National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Department of Computing

CS 354: Compiler Construction

Class: BSCS-5AB

Lab [08]: Functions using Bison

Date: 15th Nov, 2018

Time: [9:00am – 11:50am & 2:00pm – 4:50pm]

Instructor: Dr. Rabia Irfan

Lab Engineer: Mr. Azaz Farooq

National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Lab [08]: Functions using Bison

Introduction

The syntactic or the structural correctness of a program is checked in the syntax analysis phase of compilation. The structural properties of language constructs can be specified in different ways. Different styles of specifications are useful for different purposes.

Objectives

1. Successful understanding/implementation of Syntax Analysis

Tools/Software Requirement

1. Flex & Bison on Linux or Windows platform

Description

Syntax analysis or parsing, is needed to determine if the series of tokens given are appropriate in a language - that is, whether or not the sentence has the right shape/form. However, not all syntactically valid sentences are meaningful, further semantic analysis has to be applied for this purpose. For syntax analysis, context-free grammars and the associated parsing techniques are powerful enough to be used - this overall process is called parsing.

In syntax analysis, parse trees are used to show the structure of the sentence, but they often contain redundant information due to implicit definitions (e.g., an assignment always has an assignment operator in it, so we can imply that), so syntax trees, which are compact representations are used instead. Trees are recursive structures, which complement CFGs nicely, as these are also recursive (unlike regular expressions).

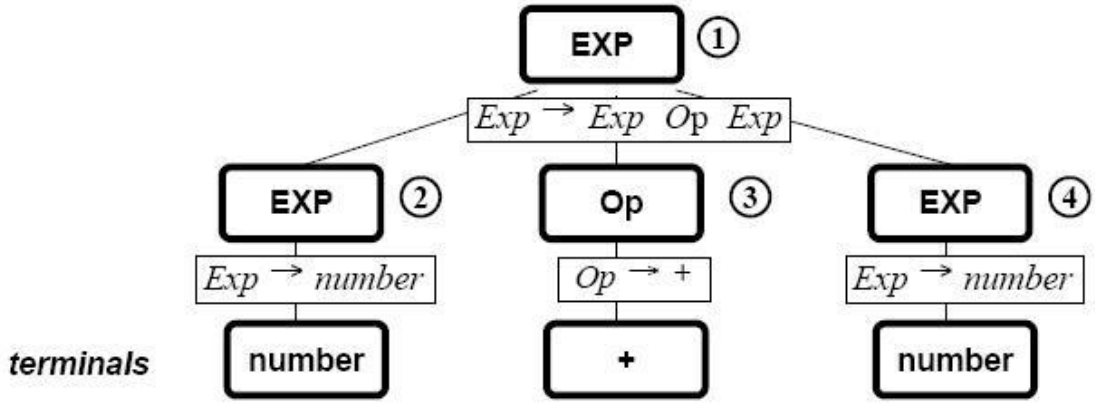
There are many techniques for parsing algorithms (vs FSA-centred lexical analysis), and the two main classes of algorithm are top-down and bottom-up parsing.

Context-free grammars can be represented using Backus-Naur Form (BNF). BNF uses three classes of symbols: non-terminal symbols (phrases) enclosed by brackets <>, terminal symbols (tokens) that stand for themselves, and the meta-symbol **::=** - *is defined to be.*

As derivations are ambiguous, a more abstract structure is needed. Parse trees generalize derivations and provide structural information needed by the later stages of compilation.

Parse Trees

Parse trees over a grammar G is a labeled tree with a root node labeled with the start symbol (S), and then internal nodes labeled with non-terminals. Leaf nodes are labeled with terminals or ε. If an internal node is labeled with a non-terminal A, and has n children with labels X1, ..., Xn (terminals or non-terminals), then we can say that there is a grammar rule of the form A → X1...Xn. Parse trees also have optional node numbers.

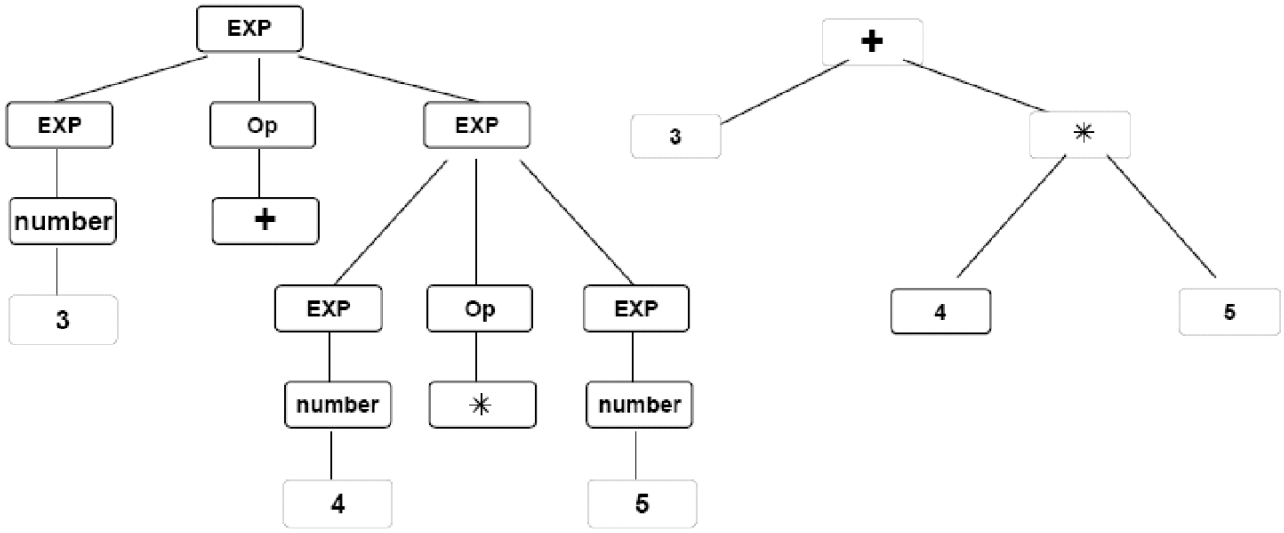


The above parse tree corresponds to a leftmost derivation.

Traversing the tree can be done by three different forms of traversal. In pre-order traversal, you visit the root and then do a pre-order traversal of each of the children, in in-order traversal, an in-order traversal is done of the left sub-tree, the root is visited, and then an in-order traversal is done of the remaining sub-trees. Finally, with post-order traversal, a post-order traversal is done for each of the children and the root is visited.

Syntax Trees

Parse trees are often converted into a simplified form known as syntax tree that eliminates un-necessary information from the parse tree.



Lab Tasks

Consider the classic grammar for arithmetic expressions:

- E → E + T | E – T | T
- T → T * F | T / F | F
- F → (E) | id

Extend the above grammar to implement exponentiation, y^x, and the functions $\log_e(x)$ and exp(x) for the natural log of x and e^x. Use the circumflex, "^", for your exponentiation operator.

Write a program using Flex, Bison, and C to implement your grammar. The input of the program consists of the expression to be evaluated and the output is the result. Use only integer or floating point constants for CONST. The lexical analyzer should convert a floating point number into a double (or float) so it should recognize signs and exponents. Any other input should generate an error.

HINTS:

- Use sscanf(), atof(), or strtod() in your lexical analyzer to convert a string to a floating point number.
- Have your lexical analyzer discard WHITE SPACE such as spaces, carriage returns and new lines (otherwise, your parser will have to process it!)
- Use „#define YYDEBUG 1" and set the variable „yydebug = 1;" to enable debug output.
- You can refer to syntax_analysis_with_bison.pdf on LMS for details on associativity and parsing examples in bison.

Deliverables

You are required to upload your task (Sources & PDF document) using the link created on LMS, may be followed by a viva.