Semantic Actions and 3-Address Code Generation

Introduction

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

Grammar of Simple Variable Declaration

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

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.

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 need to store the size and the offset of the memory location from a base address.

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.

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

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

$$TY \rightarrow \text{int } \{TY.type = INT\}$$

$$TY \rightarrow \text{real } \{TY.type = FLOAT\}$$

$$VL \rightarrow \text{id}$$

$$\{ \text{ temp} = \text{searchInsert(symTab, id.lexme, err)}$$

$$\text{VL.locLst} = \text{mkLocLst(temp)} \}$$

$$VL \rightarrow VL_1 \text{ , id}$$

$$\{ \text{ temp} = \text{searchInsert(symTab, id.lexme, err)}$$

$$\text{VL.locLst} = \text{catLocLst}(\text{VL}_1\text{.locLst, mkLocLst(temp)}) \}$$

$$D \rightarrow TY\ VL\ \{ \text{ updateType(symTab, VL.locLst, TY.type} \}$$

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

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

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

$$E \rightarrow \mathrm{id}$$

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

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

$$E \rightarrow \mathrm{ic}$$

{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

$E \rightarrow \text{fc}$

{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

$$AS \rightarrow id = E$$

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 are the statements that control the flow of execution. Here we use a technique known as backpatching to fill the jump/branch addresses.

Backpatching in 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 inherited attribute.

Backpatching in Control Flow Statements

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

We use or, and and not for clarity.

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

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

BE $\rightarrow$ exp_1 relOP exp_2

{BE.trueLst = mkLst(nextInd)

BE.falseLst = mkLst(nextInd+1)

codeGen('if relOP', exp_1.loc,

exp_2.loc, 'goto' $\cdots$)

codeGen('goto' $\cdots$)

nextInd = nextInd+2 }

Semantic Actions and Code Generation

$$BE \rightarrow BE_1 \text{ or } mR\ BE_2$$

$$\{\text{fill}(BE_1.\text{falseLst}, mR.\text{nextInd})$$

$$BE.\text{trueLst} = \text{catLst}(BE_1.\text{trueLst}, BE_2.\text{trueLst})$$

$$BE.\text{falseLst} = BE_2.\text{falseLst}\ \}$$

Semantic Actions and Code Generation

$$BE \rightarrow BE_1 \text{ and } mR\ BE_2$$

$$\{\text{fill}(BE_1.\text{trueLst}, mR.\text{nextInd})$$

$$BE.\text{falseLst} = \text{catLst}(BE_1.\text{falseLst}, BE_2.\text{falseLst})$$

$$BE.\text{trueLst} = BE_2.\text{trueLst}\ \}$$

Semantic Actions and Code Generation

$$BE \rightarrow \text{not } BE_1$$

$$\{BE.\text{falseLst} = BE_1.\text{trueLst}$$

$$BE.\text{trueLst} = BE_1.\text{falseLst} \}$$

Semantic Actions and Code Generation

$$BE \rightarrow (BE_1)$$

$$\{BE.falseLst = BE_1.falseLst$$

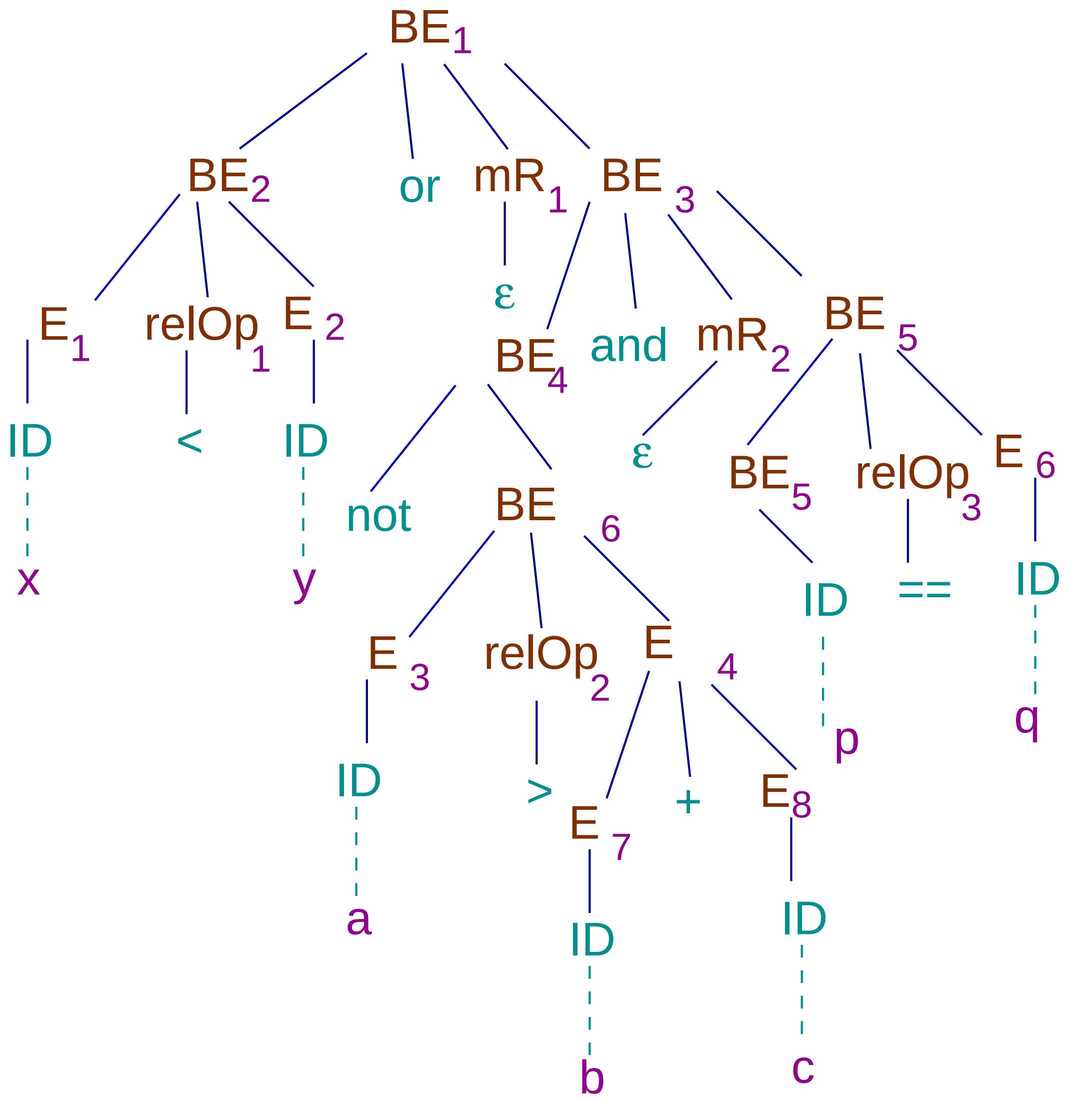
$$BE.trueLst = BE_1.trueLst \}$$

Example

Consider the Boolean expression

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

Boolean Expression: Parse Tree


BE1
BE2
or
mR1
BE 3
E1
relOp1
E 2
ε
BE4
and
mR2
BE 5
ID
<
ID
not
BE 6
ε
BE5
relOp3
E 6
x
y
E 3
relOp2
E 4
ID
==
ID
ID
>
E 7
+
E8
p
q
a
ID
ID
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 $mR_1 \rightarrow \varepsilon$. 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 ···`

  104 `goto ···`

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 ···
```

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 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

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

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

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

$$\{fill(S_1.nextLst,\ mR.nextInd)$$

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

$$SL \rightarrow S$$

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

Backpatching: Assignment, `nop` Statement and Marker

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

Backpatching: `if`-Statement

$S \rightarrow$ if BE mR then SL_1 kR else SL_2 ;

{fill(BE.trueLst, mR.nextInd)

fill(BE.falseLst, kR.nextInd+1)

temp = catLst(SL_1.nextLst, mkLst(kR.nextInd))

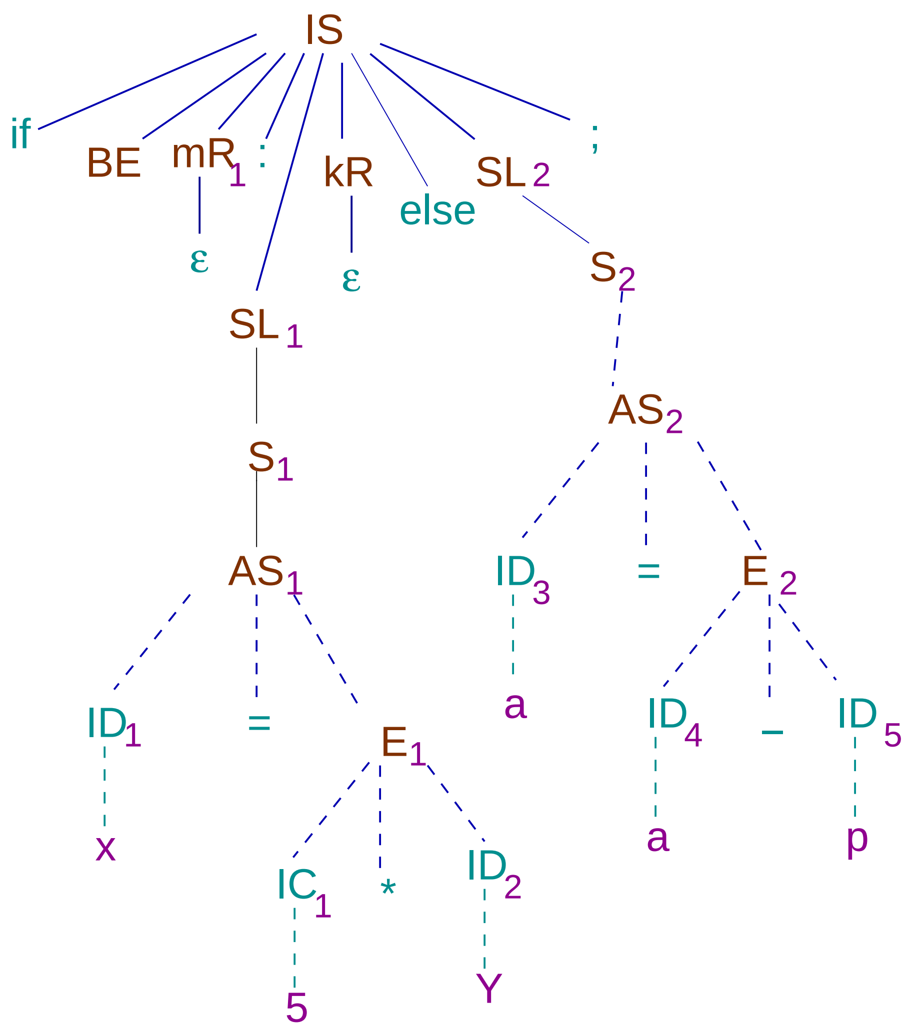
S.nextLst = catLst(temp, SL_2.nextLst) }

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

- $\text{BE.trueLst} = \{100, 105\}$ and $\text{BE.falseLst} = \{103, 106\}$.
- $\text{mR}_1\text{.nextInd} = 107$.
- $\text{kR.nextLst} = \{109\}$
- $\text{SL}_1\text{.nextLst} = \text{SL}_2\text{.nextLst} = \text{nil}$

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

Modified Grammar of `while`

Grammar After First Modification		
S	$\rightarrow$	while mR BE mR : SL end
mR	$\rightarrow$	ε

Grammar After Second Modification		
S	$\rightarrow$	while eR BE mR : SL end
mR	$\rightarrow$	ε
eR	$\rightarrow$	ε

Semantic Actions for eR

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

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

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

Semantic Actions for eR

$$S \rightarrow \text{EXITLOOP}$$

{ extLst = catLst(extLst, mkLst(nextInd))
codeGen('goto', -)
nextInd = nextInd + 1}

Backpatching Modified: `while` Statement

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

{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.

Grammar of Array Declaration

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

Grammar of Array Declaration

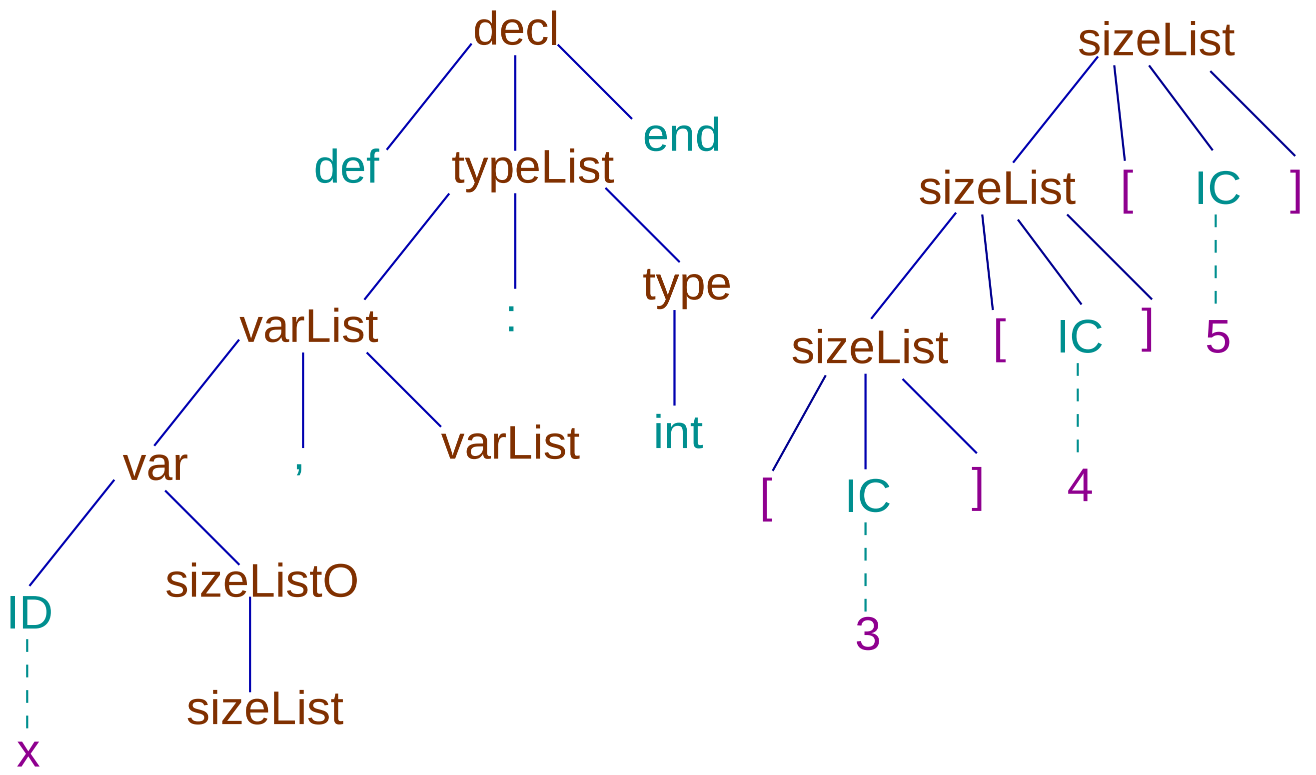
$$
\begin{aligned}
\text{var} &\rightarrow \text{ID sizeListO} \\
\text{sizeListO} &\rightarrow \text{sizeList} \\
\text{sizeList} &\rightarrow \text{sizeList [INT\_CONST]} \\
&\rightarrow \text{[INT\_CONST]}
\end{aligned}
$$

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.

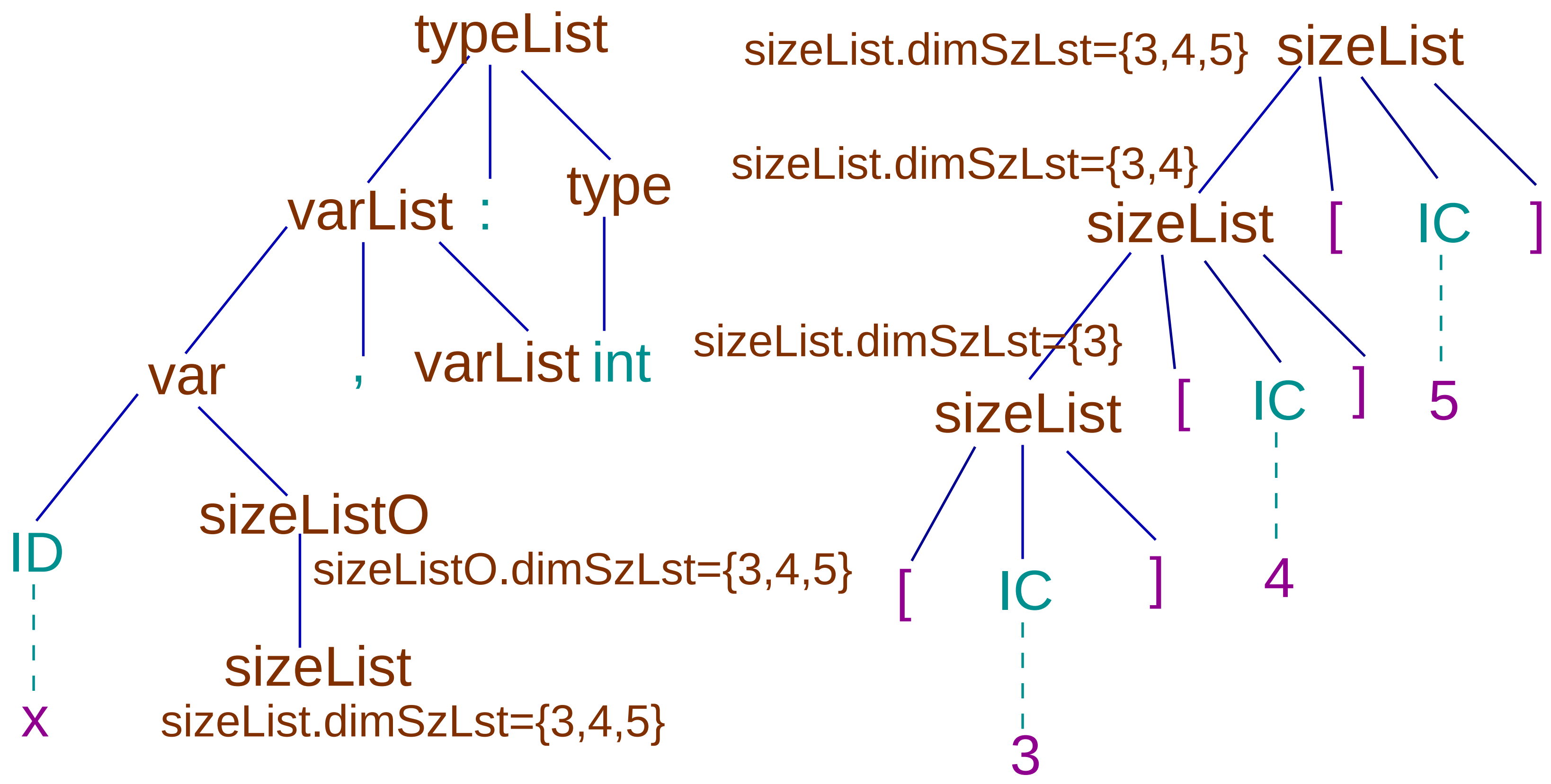
Note

- 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 := ··· x[`e_1`][`e_2`] ···`

- 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]`.

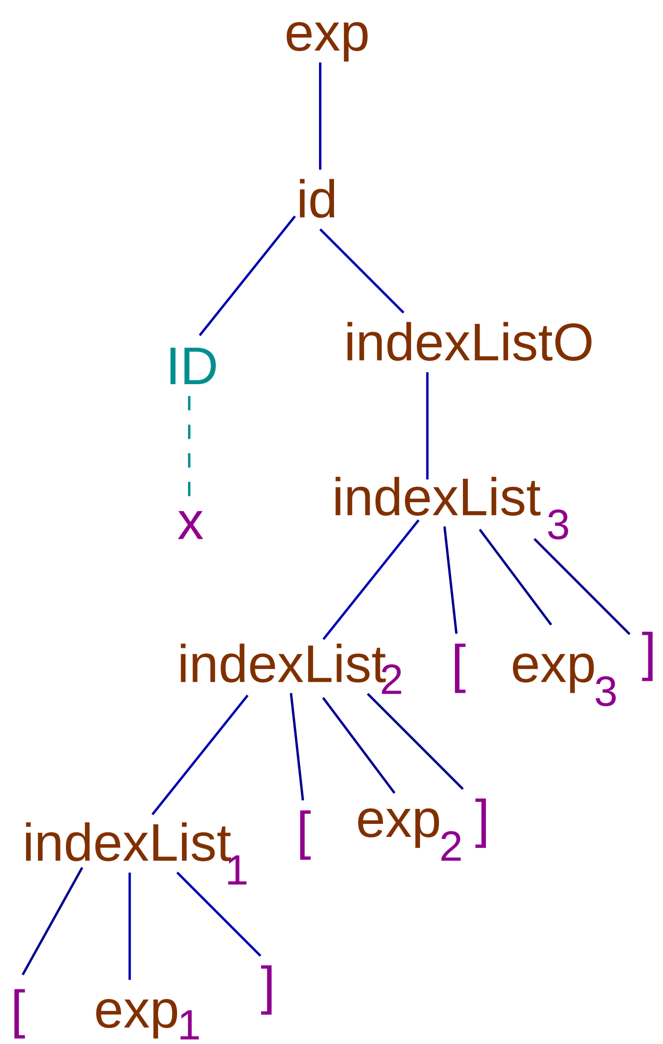
Offset Computation: an Example

- 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

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

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 = $\{\text{exp}_1.\text{loc}, \cdots, \text{exp}_k.\text{loc}\}$.
The address computation of the array element and the semantic actions corresponding to the reduction
id → ID indexListO
is as follows:
temp1 = searchInsert(symTab, newTemp(), err)
updateType(mkLocLst(temp1), ADDR)
codeGen(assign, exp_1.loc, temp1)
$\$_{j+1} = \text{exp}_1.\text{loc}$

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

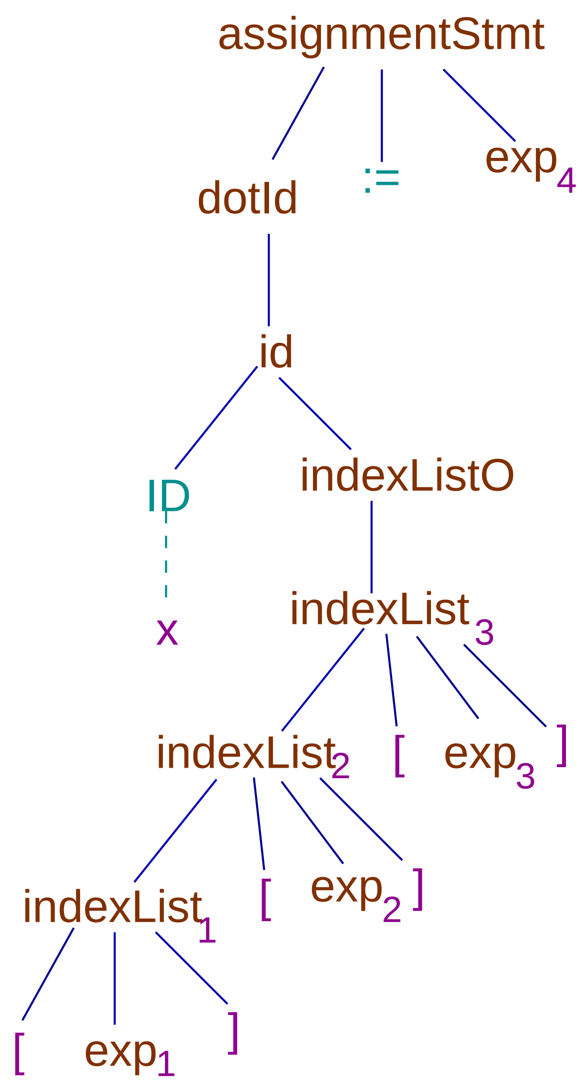
temp2 = searchInsert(symTab, newTemp(), err)
updateType(mkLocLst(temp1), ADDR)
codeGen(assAddrMultConst, temp1, w, temp2)
$\$_{j+2k} = \$_{j+2k-1} \times w$
id.base = searchInsert(symTab,ID.lexme,err).sTab.offset
id.offset = temp2

Code Generation

The 3-address code corresponding to exp → 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

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

Grammar of Function Declaration

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

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

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

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?