Assignment

Program : B.Tech. (*Information Technology and Mathematical Innovations)*

Department : Cluster Innovation Centre, University of Delhi

Semester : VII

Title of Paper : Computer Language Design & Engineering

Name of Student : Ashutosh Jha

Roll No. of Student : 11811

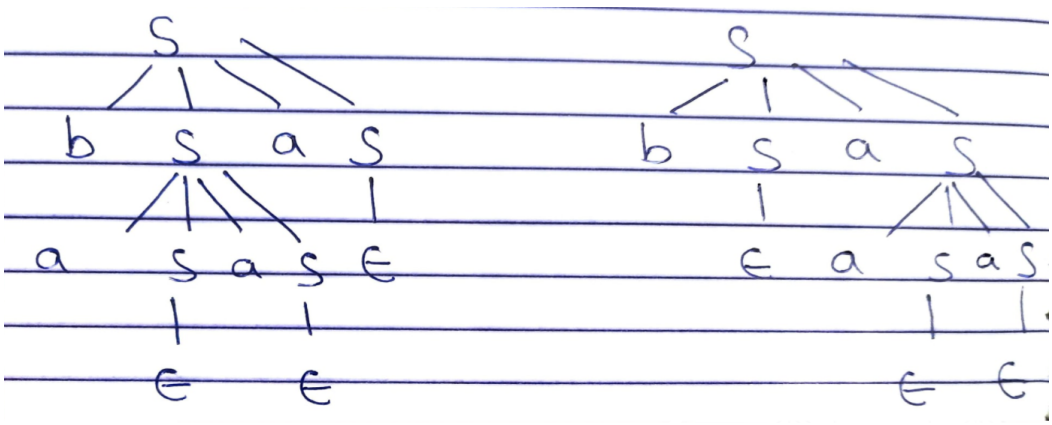
Date of Submission : Nov 19, 2021

Ques. 1: Check if Grammar is Ambiguous or not?

$S \rightarrow bSaS$

$S \rightarrow aSaS$

$S \rightarrow \in$



Ans. 1: Grammar is Ambiguous.

Ques. 2: Find First and Flow for :

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC / \in$

$D \rightarrow EF$

$E \rightarrow g / \in$

$F \rightarrow f / \in$

Ans. 2: First (S) = {a}

(B) = {c}

(C) = {b,∈}

(D) = {g,f,,∈}

(E) = {g,,∈}

(F) = {f,,∈}

Flow (S) = { $ }

(B) = {g,f,h}

(C) = {g,f,h}

(D) = {h}

(E) = {f,h}

(F) = {h}

Ques. 3: Provide a detailed explanation of each phase of Compiler with the help of the example?

Ans. 3: The compiler operates in various phases; each phase transforms the source program from one representation to another. Every phase takes inputs from its previous stage and feeds its output to the next phase of the compiler.

There are 6 phases in a compiler. Each of these phases helps in converting the high-level language to machine code. The phases of a compiler are:

1. **Lexical analysis:** Lexical Analysis is the first phase when the compiler scans the source code. This process can be left to right, character by character, and group these characters into tokens.

   Here, the character stream from the source program is grouped in meaningful sequences by identifying the tokens. It makes the entry of the corresponding tickets into the symbol table and passes that token to the next phase.

   The primary functions of this phase are:

   - Identify the lexical units in a source code.
   - Classify lexical units into classes like constants, reserved words, and enter them in different tables. It will Ignore comments in the source program.
   - Identify a token that is not a part of the language.

   Example:

   x = y + 10

   Tokens

X	identifier
=	Assignment operator
Y	identifier
+	Addition operator
10	Number

2. **Syntax analysis:** Syntax analysis is all about discovering structure in code. It determines whether or not a text follows the expected format. The main aim of this phase is to make sure that the source code written by the programmer is correct or not. Syntax analysis is based on the rules based on the specific programming language by constructing the parse tree with the help of tokens. It also determines the structure of the source language and the grammar or syntax of the language.

   Here, is a list of tasks performed in this phase:

   - Obtain tokens from the lexical analyzer
   - Checks if the expression is syntactically correct or not
   - Report all syntax errors
   - Construct a hierarchical structure which is known as a parse tree

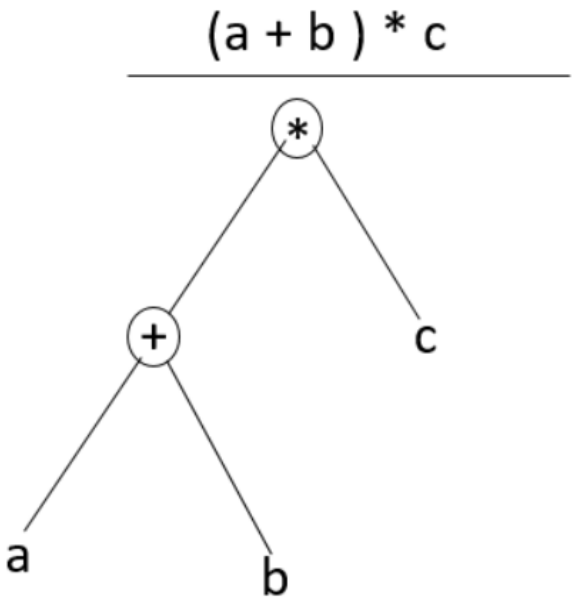
   Example

   Any identifier/number is an expression

   If x is an identifier and y+10 is an expression, then x= y+10 is a statement.

   Consider parse tree for the following example

   (a+b)*c



In Parse Tree

- Interior node: record with an operator filed and two files for children
- Leaf: records with 2/more fields; one for token and other information about the token
- Ensure that the components of the program fit together meaningfully
- Gathers type information and checks for type compatibility
- Checks operands are permitted by the source language

3. **Semantic analysis:** Semantic analysis checks the semantic consistency of the code. It uses the syntax tree of the previous phase along with the symbol table to verify that the given source code is semantically consistent. It also checks whether the code is conveying an appropriate meaning.
   Semantic Analyzer will check for Type mismatches, incompatible operands, a function called with improper arguments, an undeclared variable, etc.
   Functions of the Semantic analysis phase are:
   - Helps you to store type information gathered and save it in a symbol table or syntax tree
   - Allows you to perform type checking
   - In the case of type mismatch, where there are no exact type correction rules which satisfy the desired operation a semantic error is shown
   - Collects type information and checks for type compatibility
   - Checks if the source language permits the operands or not

   Example

   ```
   float x = 20.2;
   float y = x*30;
   ```

   In the above code, the semantic analyzer will typecast the integer 30 to float 30.0 before multiplication
4. **Intermediate code generator:** Once the semantic analysis phase is over the compiler generates intermediate code for the target machine. It represents a program for some abstract machine.
   Intermediate code is between the high-level and machine-level language. This intermediate code needs to be generated in such a manner that makes it easy to translate it into the target machine code.

Functions on Intermediate Code generation:

- It should be generated from the semantic representation of the source program
- Holds the values computed during the process of translation
- Helps you to translate the intermediate code into the target language
- Allows you to maintain precedence ordering of the source language
- It holds the correct number of operands of the instruction

Example

total = count + rate * 5

Intermediate code with the help of address code method is:

t1 := int_to_float(5)
t2 := rate * t1
t3 := count + t2
total := t3

5. **Code optimizer:** The next phase is code optimization or Intermediate code. This phase removes unnecessary code lines and arranges the sequence of statements to speed up the execution of the program without wasting resources. The main goal of this phase is to improve on the intermediate code to generate a code that runs faster and occupies less space.

   The primary functions of this phase are:

   - It helps you to establish a trade-off between execution and compilation speed
   - Improves the running time of the target program
   - Generates streamlined code still in the intermediate representation
   - Removing unreachable code and getting rid of unused variables
   - Removing statements that are not altered from the loop

   Example:

   Consider the following code

   a = intofloat(10)
   b = c * a
   d = e + b
   f = d

Can become

b =c * 10.0

f = e+b

6. **Code generator:** Code generation is the last and final phase of a compiler. It gets inputs from code optimization phases and produces the page code or object code as a result. The objective of this phase is to allocate storage and generate relocatable machine code.

   It also allocates memory locations for the variable. The instructions in the intermediate code are converted into machine instructions. This phase covers the optimized or intermediate code into the target language.

   The target language is the machine code. Therefore, all the memory locations and registers are also selected and allotted during this phase. The code generated by this phase is executed to take inputs and generate expected outputs.

   Example:

   a = b + 60.0

   Would be possibly translated to registers.

   MOVF a, R1

   MULF #60.0, R2

   ADDF R1, R2

Ques. 4: Please check if the grammar is left recursive or not?

A → aA’

A’ → BdA’ / aA’ / ∈

B → bB’

B’ → eB’ / ∈

Ans. 4: First (A) = {a}

(A')= {b,a, ∈)

(B) = {b}

(B') = {e,∈}

Flow (A) = { $ }

(A')= { $ }

(B) = {d}

(B') = {d}

LL(1) Table:

	a	b	d	e	$
A	A→ $aA^{'}$				
$A^{'}$	$A^{'} \rightarrow aA^{'}$	$A^{'} \rightarrow BdA^{'}$			$A^{'} \rightarrow \in$
B		$B \rightarrow bB^{'}$			
$B^{'}$			$B^{'} \rightarrow \in$	$B' \rightarrow eB^{'}$	

Hence there is no clash in the Table so it is LL(1) parsable.