CSCI-2320: Principles of Programming Languages
Programming Assignment 2: Syntactic Analysis
Points: 20

Collaboration Level: 1 (https://turing.bowdoin.edu/dept/collab.php)
Partial Solution (2 points) Due: Thursday, October 5 (11:59pm)
Full Submission (18 points) Due: Thursday, October 19 (11:59pm)

Submission Instructions

Use Blackboard to submit the whole project in a zipped format. Include a "readme.txt" file that says how to run your program. You may include some information about the compiler/IDE you have used. We prefer a program that takes the input file name as a command line argument.

- *Unlimited submissions* are allowed. Only the last submission will be graded.
- *Partial Solution* means some (possibly non-working) code that demonstrates that you have got started and are on track for the full submission. We expect at least a skeleton.
- *Full Submission* means your last submission, which will be graded

Programming Languages

Use Python 3. You may use C, C++, or Java, but do so if you are an expert in it, because subsequent assignments will be built on top of this assignment.

The Task: Syntactic Analysis

The assignment is to build a syntactic analyzer for the CLite programming language. Intuitively, you will use the output of Programming Assignment 1, which is a stream of tokens and lexemes, as the input here. However, please treat this assignment as a completely separate project from Assignment 1. That is, do not extend/build upon Assignment 1. This is to make sure that any bug from Assignment 1 does not propagate to this assignment. Your input here will be a stream of tokens and lexemes.

The goal of this assignment is to check if the stream of input tokens is syntactically correct or not. Note that it's NOT our goal here to check the semantics of a program, e.g., whether a variable is declared before use or whether the data types in an assignment statement are consistent or not. We are rather interested in the syntax, e.g., whether the parentheses are matched or not or whether a semicolon is missing at the end of an assignment statement.

One possible direction for syntactic analysis is to implement a recursive descent (RD) parser for it. An RD parser requires that the grammar be free of left-recursions. One way of eliminating left recursions is to start with an EBNF grammar. Note that right recursions in EBNF are all right for an RD parser. An EBNF grammar for the CLite language is given on the next page. Here, Bold and italics are part of the meta-language (that is, non-terminal symbols and meta-

symbols). Terminal symbols are shown in the Courier New font. Gray color denotes optional tasks.

Program → *type* `main ( ) {` *Declarations Statements* `}`

Declarations → { *Declaration* }

Declaration → *type id* { , *id*} ;

type → `int` | `bool` | `float` | `char`

Statements → { *Statement* }

Statement → *Assignment* | *PrintStmt* | *IfStatement* | *WhileStmt* | *ReturnStmt*

PrintStmt → `print` *Expression* ;

IfStatement → `if (` *Expression* `)` *Statement* [else *Statement*]

WhileStmt → `while (` *Expression* `)` *Statement*

ReturnStmt → `return` *Expression* ;

Assignment → *id assignOp Expression* ;

id → *letter* { *letter* | *digit* }

assignOp → =

Expression → *Conjunction* { || *Conjunction* }

Conjunction → *Equality* { && *Equality* }

Equality → *Relation* [*equOp Relation*]

equOp → == | !=

Relation → *Addition* [*relOp Addition*]

relOp → < | <= | > | >=

Addition → *Term* { *addOp Term* }

addOp → + | -

Term → *Factor* { *multOp Factor* }

multOp → * | / | %

Factor → *id* | *intLiteral* | *boolLieteral* | *floatLiteral* |
(*Expression*) | *charLieteral*

letter → a | b | … | z | A | B | … | Z

digit → 0 | 1 | … | 9

intLiteral → *digit* {*digit*}

boolLiteral → `true` | `false`

floatLiteral → *intLiteral* . *intLiteral*

Note: In our lexical analyzer, the logical operators || and && were not considered. For this assignment, you must incorporate the token || for lexeme || and && for &&.

All the terminal symbols have been taken care of during lexical analysis and have been categorized into tokens, which are the inputs here. Now, all you need is to consume the tokens in a systematic way. RD parsing gives one such systematic framework. Here, for each nonterminal that does not have a corresponding token, you will define a function. For example, the nonterminal *Assignment* should have a corresponding function assignment(). This function will first try to consume the tokens id and assignOp (see the EBNF grammar on the next page). The function will then call another function for the nonterminal *Expression*. After that, the assignment() function will try to consume the token ; . If the function is able to do all of these successfully, then the assignment statement is syntactically correct. Otherwise, the assignment statement has syntactic error (such as, missing semicolon).

If you have read the textbook (Ch 3), you will notice that we are not using the FIRST(X) function for this assignment (see the textbook for the definition of FIRST). This is due to the simplicity of our CFG/EBNF. The job of the FIRST function is to help detect which production rule to apply for a nonterminal X just by looking at the next token. In more involved CFGs (not in this assignment), we would need to implement the FIRST(X) function.

You may assume that the program for which the input tokens and lexemes are given will have the following elements as per the EBNF in the previous page.

- Main function definition, with parentheses
- Variable declaration
- Assignment statements
- if statements
- while loops

Other elements, such as for loops or if statements with braces, will not be in the input, but you are very welcome to implement them if you are interested.

Sample Input (input1.txt on Blackboard): Input here is a stream of tokens and lexemes. There's a single tab character between a token and its corresponding lexeme.

```
type	int
main	main
(	(
)	)
{	{
type	int
id	x
;	;
id	x
assignOp	=
```

```
intLiteral      0
;       ;
id      x
assignOp        =
id      x
addOp +
intLiteral      10
;       ;
if      if
(       (
id      x
relOp   >
intLiteral      0
)       )
id      x
assignOp        =
boolLiteral     false
;       ;
print   print
id      x
;       ;
return  return
id      x
;       ;
}       }
```

Sample Output:

Syntactically Correct

Note that in the above example, the if-statement assigns false to an integer variable x. It is semantically incorrect, but syntactically correct. The next assignment (*not this one*) will concern semantic analysis.

Sample Codes

In Blackboard, you will find a syntactic analyzer (parser_v1.py) for very elementary arithmetic expressions implemented in Python 3 and some input files particularly for that program. Do not mistake these input files (tokens1.txt, tokens2.txt, tokens3.txt) as the input files for this assignment.