLst} = \text{mkLst}(\text{nextInd}) \\
&\quad \text{codeGen}(\text{`goto'} \cdots) \\
&\quad \text{nextInd} = \text{nextInd}+1\}
\end{aligned}
$$

Backpatching: `if`-Statement

```
stmt → if bExp mR : stmtList kR elsePart end
       {fill(bExp.trueLst, mR.nextInd)
       if(elsePart.nextInd == −1)
           stmt.nextLst = catLst(bExp.falseLst,
                                 stmtList.nextLst)
       else
           fill(bExp.falseLst, elsePart.nextInd)
           stmt.nextLst = catLst(stmtList.nextLst,
               catLst(kR.nextLst, elsePart.nextLst))}
```

Backpatching: `else` Part

elsePart → else mR stmtList

{elsePart.nextInd = mR.nextInd

elsePart.nextLst = stmtList.nextLst }

elsePart → ε

{elsePart.nextInd = -1

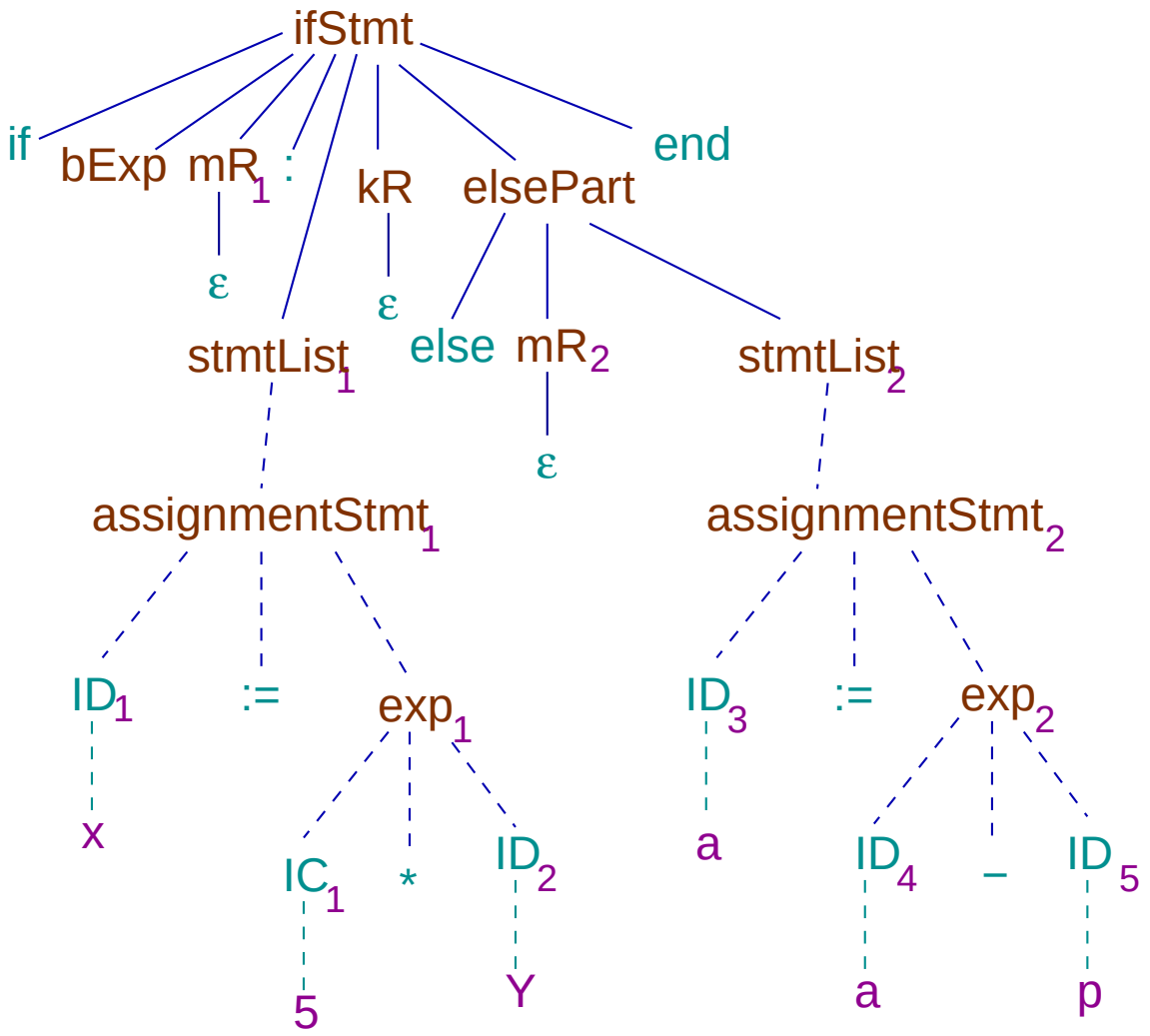
elsePart.nextLst = nil }

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:
   x := 5*y
else
   a := a - p
end
```

if-statement: Parse Tree



Example

We already know that the code corresponding to bExp 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 ···
```

bExp.TrueLst = {100, 105} and bExp.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 $stmtList_1$ 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 reduction of $\text{mR}_2 \rightarrow \varepsilon$ synthesize the attribute $\text{mR}_2.\text{nextInd} \leftarrow 110$.
- The code corresponding to stmtList_2 is

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

  111 `a = $(i+2)`
- Reduction to elsePart copies $\text{stmtNextLst}_2 =$ nil to elsePart.nextLst.

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

- bExp.TrueLst $= \{100, 105\}$ and bExp.FalseLst $= \{103, 106\}$.
- mR_1.nextInd $= 107$.
- kR.nextLst $= \{109\}$
- elsePart.nextInd $= \text{mR}_2$.nextInd $= 110$.
- stmtList_1.nextLst $=$ elsePart.nextLst $=$ nil

Example

During the reduction to ifStmt following actions are taken.

- Backpatch bExp.TrueLst with mR_1.nextInd.
- Backpatch bExp.FalseLst with elsePart.nextInd.
- ifStmt.nextLst = kR.nextLst as $stmtList_1$.nextLst = elsePart.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: `while` Statement

$$
\begin{aligned}
\text{stmt} \rightarrow\ & \text{while } \text{mR}_1 \text{ bExp } \text{mR}_2 : \text{stmtList end} \\
& \{\text{fill}(\text{stmtList.nextLst}, \text{mR}_1\text{.nextInd}) \\
& \text{fill}(\text{bExp.trueLst}, \text{mR}_2\text{.nextInd}) \\
& \text{stmt.nextLst} = \text{bExp.falseLst} \\
& \text{codeGen}(\text{`goto'}, \text{mR}_1\text{.nextInd}) \ \}
\end{aligned}
$$

Loop Statement Grammar

The grammar rule for our loopStmt is as follows:

$$\begin{aligned} \text{loopStmt} &\rightarrow \text{from ID} := \text{exp}_1 \text{ to } \text{exp}_2 \text{ stepPart} \\ &\qquad\qquad : \text{stmtListO end} \\ \text{stepPart} &\rightarrow \text{step } \text{exp}_3 \mid \varepsilon \end{aligned}$$

Loop Statement

For simplicity we assume that ID is a scalar variable. We also assume, again to make life simple, that type of all expressions are integers. The informal semantics of this deterministic loop[a] is as follows:

[a] Unlike the `for`-loop of C/C++.

Semantics of Loop Statement

- `ID` will be assigned the value of $\texttt{exp}_1$.
- The number of iterations of the loop is determined at the beginning. Compiler defines a loop count variable lC. Its value is evaluated as

  $$\text{lC} = (\text{exp}_2 - \text{exp}_1)/\text{exp}_3 + 1$$

Semantics of Loop Statement

- The body of the loop, stmtListO, is computed if lc ≥ 0.
- After every iteration ID is incremented by exp_3, the loop count lC is decremented by 1, and the control is transferred to the beginning of the test for lC.
- The default value of exp_3 is 1.

Stages of Semantic Actions

- Initialization of ID requires evaluation of $\exp_1$.
- Computation and initialization of lC requires evaluation up to $\exp_3$.
- Both initializations are before iterations. But test of lc ≥ 0 and decrement of lCis part of each iteration.

In view of these, we modify the grammar as follows:

Modified Loop Statement

$$
\begin{aligned}
\text{loopStmt} &\rightarrow \text{from loopCount lM : mR} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{stmtListO end} \\
\text{loopCount} &\rightarrow \text{ID} := \text{exp}_1 \text{ to exp}_2 \text{ stepPart} \\
\text{stepPart} &\rightarrow \text{step exp} \\
\text{stepPart} &\rightarrow \varepsilon \\
\text{mR} &\rightarrow \varepsilon \\
\text{lM} &\rightarrow \varepsilon
\end{aligned}
$$

Semantic Actions for stepPart

- The non-terminal stepPart should save the symbol table entry of the internal variable of step exp.
- If exp is absent, we use an indicator -1.

$$\text{stepPart} \rightarrow \text{step exp} \quad \{\text{stepPart.loc} = \text{exp.loc}\}$$

$$\text{stepPart} \rightarrow \varepsilon \quad \{\text{stepPart.loc} = -1\}$$

Synthesized Attributes of loopCount

The non-terminal loopCount saves the following information in its synthesized attributes.

- Location of ID - loopCount.id
- Location of the iteration count of the loop - loopCount.count
- Location of the iteration step (or -1) - loopCount.step

Semantic Actions for loopCount

loopCount

→ ID := exp_1 to exp_2 stepPart

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

codeGen(assign, exp_1.loc, temp)

temp1 = searchInsert(symTab, newtemp(), err)

updateType(temp1, INT)

codeGen(assignSubInt, exp_2.loc, exp_1.loc, temp1)

temp2 = searchInsert(symTab, newtemp(), err)

updateType(temp2, INT)

Semantic Actions for loopCount

```
if (stepPart.loc = -1) then
    codeGen(assignInt, temp1, temp2)
else
    codeGen(assignDivInt, temp1, stepPart.loc, temp2)
temp3 = searchInsert(symTab, newtemp(), err)
updateType(temp3, INT)
codeGen(assignAddIntConst, temp2, 1, temp3)
```

Semantic Actions for loopCount

loopCount.id = temp

loopCount.count = temp3

loopCount.step = stepPart.loc}

Synthesized Attributes of lM

- The non-terminal lM generates code to decide whether to enter in the loop or exit. So it has two synthesized attributes, lM.trueLst and lM.falseLst to be backpatched later.
- It also remembers the index of the first instruction of the test code. This index will be used to backpatch the nextLst of stmtListO.

Semantic Actions for lM

lM $\rightarrow$ ε

{lM.nextInd = nextInd

lM.trueLst = mkLst(nextInd)

lM.falseLst = mkLst(nextInd+1)

codeGen(ifGEZero, loopCount.count, -)

codeGen('goto', -)

nextInd = nextInd+2}

Note

When the reduction of lM $\rightarrow \varepsilon$ takes place, the non-terminal loopCount is available immediately below it in the value stack. So loopCount.count can be accessed.

Semantic Actions for loopStmt

loopStmt → from loopCount lM : mR stmtListO end

{codeGen(assignPlusInt, loopCount.id,
loopCount.step, loopCount.id)
codeGen(assignAddIntConst, loopCount.count, -1,
loopCount.count)
codeGen('goto', lM.nextInd)
fill(lM.trueLst, mR.nextInd)
fill(stmtListO.nextLst, lM.nextInd)
loopStmt.nextLst = lM.falseLst }

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 stmt $\rightarrow$ `while` $\cdots$, the exit-list is merged with the stmt.nextLst.

Modified Grammar of `while`

Grammar After First Modification

$$\begin{aligned} \text{stmt} &\rightarrow \text{while mR bExp mR : stmtList end} \\ \text{mR} &\rightarrow \varepsilon \end{aligned}$$

Grammar After Second Modification

$$\begin{aligned} \text{stmt} &\rightarrow \text{while eR bExp mR : stmtList end} \\ \text{mR} &\rightarrow \varepsilon \\ \text{eR} &\rightarrow \varepsilon \end{aligned}$$

Semantic Actions for eR

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

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

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

Semantic Actions for eR

stmt → EXITLOOP

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

codeGen('goto', -)

nextInd = nextInd + 1}

Backpatching Modified: `while` Statement

stmt → while eR bExp mR : stmtList end

{fill(stmtList.nextLst, eR.nextInd)

fill(bExp.trueLst, mR.nextInd)

stmt.nextLst = catLst(bExp.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.

Grammar of Array Declaration

$$
\begin{aligned}
\text{decl} &\rightarrow \text{def typeList end} \\
\text{typeList} &\rightarrow \text{typeList ; varList : type} \\
&\rightarrow \text{varList : type} \\
\text{varList} &\rightarrow \text{var , varList} \\
&\rightarrow \text{var} \\
\text{type} &\rightarrow \text{INT} \mid \text{FLOAT}
\end{aligned}
$$

Grammar of Array Declaration

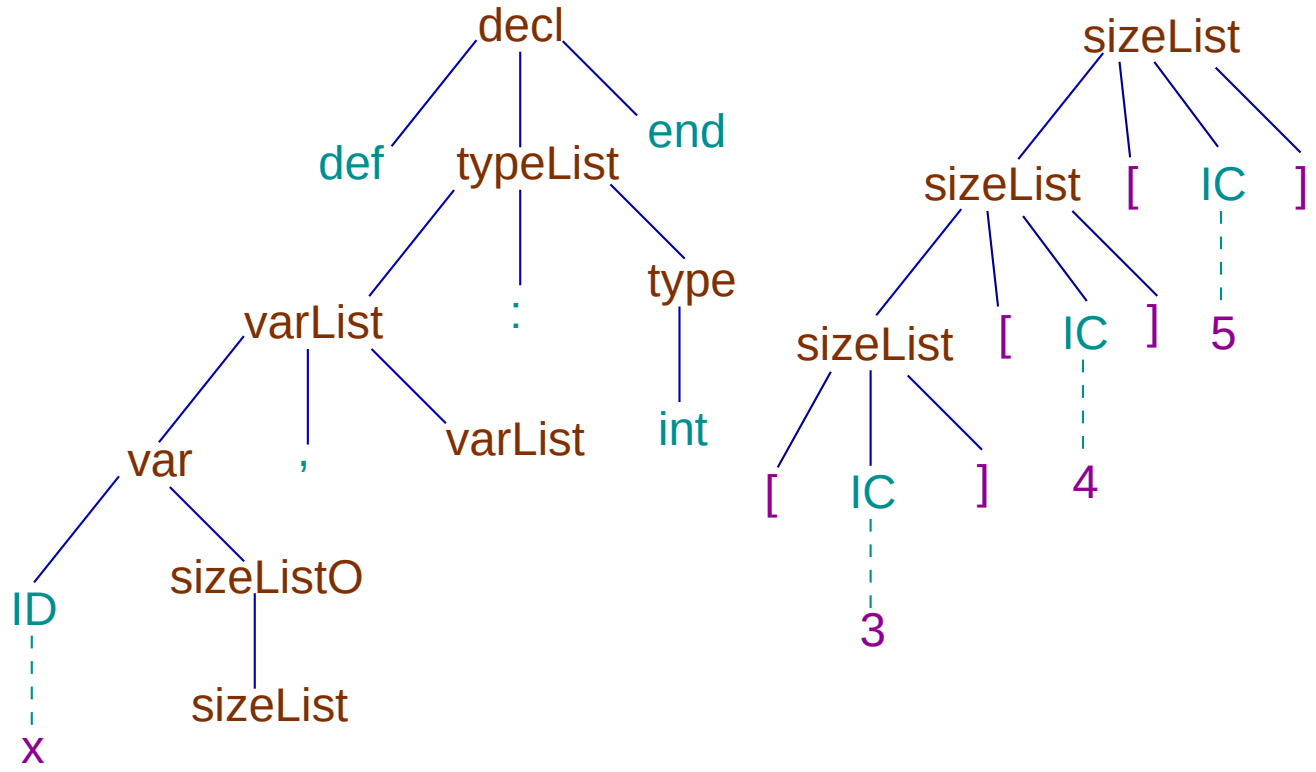
$$
\begin{aligned}
\text{var} &\rightarrow \text{ID sizeListO} \\
\text{sizeListO} &\rightarrow \text{sizeList} \\
\text{sizeList} &\rightarrow \text{sizeList [INT\_CONST]} \\
&\rightarrow \text{[INT\_CONST]}
\end{aligned}
$$

Array Declaration

A typical array deceleration is as follows:

```
def
    ...
    x[3][4][5] :  int ;
    ...
end
```

Array Declaration: Parse Tree



Information in Symbol Table

- Array `int x[3][4][5]` may be viewed as follows:
- A 3-element array of 4-element array of 5-element array of base type int.
- Important information are base type, range of each dimension and the total size in bytes.

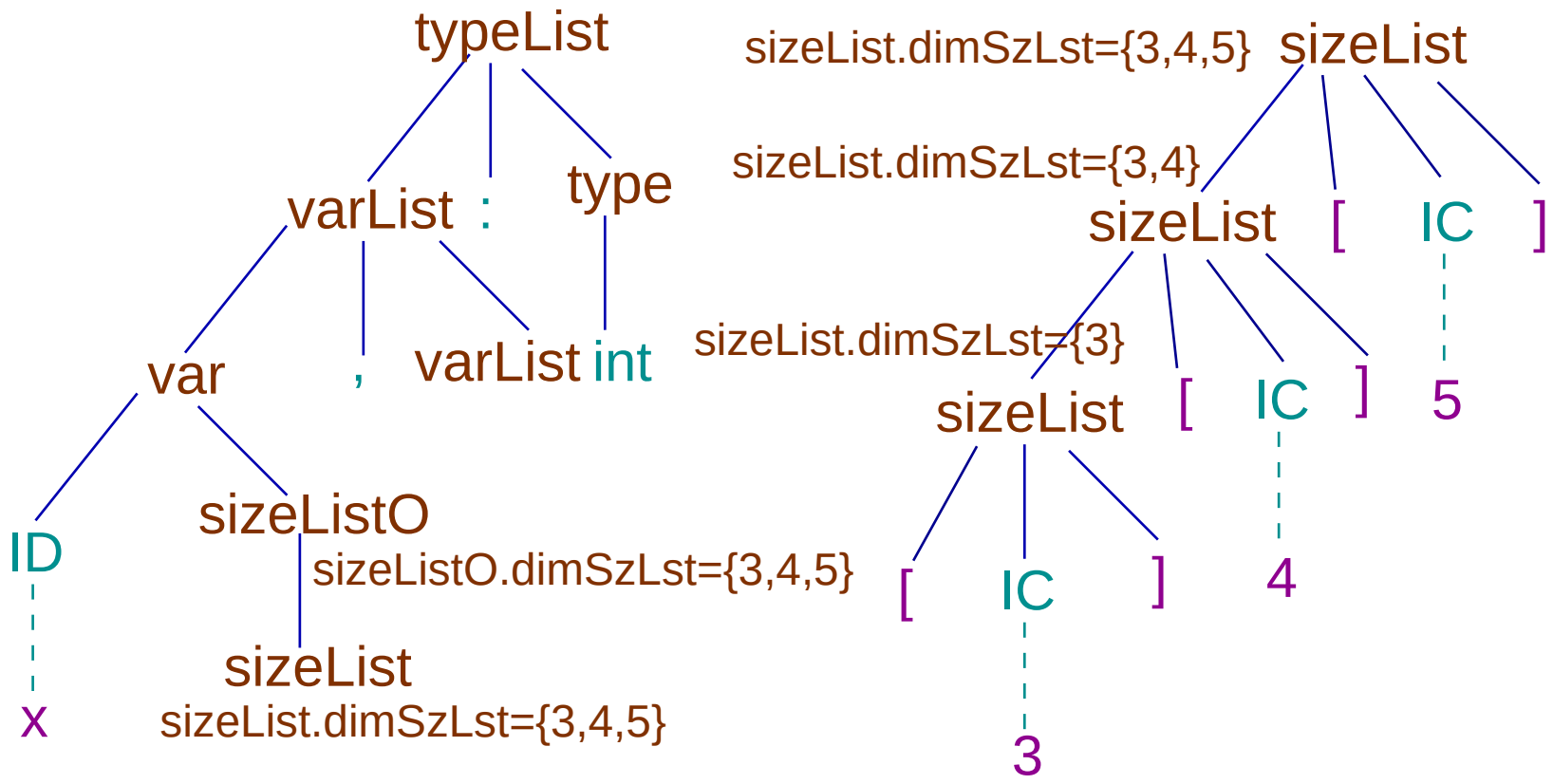
Note

- In some programming languages the upper and lower bounds of each dimension can be specified.
- More information such as lower bound and upper bound of of indices is necessary to save in such a situation.

Synthesis of Attributes and Semantic Actions

- The non-terminal sizeList and sizeListO maintains the list of sizes (dimSzLst).
- This list may be put in the symbol table during the reduction of `var` $\rightarrow$ `ID sizeListO`.
- The base type and displacement (depends on the total size) are updated during the reduction `typeList` $\rightarrow$ `varList : type`.

Array Declaration: Decorated Parse Tree



Array Expression and Assignment

- An array element may be present in an expression or a value may be assigned to an array element.

  `x[`e_1`][`e_2`] := exp`

  `a := ` $\cdots$ `x[`e_1`][`e_2`]` $\cdots$

- In both the cases it is necessary to compute the offset of the element from the base of the array.

Offset Computation: an Example

- We consider a 3-D array of base type int: `x[3][5][7] : int`.
- The array is stored in the memory in row-major order.
- Let the address of the `x[0][0][0]` (starting address) be x_a; and the size of `int` be w.
- The address of `x[i][j][k]` is $x_a + (((i \times 5 + j) \times 7) + k) \times w$

Note

Essential information to compute the offset of `x[i][j][k]` are starting address x_a, values of three indices i, j, k, the sizes of the second and the third dimensions, 5, 7 respectively, and size of the base type.

Offset Computation: an Example

- If the array is stored in column-major order, the address of `x[i][j][k]` is $x_a + (((k \times 5 + j) \times 3) + i) \times w$. Here the sizes of the first and the second dimensions are useful for offset computation.
- In both the cases we assume that when the range of a dimension [n] is specified, the indices are $0, \cdots, n-1$.

Offset Computation: an Example

- In some languages the ranges of indices of different dimensions are given explicitly, e.g. `int x[1-3][2-5][3-7]`, where possible values of first indices are 1,2,3; second indices are 2,3,4,5; and third indices are 3,4,5,6,7.
- In row-major storage the address of `x[i][j][k]` is $x_a + ((((i-1) \times (5-2+1) + (j-2)) \times (7-3+1)) + (k-3)) \times w$, where x_a is the address of `x[1][2][3]`.

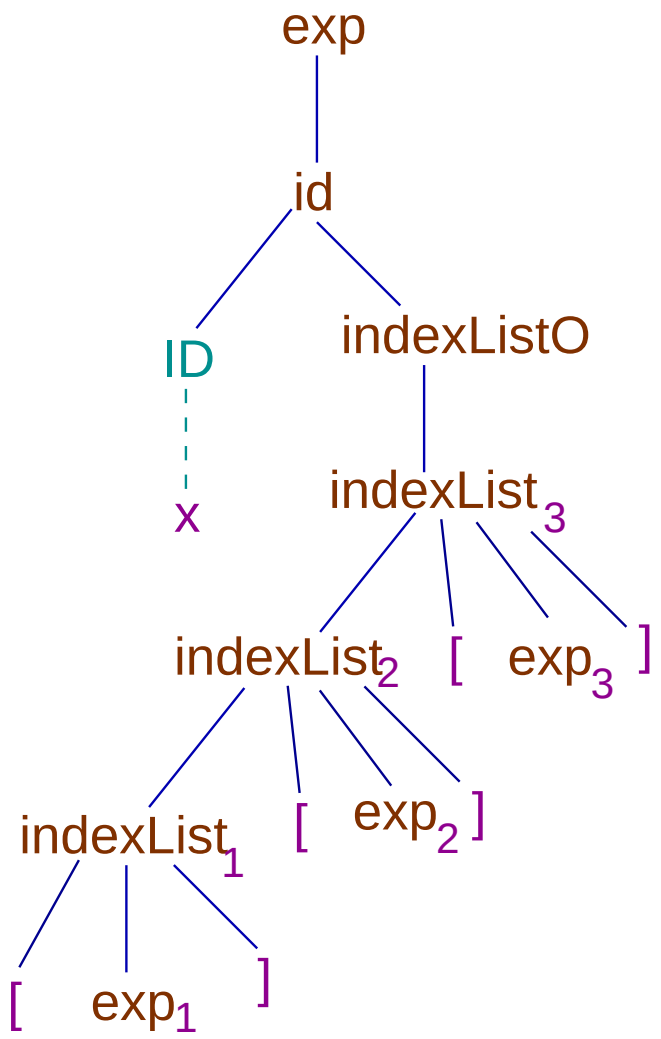
Offset Computation: an Example

- Let $s_2 = 5 - 2 + 1$ and $s_3 = 7 - 3 + 1$ be the sizes of second and third dimensions.
- The expression can be rewritten as $x_a - ((((1 \times s_2 + 2) \times s_3) + 3) \times w) + ((((i \times s_2 + j) \times s_3) + k) \times w)$.
- The first two terms are independent of (i, j, k). In a nested loop they can be computed outside it.

Grammar of Array in Expression and Assignment

$$
\begin{aligned}
\text{id} &\rightarrow \text{ID indexListO} \\
\text{indexListO} &\rightarrow \text{indexList} \\
\text{indexListO} &\rightarrow \varepsilon \\
\text{indexList} &\rightarrow \text{indexList [exp]} \\
\text{indexList} &\rightarrow \text{[exp]}
\end{aligned}
$$

Array in Expression: Parse Tree



Note

- Each expression has an attribute exp.loc, an index of the symbol table corresponding to a variable.
- The symbol-table entry of the array identifier has the sizes of different dimensions. But it is not available during the reduction of [exp] to indexList or indexList [exp] to indexList[a]

[a] Though it is available immediately below the handle in the stack.

Synthesized Attributes and Semantic Actions

- Both indexList and indexListO has synthesized attributes locLst that carries list of symbol-table indices corresponding to the expressions.
- The computation of offset takes place during the reduction of ID indexListO to id.
- The non-terminal id may have two attributes, id.base and id.offset.

Code Generation

- Let array deceleration be
  `x[`r_1`][`r_2`]` $\cdots$ `[`r_k`]:int`.
- Let the use of an array in an expression is
  `x[`exp_1`][`exp_2`]` $\cdots$ `[`exp_k`]`
- Let the base address of the array be x_b.
- Let the width of the base type be w.

Code Generation

Note that
indexListO.locLst = $\{exp_1.loc, \cdots, exp_k.loc\}$.
The address computation of the array element and the semantic actions corresponding to the reduction
id $\rightarrow$ ID indexListO
is as follows:
temp1 = searchInsert(symTab, newTemp(), err)
updateType(mkLocLst(temp1), ADDR)
codeGen(assign, exp_1.loc, temp1)
$\$_{j+1} = exp_1.loc$

Code Generation

for $i = 1$ to $k - 1$ do

temp2 = searchInsert(symTab, newTemp(), err)

updateType(mkLocLst(temp2), ADDR)

codeGen(assAddrMultConst, temp1, r_{i+1}, temp2)

$\$_{j+2i} = \$_{j+2i-1} \times \mathrm{r}_{i+1}$

temp1 = searchInsert(symTab, newTemp(), err)

updateType(mkLocLst(temp1), ADDR)

codeGen(assAddrAdd, temp2, exp_{i+1}, temp1)

$\$_{j+2i+1} = \$_{j+2i} + \mathrm{exp}_{i+1}$

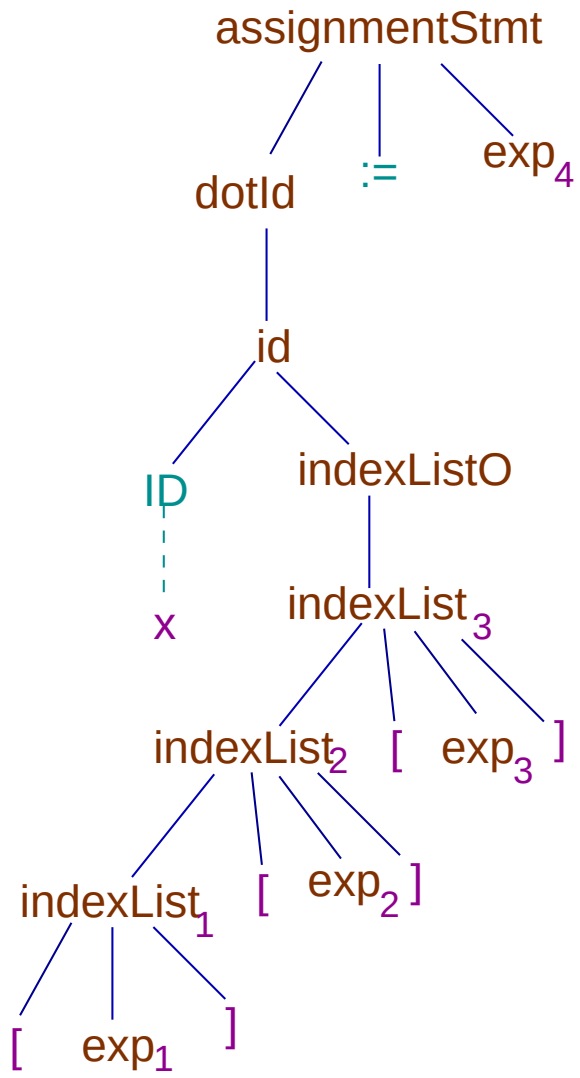
Code Generation

temp2 = searchInsert(symTab, newTemp(), err)
updateType(mkLocLst(temp1), ADDR)
codeGen(assAddrMultConst, temp1, w, temp2)
$\$_{j+2k} = \$_{j+2k-1} \times w$
id.base = searchInsert(symTab,ID.lexme,err).sTab.offset
id.offset = temp2

Code Generation

The 3-address code corresponding to exp $\rightarrow$ id is,
temp = searchInsert(symTab, newTemp(), err)
updateType(mkLocLst(temp1), ADDR)
codeGen(assAddrAddConst, id.base, id.offset, temp)
$\$_{j+2k+1} = \$_{j+2k} + x_b$
temp1 = searchInsert(symTab, newTemp(), err)
updateType(mkLocLst(temp1), TYPE)
codeGen(assignIndirFrm, temp, temp1)
$\$_{j+2k+2} = {}^*\$_{j+2ki+1}$

Array in Assignment: Parse Tree



Synthesized Attributes and Semantic Actions

- The semantic actions upto id are identical.
- The non-terminal dotId will have attributes of id i.e. dotId.base and dotId.offset.
- The value of dotId.base + dotId.offset is computed. The location corresponding to this address is indirectly assigned exp.loc.

Code Generation

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

updateType(mkLocLst(temp), ADDR)

codeGen(assAddrPlus, dotId.base, dotId.offset, temp)

codeGen(assIndirTo, exp_4.loc, temp)

Grammar of Function Declaration

$$
\begin{aligned}
\text{decl} &\rightarrow \text{fun funDef end} \\
\text{funDef} &\rightarrow \text{funID fparamListO -> type} \\
&\qquad\qquad\qquad\qquad\qquad\quad \text{funBody} \\
\text{funID} &\rightarrow \text{ID} \\
\text{fparamListO} &\rightarrow \text{fparamList} \\
\text{fparamListO} &\rightarrow \varepsilon
\end{aligned}
$$

Grammar of Function Declaration

$$
\begin{aligned}
\text{fparamList} &\rightarrow \text{fparamList ; pList : type} \\
\text{fparamList} &\rightarrow \text{pList : type} \\
\text{pList} &\rightarrow \text{pList , idP} \\
\text{pList} &\rightarrow \text{idP} \\
\text{idP} &\rightarrow \text{ID sizeListO} \\
\text{funBody} &\rightarrow \text{declList stmtListO}
\end{aligned}
$$

Note

- We may rewrite the rule

  funDef → funID fparamListO -> type funBody as

  funDef → funHeader funBody

  funHeader → funID fparamListO -> type

- The name of the function and its type information, an ordered list of return type and types of formal parameters, can be inserted in the current symbol table during the reduction to funHeader.

Note

- It is necessary to save the current symbol table (ct) (pointer to it) in a stack and create a new symbol table (nt) for the new environment of the function.
- There is a link from the the function name entry in ct to the new table nt.

Note

- It is also necessary to insert the formal parameter names and their types in the new symbol-table (nt) as they will be used as variables during the translation of the function body.

Grammar of Function Call

$$
\begin{aligned}
\text{callStmt} &\rightarrow \text{(ID : actParamListO)} \\
\text{exp} &\rightarrow \text{(ID : actParamListO)} \\
\text{actParamList} &\rightarrow \text{actParamList , exp} \\
\text{actParamList} &\rightarrow \text{exp}
\end{aligned}
$$

Note

- Corresponding to every reduction to actParamList the following three address code may be generated.
  codeGen(param, exp.loc)
- But we shall delay the generation of this code due to several reasons.

Note

- It is necessary to check type equivalence of actual and formal parameters. It may also be necessary to write code for type conversion. But none of these can be easily done during the reduction to actParamLst.
- Moreover we want to group all actual parameter codes together, without mixing them with the code to evaluate expressions.

Note

- So we save the list of locations of exp's as synthesized attribute of actParamList.
- Finally during the reduction to exp or callStmt, a sequence of
  codeGen(param, exp.loc)
  3-address codes are emitted.

Note

- Actual function call will be made during the reduction to exp or the callStmt.
- The 3-address code in case of reduction to callStmt is
  codeGen(call, temp, count),
  where temp is the symbol-table index corresponding to the function name and count is the number of actual parameters.

Note

- The 3-address code corresponding to the reduction to exp is slightly different. A new variable name is created and inserted in the symbol table with its type information etc. The code is
  codeGen(assCall, temp, count, $temp_1$),
  where $temp_1$ is the index of the symbol-table corresponding to the new variable.

Parameter Passing

Our discussion on parameter passing assumes call-by-value. We have not talked about call-by-reference, call-by-name, call-by-need, etc.

Code for Structure

We have not talked about code generation for structure or record declaration and access.

Switch Statement

There is no switch statement in our language. But what are the possible translation mechanisms of such a statement?