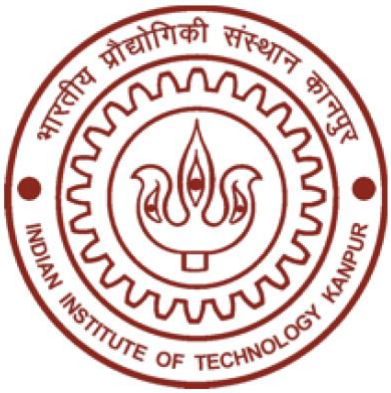


CS335

Assignment III

Course Instructor :

Dr. Swarnendu Biswas

Submitted By:

Nishant Roshan
(200643)

Department of Computer science and Engineering
Indian Institute of Technology, Kanpur

April 2, 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

(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.

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

(c) Given *type*(x_i) = A *type*(y_i) = B, draw the annotated parse tree for the expression:

$$\gamma(\gamma(\alpha(x_1), x_2, \alpha(x_2)), \beta(y_1, y_2), \beta(y_2, y_3))$$

Neither x_1 or y_1 by themselves are valid expressions. An expression must have an operator.

Solution:

a) In the proposed CFG, let S be the start non-terminal. Let E denote the non-terminal representing expressions, T denotes the non-terminal representing a term in the expression, Id denotes the non-terminal representing Identifiers, while '(' ')' ',' x and y are the terminals in the grammar. The proposed grammar:

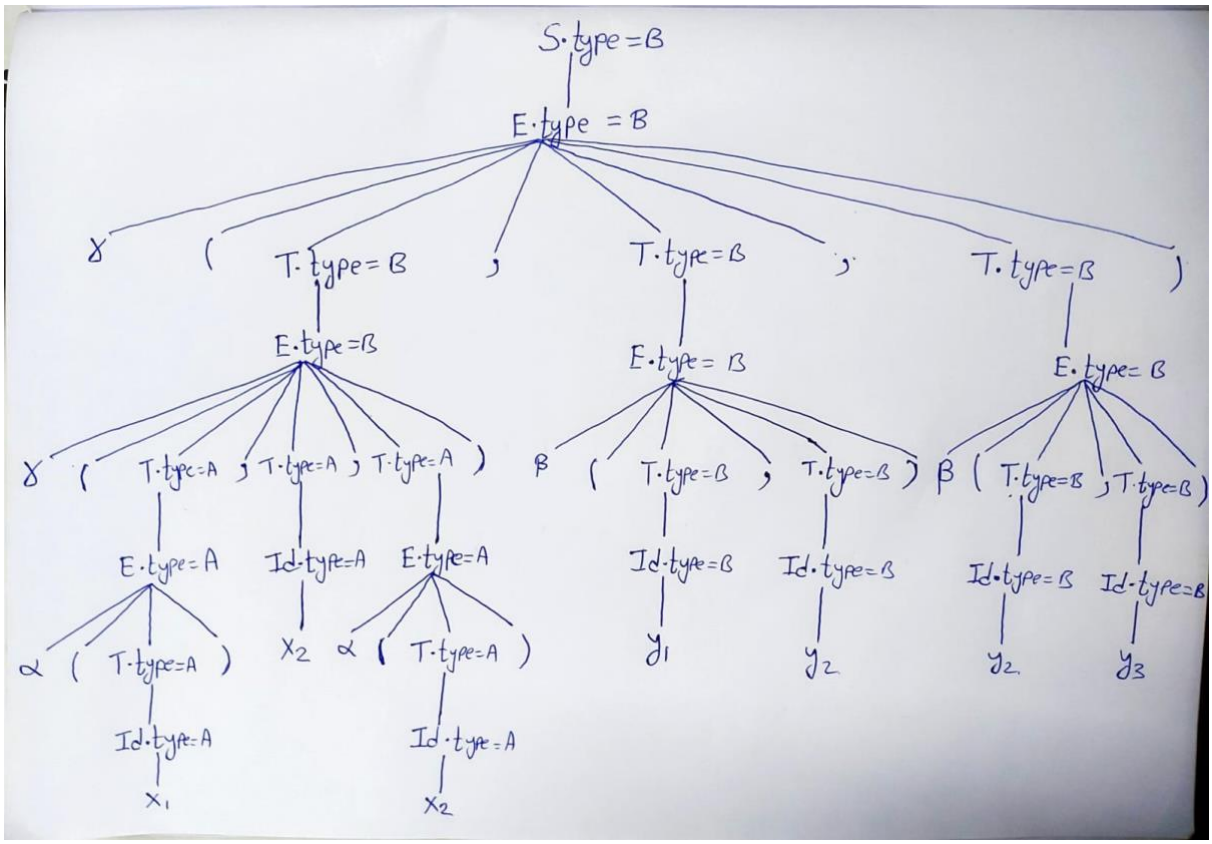
$$S \rightarrow E$$
$$E \rightarrow \alpha(T) \mid \beta(T_1, T_2) \mid \gamma(T_1, T_2, T_3)$$
$$T \rightarrow E \mid Id$$
$$Id \rightarrow x \mid y$$

NOTE: T_1 , T_2, T_3 denote the same non-terminal T. The subscripts are given only for understanding purpose.

b) The corresponding SDT is:

Productions:	Semantic actions:
S —> E	S.type = E.type
E —> α(T)	if(T.type!=A)print{error,expected argument of type A} else E.type = A
E —> β(T1 ,T2)	if(T1.type != B || T2.type != B) print{error, both arguments must be of type B} else E.type = B
E —> γ(T1 ,T2 ,T3)	if(T1.type != T2.type || T2.type != T3.type) print{error, all the three arguments must be of same type} else E.type = B
T —> E	T.type = E.type
T —> Id	T.type = Id.type
Id —> x	Id.type = A
Id —> y	Id.type = B

c) Annotated parse tree for the given expression:



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.

- 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,
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.
iii. Indicate for each attribute whether it is inherited or synthesized.

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

Solution:

a) Let P denote the starting non-terminal which representing the beginning of the program. Let stmts denote the non-terminal representing two or more statements. Let stmt denote the non-terminal representing a statement. A and B are two non-terminals introduced to take into account precedence of multiplication over addition. x and 1 are terminals denoting the variable x and the constant 1.

Assumption: The grammar I have built is left associative and multiplication has higher precedence than addition.

Grammar:

$$P \rightarrow stmts$$
$$stmts_1 \rightarrow stmts_2; stmt$$
$$stmts \rightarrow stmt_1; stmt_2$$
$$stmt \rightarrow x = E$$
$$E_1 \rightarrow E_2 + A$$
$$E \rightarrow A$$

$A_1 \rightarrow A_2 * B$

$A \rightarrow B$

$B \rightarrow x$

$B \rightarrow 1$

NOTE: A_1, A_2 correspond to the same non-terminal A. Similar is the case for E_1 and E_2 & $stmt_1$ and $stmt_2$. The subscripts are given only for understanding the relation between attributes of the same non-terminal defined on two different sides of the production.

b) The corresponding SDT would be:

Productions:	**Semantic action:**
$P \rightarrow stmts$	$P.val = stmts.val$ $stmts.inh = 0$
$stmts_1 \rightarrow stmts_2; stmt$	$stmts_1.val = stmt.val$ $stmts_2.inh = stmts_1.inh$ $stmt.inh = stmts_2.val$
$stmts \rightarrow stmt_1; stmt_2$	$stmts.val = stmt_2.val$ $stmt_1.inh = stmts.inh$ $stmt_2.inh = stmt_1.val$
$stmt \rightarrow x = E$	$stmt.val = E.val$ $E.inh = stmt.inh$
$E_1 \rightarrow E_2 + A$	$E_1.val = E_2.val + A.val$ $E_2.inh = E_1.inh$ $A.inh = E_1.inh$
$E \rightarrow A$	$E.val = A.val$ $A.inh = E.inh$
$A_1 \rightarrow A_2 * B$	$A_1.val = A_2.val * B.val$ $A_2.inh = A_1.inh$ $B.inh = A_1.inh$
$A \rightarrow B$	$A.val = B.val$ $B.inh = A.inh$
$B \rightarrow x$	$B.val = B.inh$
$B \rightarrow 1$	$B.val = 1$

d) The type of attributes are:

Inherited attributes:	**Synthesized attributes:**
stmts.inh, stmt.inh, E.inh, B.inh	stmts.val, stmt.val, E.val, A.val, B.val

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 S1;S2, variables defined after S1 are defined before S2.

stmt → var = expr
stmt → stmt ; stmt
stmt → **if** expr **then** stmt **else** stmt **fi**
expr → expr + expr
expr → expr < expr
expr → var
expr → **int_const**

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.

Solution:

For the grammar given in the question, consider the following SDT:

Productions:	Semantic actions:
stmt —> var = expr	if(expr.refd U stmt.indef != stmt.indefs) {print{"A variable may be undefined"}; return 1;} else stmt.outdefs = stmt.indefs U {var.name}
$stmt_1$ —> $stmt_2$; $stmt_3$	$stmt_3$.indefs = $stmt_2$.outdefs; $stmt_2$.indefs = $stmt_1$.indefs; $stmt_1$.outdefs = $stmt_3$.outdefs

$stmt_1$ —> if expr then $stmt_2$ else $stmt_3$ fi	if(expr.refd U $stmt_1$.indef != $stmt_1$.indefs)
	{ print{"A variable may be undefined"}; return 1; }
	
	else{ $stmt_2$.indefs = $stmt_1$.indefs;
	$stmt_3$.indefs = $stmt_1$.indefs;
	$stmt_1$.outdefs = $stmt_2$.outdefs U $stmt_3$.outdefs;}

$expr_1$ —> $expr_2$ + $expr_3$	$expr_1$.refd = $expr_2$.refd U $expr_3$.refd
$expr_1$ —> $expr_2$ < $expr_3$	$expr_1$.refd = $expr_2$.refd U $expr_3$.refd
expr —> var	expr.refd = {var.name}
expr —> int_const	expr.refd = { }

NOTE:

1) $stmt_1$, $stmt_2$, $stmt_3$ refer to the same non-terminal stmt. Similarly, $expr_1$, $expr_2$, $expr_3$ refer to the same non-terminal expr. The subscripts are given just to differentiate their attributes for easy understanding.

2) The attributes are sets which are initially all empty.

Question 4) **[50 marks]**

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

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

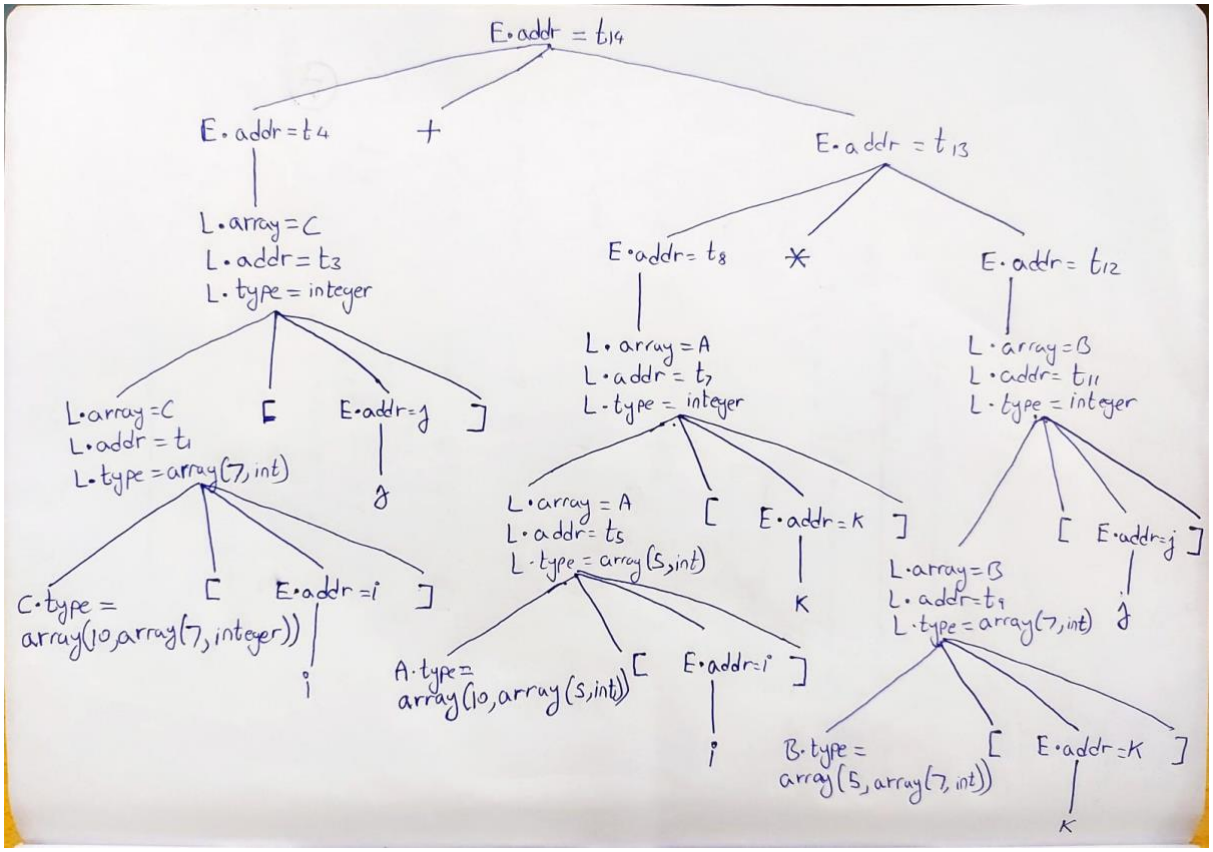
Assume the size of integers to be four bytes, and that the arrays are zero-indexed. Let A, B, and C be integer arrays of dimensions 10 × 5, 5 × 7, and 10 × 7 respectively. Construct an annotated parse tree for the expression C[i][j] + A[i][k] × B[k][j] and show the 3AC code sequence generated for the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

Solution:

Given expression: C[i][j] + A[i][k] × B[k][j]

Assumption: For the given array elements (integers), multiplication has higher precedence than addition. This means that our parse tree will evaluate C[i][j] and A[i][k] × B[k][j] separately and then add both to get the final result.

Annotated Parse Tree:



The generated 3AC code:

- $t_1 = i * 28$
- $t_2 = j * 4$
- $t_3 = t_1 + t_2$
- $t_4 = C[t_3]$
- $t_5 = i * 20$
- $t_6 = k * 4$
- $t_7 = t_5 + t_6$
- $t_8 = A[t_7]$
- $t_9 = k * 28$
- $t_{10} = j * 4$
- $t_{11} = t_9 + t_{10}$
- $t_{12} = B[t_{11}]$
- $t_{13} = t_8 * t_{12}$
- $t_{14} = t_4 + t_{13}$

Thank you!