CS335 Assignment - 3

Aditya Tanwar
200057

April 2023

Question 1 **[50 marks]**

Consider a computation with three operators: α, β and γ. The inputs can be of two types: A and B.

Operator	**# Inputs**	**Input Types**	**# Outputs**	Output Types
α	1	A	1	A
β	2	B, B	1	B
γ	3	A, A, A or B, B, B	1	B

Neither x nor y by itself is a valid expression. An expression must have an operator.

(a) Propose a context-free grammar (CFG) to generate expressions of the desired form. Given $type(x) = A$ and $type(y) = B$, $\beta(\gamma(\alpha(x), x, x), y)$ is an example of a type-correct expression. The CFG should allow generating incorrect expressions with wrong types.

Solution: We construct the following CFG to generate the expressions:

$$S \to \alpha(G) \mid \beta(G, G) \mid \gamma(G, G, G)$$
$$G \to \alpha(G) \mid \beta(G, G) \mid \gamma(G, G, G)$$
$$G \to x \mid y$$

(b) Define an SDT based on your grammar from part (a) for type checking expressions. Include the wrong type information in the error message.

Solution: We provide the following SDT for type checking expressions constructed from the grammar of part (a):

- $S \to \alpha(G)$ **if** (G.`type` = A) **then** S.`type` $\leftarrow A$ **else** `ERROR`$(\alpha, \langle G$.`type`$\rangle)$
- $S \to \beta(G_1, G_2)$ **if** (G_1.`type` $= B$) **and** (G_2.`type` $= B$) **then** S.`type` $\leftarrow B$ **else** `ERROR`$(\beta, \langle G_1$.`type`, G_2.`type`$\rangle)$
- $S \to \gamma(G_1, G_2, G_3)$ **if** (G_1.`type` $= G_2$.`type`) **and** (G_1.`type` $= G_3$.`type`) **then** S.`type` $\leftarrow B$ **else** `ERROR`$(\gamma, \langle G_1$.`type`, G_2.`type`, G_3.`type`$\rangle)$

- $G_0 \to \alpha(G_1)$ **if** (G_1.type = A) **then** G_0.type $\leftarrow A$ **else** ERROR($\alpha, \langle G$.type$\rangle$)
- $G_0 \to \beta(G_1, G_2)$ **if** (G_1.type $= B$) **and** (G_2.type $= B$) **then** G_0.type $\leftarrow B$ **else** ERROR($\beta, \langle G_1$.type, G_2.type$\rangle$)
- $G_0 \to \gamma(G_1, G_2, G_3)$ **if** (G_1.type $= G_2$.type) **and** (G_1.type $= G_3$.type) **then** G_0.type $\leftarrow B$ **else** ERROR($\gamma, \langle G_1$.type, G_2.type, G_3.type$\rangle$)
- $G \to x$ G.type$\leftarrow A$
- $G \to y$ G.type$\leftarrow B$

ERROR() is just an auxiliary function to help print error statements. It takes input two parameters, first the operator, and second the list of arguments passed to the operator. Its definition is:

```
ERROR(op, args) {
  print($op, "expected arguments of type (")
  if ($op = α) : print("A")
  if ($op = β) : print("B, B")
  if ($op = γ) : print("A, A, A) or (B, B, B")
  print(", but received arguments", $args)
  exit()
}
```

(c) Given $type(x_i) = A$, $type(y_i) = B$, draw the annotated parse tree for the expression: $\gamma(\gamma(\alpha(x_1), x_2, \alpha(x_2)), \beta(y_1, y_2), \beta(y_2, y_3))$

Solution: The annotated parse tree (Figure 1) has been provided towards the end of the document. The tree has non-terminals/terminals in circular nodes and only the type in rectangular nodes to avoid cluttering.

Question 2 **[50 marks]**

A program P is a sequence of two or more statements separated by semicolons. A semicolon is not required for the last statement in P. Each statement assigns the value of an expression E to the variable x. An expression is either the sum of two expressions, a multiplication of two expressions, the constant 1, or the current value of x.

Statements are evaluated in left-to-right order.

Your solution should not use any global state. You can also assume that P cannot be empty.

- For the i^{th} statement $x = E_i$, the value of references to x inside E_i is the value assigned to. x in the previous statement $x = E_{i-1}$.
- For the first statement $x = E_1$, the value of references to x in E_1 is 0.

- The value of a program is the value assigned to x by the last statement.

Answer the following:

(i) Propose a CFG to represent programs generated by the above specification.

Solution: We construct the following CFG to generate the programs by the above specification:

$$P \to T; T$$
$$T \to x = E \mid T; T;$$
$$E \to 1 \mid x \mid E + E \mid E * E$$

where P is a non-terminal corresponding to the whole program, T corresponds to a statement, and E corresponds to any expression (or sub-expression).

(ii) Propose an SDT to compute the value of the program generated by P. Your solution should assign attribute $P.val$ the value of the program generated by P.
To avoid verbosity, we only include numbers in the middle of production rules instead of the complete actions. The corresponding actions are written immediately after.

Solution: We provide the following SDT to compute the value of the program generated by P, assuming there to be a **single** variable:

- $P \to (1)\ T_1;\ (2)\ T_2\ (3)$
  - (1) $T_1.x \leftarrow 0$
  - (2) $T_2.x \leftarrow T_1.val$
  - (3) $P.val \leftarrow T_2.val$
- $T \to x = (1)\ E\ (2)$
  - (1) $E.x \leftarrow T.x$
  - (2) $T.val \leftarrow E.val$
- $T_0 \to (1)\ T_1; (2)\ T_2;\ (3)$
  - (1) $T_1.x \leftarrow T_0.x$
  - (2) $T_2.x \leftarrow T_1.val$
  - (3) $T_0.val \leftarrow T_2.val$
- $E \to 1\ (1)$
  - (1) $E.val \leftarrow 1$
- $E \to x\ (1)$
  - (1) $E.val \leftarrow E.x$
- $E_0 \to (1)\ E_1 + (2)\ E_2\ (3)$
  - (1) $E_1.x \leftarrow E_0.x$
  - (2) $E_2.x \leftarrow E_0.x$
  - (3) $E_0.val \leftarrow E_1.val + E_2.val$

- $E_0 \rightarrow (1)\ E_1 * (2)\ E_2\ (3)$

  (1) $E_1.x \leftarrow E_0.x$

  (2) $E_2.x \leftarrow E_0.x$

  (3) $E_0.val \leftarrow E_1.val * E_2.val$

  where the attribute x is used to store the last assigned value of x in the statement preceding the non-terminal, and *val* stores the the value computed in an expression for the non-terminal E, or the final value assigned to x for non-terminals like P and T.

  A solution using linked lists is provided for the case of **multiple variables** at the end of the document, if required.

(iii) Indicate for each attribute whether it is inherited or synthesized.

Solution: The attribute x is inherited for all non-terminals T, E, while the attribute *val* is synthesized for all non-terminals P, T, E.

Question 3 **[50 marks]**

Consider a programming language where reading from a variable before it has been assigned is an error. You are required to design an "undefined variable" checker for the language. Do not modify the grammar.

Your SDT should support the following requirements:

- If a statement S contains an expression E, and E references a variable that maybe undefined before reaching S, print the error message "A variable may be undefined. You need not print which variable (or variables) is undefined". It is okay to exit the program after encountering an error.
- If v is defined before a statement S, then v is also defined after S.
- Variable v is defined after the statement $v = E$.
- A variable defined inside an if is defined after the if when it is defined in both branches.
- In a statement sequence $S_1; S_2$, variables defined after S_1 are defined before S_2.

$$
\begin{aligned}
&\textit{stmt} \rightarrow \textit{var} = \textit{expr}\\
&\textit{stmt} \rightarrow \textit{stmt}\ ;\ \textit{stmt}\\
&\textit{stmt} \rightarrow \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt}\ \textbf{fi}\\
&\textit{expr} \rightarrow \textit{expr}\ +\ \textit{expr}\\
&\textit{expr} \rightarrow \textit{expr}\ <\ \textit{expr}\\
&\textit{expr} \rightarrow \textit{var}\\
&\textit{expr} \rightarrow \textbf{int\_const}
\end{aligned}
$$

Your solution should include the following attributes. You can assume the sets start empty.

`var.name` is a string containing the variable's name. This is defined by the lexer, so you do not need to compute it, you can just use it.

`expr.refd` is the set of variables referenced inside the expression.

`stmt.indefs` is the set of variables defined at the beginning of the statement.

`stmt.outdefs` is the set of variables defined at the end of the statement.

You can invoke common set operations like unions and intersections on the set attributes. You are also allowed to use temporaries for intermediate computations.

Assumption(s)

Two variables having the same name implies that the two variables are identical. Also, the "set of variables" corresponding to a non-terminal only contains variable names, i.e., `var.name`'s.

Solution

Again, to avoid verbosity, we only include numbers in the middle of production rules instead of the complete actions. The corresponding actions are written immediately after. We provide the following SDT:

- *stmt* → *var* = *expr* (1)

  (1) **foreach** *name* ∈ `expr.ref:`
  if *name* ∉ `stmt.indefs:`
  `print("A variable may be undefined.")`
  `exit()`
  `stmt.outdefs` ← `stmt.indefs` ∪ `{var.name}`

- $stmt_0$ → (1) $stmt_1$; (2) $stmt_2$ (3)

  (1) $\texttt{stmt}_1\texttt{.indefs} \leftarrow \texttt{stmt}_0\texttt{.indefs}$

  (2) $\texttt{stmt}_2\texttt{.indefs} \leftarrow \texttt{stmt}_1\texttt{.outdefs}$

  (3) $\texttt{stmt}_0\texttt{.outdefs} \leftarrow \texttt{stmt}_2\texttt{.outdefs}$

- $stmt_0$ → **if** *expr* (1) **then** (2) $stmt_1$ **else** (3) $stmt_2$ **fi** (4)

  (1) **foreach** *name* ∈ `expr.ref:`
  if *name* ∉ $\texttt{stmt}_0\texttt{.indefs:}$
  `print("A variable may be undefined.")`
  `exit()`

(2) $\texttt{stmt}_1\texttt{.indefs} \leftarrow \texttt{stmt}_0\texttt{.indefs}$

(3) $\texttt{stmt}_2\texttt{.indefs} \leftarrow \texttt{stmt}_0\texttt{.indefs}$

(4) $\texttt{stmt}_0\texttt{.outdefs} \leftarrow \texttt{stmt}_1\texttt{.outdefs} \cap \texttt{stmt}_2\texttt{.outdefs}$

- $expr_0 \rightarrow expr_1 \ + \ expr_2$ (1)

  (1) $\texttt{expr}_0\texttt{.refd} \leftarrow \texttt{expr}_1\texttt{.refd} \cup \texttt{expr}_2\texttt{.refd}$

- $expr_0 \rightarrow expr_1 \ < \ expr_2$ (1)

  (1) $\texttt{expr}_0\texttt{.refd} \leftarrow \texttt{expr}_1\texttt{.refd} \cup \texttt{expr}_2\texttt{.refd}$

- $expr \rightarrow var$ (1)

  (1) `expr.refd` $\leftarrow$ `{var.name}`

- $expr \rightarrow$ **int_const**

Question 4 **[50 marks]**

We have discussed generating 3AC for array accesses using semantic translations. Consider the following extended grammar with semantic translation.

$S \rightarrow \mathbf{id} = E$	$\{gen(symtop.get(\mathbf{id}.lexeme)\text{“ = ”}E.addr)\}$
$S \rightarrow L = E$	$\{gen(L.array.base\text{“[”}L.addr\text{“]”“ = ”}E.addr)\}$
$E \rightarrow E_1 + E_2$	$\{E.addr = newTemp(); gen(E.addr\text{“ = ”}E_1.addr\text{“ + ”}E_2.addr)\}$
$E \rightarrow E_1 * E_2$	$\{E.addr = newTemp(); gen(E.addr\text{“ = ”}E_1.addr\text{“ * ”}E_2.addr)\}$
$E \rightarrow \mathbf{id}$	$\{E.addr = symtop.get(\mathbf{id}.lexeme)\}$
$E \rightarrow L$	$\{E.addr = newTemp(); gen(E.addr\text{“ = ”}L.array.base\text{“[”}L.addr\text{“]”})\}$
$L \rightarrow \mathbf{id}[E]$	$\{L.array = symtop.get(\mathbf{id}.lexeme); L.type = L.array.type.elem;$ $L.addr = newTemp(); gen(L.addr\text{“ = ”}E.addr\text{“ * ”}L.type.width)\}$
$L \rightarrow L_1[E]$	$\{L.array = L_1.array; L.type = L_1.type.elem; t = newTemp();$ $gen(t\text{“ = ”}E.addr\text{“ * ”}L.type.width); gen(L.addr\text{“ = ”}L_1.addr\text{“ + ”}t); \}$

Assume the size of integers to be four bytes, and that the arrays are zero-indexed. Let A, B, and C be integer arrays of dimensions 10×5, 5×7, and 10×7 respectively. Construct an annotated parse tree for the expression $C[i][j] + A[i][k] \times B[k][j]$ and show the 3AC code sequence generated for the expression.

Assumptions/Conventions

- All the nodes are in a circular node. Rectangular boxes represent code to be generated.

- Boxes have a label of the form (i) and an optional allocation using $new()$. This has been done in favour of visuals, since the tree could not accommodate the code being expanded in the middle of it. The complete code that replaces the label has been provided in a separate place. Also, to save space, $newTemp()$ is replaced by $new()$ in the annotations.

- $E_i.addr$ is denoted by e_i and similarly, $L_i.addr$ is denoted by l_i in the 3AC. The only two exceptions are when $E.addr$ is assigned a symbol table entry using *sym-top.get(***id***.lexeme)*, in which case it is replaced by the lexeme instead in 3AC and similarly for *L.array.base*.

- When a non-terminal goes to **id**, the "id" has been replaced by its lexeme.

- To save space, square brackets have been clubbed with E, instead of providing them with a separate node in productions like $L \rightarrow L[E] \mid \mathbf{id}[E]$.

- It is assumed that the symbol table entries corresponding to the arrays A, B, C contain the sizes (i.e., the width of the types), and this is copied when the type is assigned in a semantic action. Specifically, the width of an element of A is assumed to be $5 \times 4 = 20$, and that of $A[\cdot]$ is assumed to be 4. Similarly, for B (28, 4) and C (28, 4).

Parse Tree & 3AC

The annotated parse tree can be found here (Figure 2), and the 3AC corresponding to $C[i][j] + A[i][k] \times B[k][j]$ is as follows:

(1)	$l_2 = i * 28;$	/* $L_2 \rightarrow \mathbf{id}[E_6] = C[i]$ */
(2)	$t_1 = j * 4;$	/* $L_1 \rightarrow L_2[E_5] = L_2[j]$ */
	$l_1 = l_2 + t_1;$	
(3)	$e_1 = C[l_1];$	/* $E \rightarrow L = C[i][j]$ */
(4)	$l_5 = i * 20;$	/* $L_5 \rightarrow \mathbf{id}[E_9] = A[i]$ */
(5)	$t_2 = k * 4;$	/* $L_3 \rightarrow L_5[E_7] = L_5[k]$ */
	$l_3 = l_5 + t_2;$	
(6)	$e_3 = A[l_3];$	/* $E_3 \rightarrow L_3 = A[i][k]$ */
(7)	$l_6 = k * 28$	/* $L_6 \rightarrow \mathbf{id}[E_{10}] = B[k]$ */
(8)	$t_3 = j * 4$	/* $L_4 \rightarrow L_6[E_8] = L_6[k]$ */
	$l_4 = l_6 + t_3$	
(9)	$e_4 = B[l_4]$	/* $E_4 \rightarrow L_4 = B[k][j]$ */
(10)	$e_2 = e_3 * e_4$	/* $E_2 \rightarrow E_3 * E_4$ */
(11)	$e_0 = e_1 + e_2$	/* $E_0 \rightarrow E_1 + E_2$ */

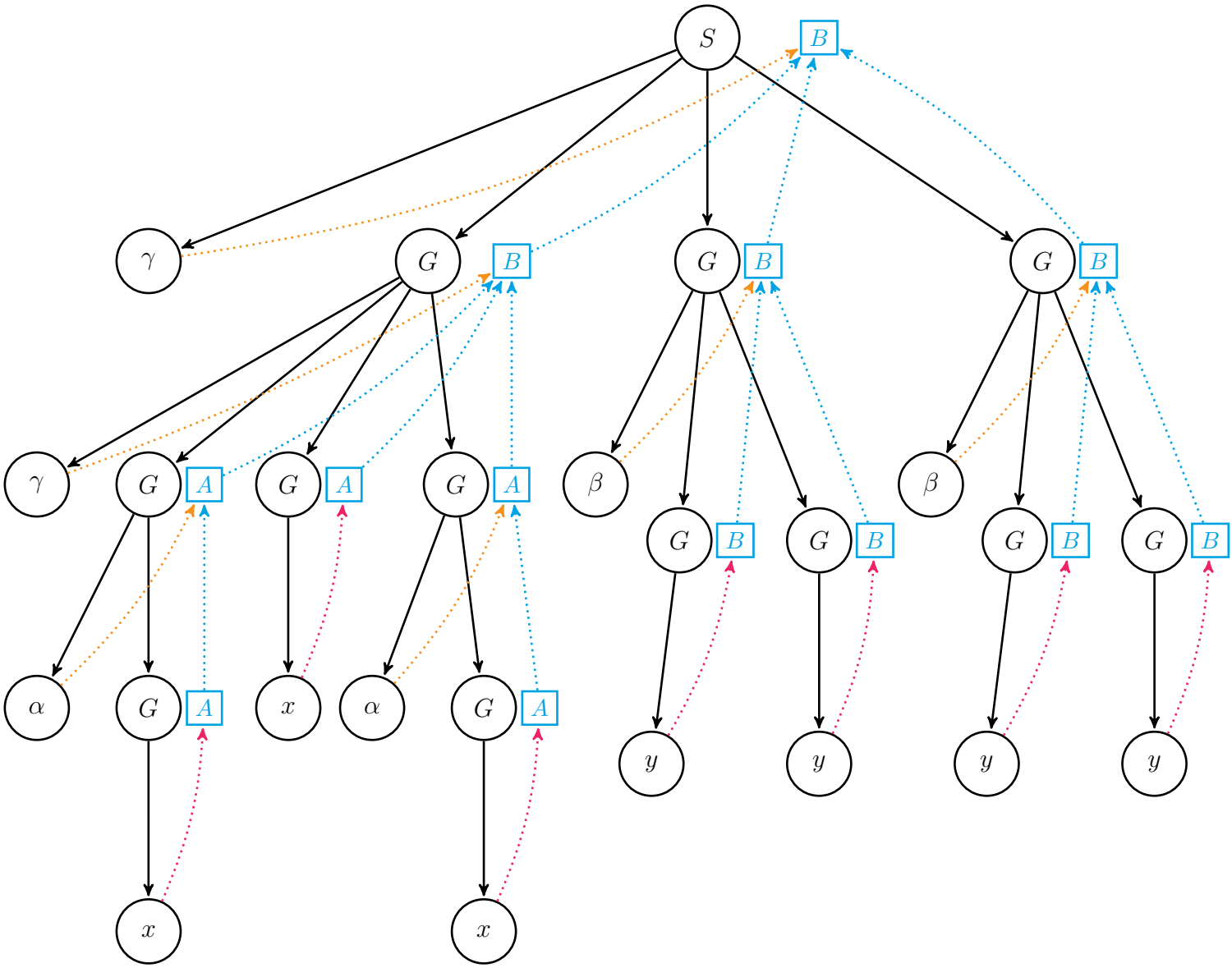

Figure 1: Annotated Parse Tree for Question 1

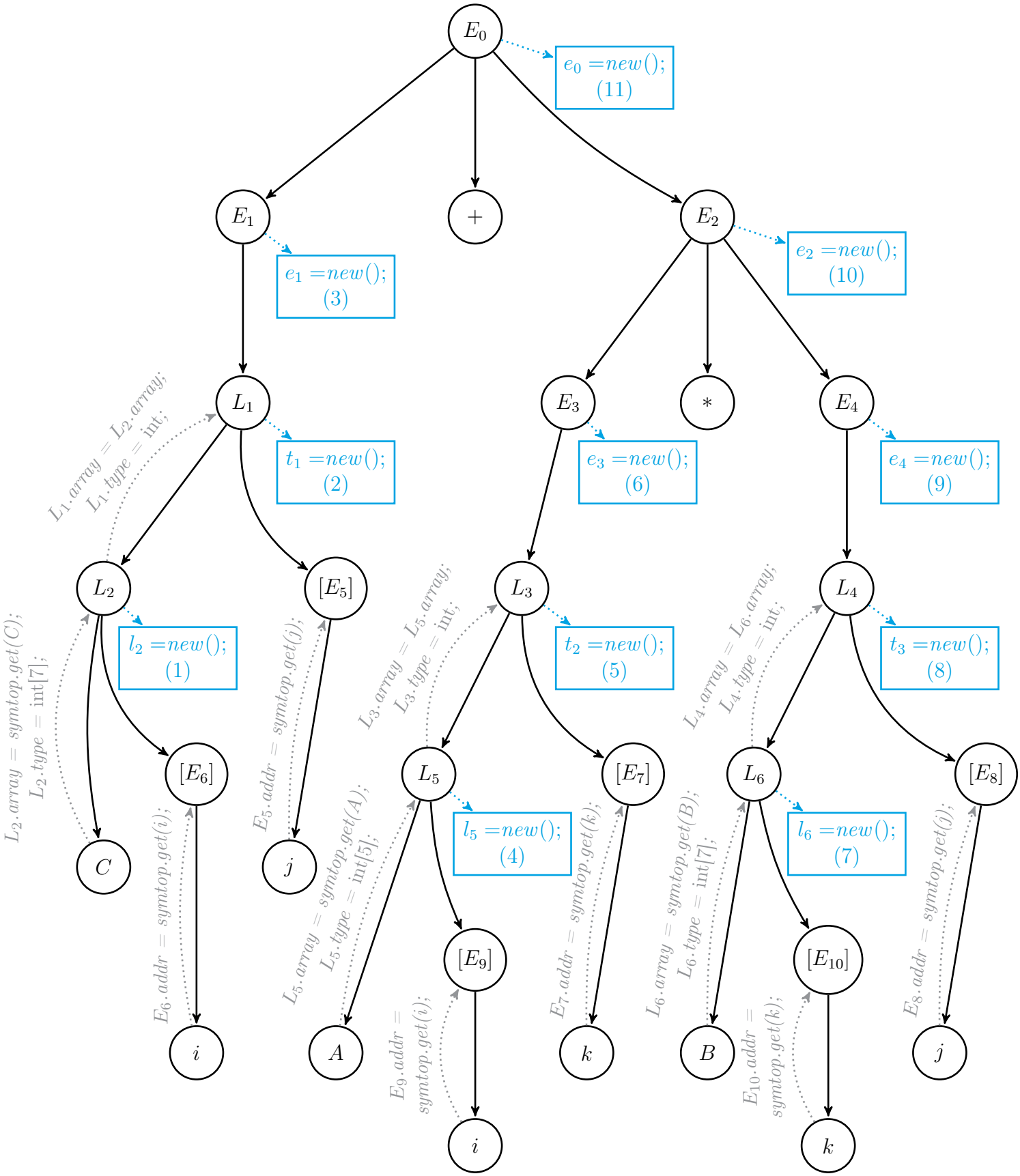

Figure 2: Annotated Parse Tree for Question 4

Alternate Solution for Q2

We provide the following SDT to compute the value of the program generated by P, assuming there to be **multiple** variables:

- $P \to (1)\ T_1;\ (2)\ T_2\ (3)$
  - (1) $T_1.defs\_in \leftarrow NULL$
  - (2) $T_2.defs\_in \leftarrow T_1.defs\_out$
  - (3) $P.val \leftarrow T_2.val$
- $T \to x = (1)\ E\ (2)$
  - (1) $E.defs \leftarrow T.defs\_in$
  - (2) $T.val \leftarrow E.val$
    $T.defs\_out \leftarrow upd(T.defs\_in, x, E.val)$
- $T_0 \to (1)\ T_1; (2)\ T_2;\ (3)$
  - (1) $T_1.defs\_in \leftarrow T_0.defs\_out$
  - (2) $T_2.defs\_in \leftarrow T_1.defs\_out$
  - (3) $T_0.val \leftarrow T_2.val$
    $T_0.defs\_out \leftarrow T_2.defs\_out$
- $E \to 1\ (1)$
  - (1) $E.val \leftarrow 1$
- $E \to x\ (1)$
  - (1) $E.val \leftarrow get(E.defs, x)$
- $E_0 \to (1)\ E_1 + (2)\ E_2\ (3)$
  - (1) $E_1.defs \leftarrow E_0.defs$
  - (2) $E_2.defs \leftarrow E_0.defs$
  - (3) $E_0.val \leftarrow E_1.val + E_2.val$
- $E_0 \to (1)\ E_1 * (2)\ E_2\ (3)$
  - (1) $E_1.defs \leftarrow E_0.defs$
  - (2) $E_2.defs \leftarrow E_0.defs$
  - (3) $E_0.val \leftarrow E_1.val * E_2.val$

where *defs/defs_in/defs_out* is a list used to store the values of variables in the previous statement, and the attribute *val* is used to store the value of the expression (for expressions)/ value assigned to x (for statements). *get()* and *upd()* are helper functions, *get* returns the last assigned value of a variable x in a list *defs*, and *upd* updates the value of a variable x in a list *defs*.

We now provide some details about the data structure of *defs* and the helper functions in the following paragraphs. Although *defs* does the function of a map, it can be implemented using a simple linked list (if time complexity is not a concern; if it is a concern, a hash-map or a binary search tree can be used, like in the STL library of C++). This linked list's nodes shall have two entries, the first one being a string representing the lexeme of the variable, and the second being an integer storing the last value assigned to the corresponding variable. The *get()* function takes two parameters, a linked list of the above-described type, and a string representing the lexeme of the variable whose value we need to get. The function simply traverses the linked list, and checks if the string is present in the list, if it is present, it returns the value stored in the same node, otherwise it returns 0.
The *upd()* function takes three parameters, a linked list of the above-described type, a string representing the lexeme of the variable we need to update, and the value which we need to update the variable to.
Pseudo-codes of the two functions follow:

```
get(list, x){
  for node ∈ list:
    if node.lexeme = x:
      return node.value
  return 0
}
```

```
upd(list, x, val){
  for node ∈ list:
    if node.lexeme = x:
      node.value ← val
      return list
  node ← new_node(x, val)
  list.insert(node)
  return list
}
```

- *defs_in*: Inherited attribute for all non-terminals T.
- *defs_out*: Synthesized attribute for all non-terminals T.
- *defs*: Inherited attribute for all expressions E.
- *val*: Synthesized attribute for all non-terminals P, T, E.

This is also apparent from the place in the production rules where these attributes are set (for example, *defs_out* and *val* are only ever set at the end of the production rules, while the other two are set before).