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 *ula*, for unconventional language.

Ula uses unconventional characters for its basic arithmetic operators: @ for addition, $ for subtraction, # for multiplication and & for division.

The lexical analyser (lexer) program should check a specified **.ula* 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 **.ula* 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 LLVM.

Input, Output and Testing

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

```
lex_ula.py  my_program.ula

parse_ula.py  my_program.ula
```

The lexical analyser (lexer) *lex_ula.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_ula.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 *ula_samples.zip* file containing **.ula* 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 letters digits and underscores, starting with either a letter or an underscore. Identifiers are also case sensitive.

Output: ID,_VALUE_

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

Numeric Literals

All numeric literals should be recognised as floats, which will make calculations simpler. Numeric literals consist of an integer part (with an optional preceding sign, + or -), an optional decimal part, and an optional exponent suffix part respectively. The integer part is a sequence of one or more digits. The decimal part is a decimal point followed by 0, 1 or more digits. The exponent part is the character e or E followed by an optional + or - sign, followed by one or more digits. The decimal and the exponent part are both optional.

Output: FLOAT_LITERAL,_VALUE_

Example: If the string '12.34' is encountered 'FLOAT_LITERAL,12.34' should be the token that is output.

Operators

An operator can be one of the following operators: @ $ # & = ( )

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 0, 1, 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 @ Term

               → Expression $ Term
```

```
            → Term
Term        → Term # Factor
            → Term & Factor
            → Factor
Factor      → ( Expression )
            → float
            → identifier
```

Due

09h00, Monday 14 September 2015

Submit

Submit *lex_ula.py* and *parse_ula.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.