CSC3003S Compilers – Assignment 1: Lexical and Syntactic Analysis

Introduction

In this assignment you will create a program which does lexical analysis and a program which does syntactic analysis for a made-up programming language called *bla*, for binary language.

Bla works with binary numbers and uses uppercase characters for its basic arithmetic operators: A for addition, S for subtraction, M for multiplication and D for integer division.

The lexical analyser (lexer) program should check a specified **.bla* input program file and convert it into the correct tokens, outputting the tokens to the screen and also into a corresponding **.tkn* file.

The syntactic analyser (parser) program should check a specified **.bla* input program file conforms to the specified grammar and generate an appropriate abstract syntax tree which is output to the screen and to a corresponding **.ast* file.

Tools

Python with PLY (www.dabeaz.com/ply), should be the programming language and compiler tool used for this assignment. This will allow integration with Compilers Assignment 2, which will use LLVMlite.

Input, Output and Testing

The input **.bla* source code file should be specified as a command line parameter when your programs are run, e.g.

lex_bla.py my_program.bla

parse_bla.py my_program.bla

The lexical analyser (lexer) *lex_bla.py* should check the file for tokens based on the definitions below and print each token on a new line to the screen and also in a corresponding token file, *my_program.tkn*. The details of how each token should be printed are specified below.

The syntactic analyser (parser) *parse_bla.py* should check the tokens conform to the grammar defined below and construct and output the abstract syntax tree as flat text using depth-first traversal, visiting the root, then children from left to right onto the screen and also in a corresponding file, *my_program.ast*.

Download the *bla_samples.zip* file containing **.bla* code input files, their corresponding output **.tkn* files and output **.ast* files, indicating what the output should be and to test your program.

Tokens

Identifiers

An identifier is a sequence of lowercase letters, digits and underscores, starting with either a lowercase letter or an underscore. Identifiers are also case sensitive.

Output: ID,_VALUE_

Example: If the string 'my_num' is encountered 'ID,my_num' should be the token that is output.

Numeric Literals

All numeric literals should be binary integers. Numeric literals consist of a binary integer part (with an optional preceding sign, + or -). The integer part is a sequence of one or more binary digits.

Output: BINARY_LITERAL,_VALUE_

Example: If the string '-1011' is encountered 'BINARY_LITERAL,-1011' should be the token that is output.

Operators

An operator can be one of the following operators: A S M D = ( )

Output: Each operator should be its own token.

Example: If the string '(' is encountered '(' should be the token that is output.

Whitespace

Whitespace is a sequence of non-printable characters. Non-printable characters includes:

space (' '), tab ('\t'), newline ('\n'), carriage return ('\r')...

Output: WHITESPACE

Example: If the tab character '\t' is encountered ' WHITESPACE ' should be the token that is output.

Comments

There are two forms of comments: One starts with /*, ends with */; another begins with // and goes to the end of the line (the end of line character should not be included in this token, rather it should be handled by the whitespace token.)

Output: COMMENT

Example: If the string '// a comment' is encountered ' COMMENT ' should be the token to be output.

Grammar

The slang grammar uses the notation N*, for a non-terminal N, meaning zero or more repetitions of N. Bold symbols are keywords and should form their own tokens, and other tokens are in italics.

Program → Statement* // this asterisk indicates closure

Statement → *identifier* = Expression

Expression → Expression *A* Term

→ Expression *S* Term

→ Term

Term → Term *M* Factor

→ Term *D* Factor

→ Factor

Factor → *(* Expression *)*

→ *binary*

→ *identifier*

Due

09h00, Monday 12 September 2016

Submit

Submit *lex_bla.py* and *parse_bla.py* to the automarker in a single ZIP file called ‘ABCXYZ123.zip’ (where ABCXYZ123 is YOUR student number).

Notes

An elegant solution to this assignment requires a moderate amount of code, but it is challenging and requires mastering the basics of PLY, so keep that in mind and start early.

The PLY notes contains a section on generating an abstract syntax tree. It’s only necessary to use the first method described. Once the tree is generated you’ll have to find a method of doing a tree traversal outputting it as flat text using depth-first traversal, visiting the root, then children from left to right.