CS320 Programming Languages
Assignment 2

Due: February 29, 2020 at 11:59 PM

Objectives

In your future career some of the common tasks that you might be involved in are to analyze logs, create utilities for generating images from text descriptions, working in a project that involve Natural Language Processing NLP or write interpreters and compilers for new programming languages, and some others. These are all applications that will require you to write a parser for a file based on a specific structure to complete the task. This can be a tedious task, but fortunately there are more efficient ways to auto generate a parser.

By completing this assignment, you will practice how to write a grammar that parse a file based on a defined syntax structure. You will practice the use of a popular parser and laxer tool (ANTLR) that is widely used by professionals. It auto generates a parser and a lexer in different languages based on your target language (e.g. Java, Python, C# and JavaScript). You can use the generated lexer and parser by invoking them in different applications. Therefore, for your projects that will need a parser, you don't have to write that code and just focus your efforts on having a correct grammar file.

Skills you gain after completing this assignment:

- You will understand how different programming language detect the syntax errors
- You will learn how to write a grammar for a file based on defined syntax structure
- You will practice how to use ANTLR parser generator that helps you to create parsers.

Tasks

You are given the following **Program.txt** file (download the file). This file has a sample code for new programming Languages, the code is trying to do some mathematical calculation for a function parameter using particular programming syntax structure. ***Please write a Grammar using ANTLR4 for parsing the code and its tokens.***

//Program.txt.

```
add <- function(x, y)
{
return(x + y)
}
subtract <- function(x, y)
{
return(x - y)
}
multiply <- function(x, y)
{
return(x * y)
}
divide <- function(x, y)
{
return(x / y)
}

countdown <- function(from)
{
 print(from)
 while(from!=0)
 {
  Sys.sleep(1)
  from <- from - 1
  print(from)
 }
}
```

The following figures (in page 3) shows the expected output of a parse tree for the given program format. It also can be accessed as .pdf and .svg file where you can zoom in and out for readability.
Please note: your Grammar needs to parse the file same way as it has been given to you. *Don't change the .txt file formats*

What to Submit

1. Please submit your grammar as a g4 file via CANVAS and
2. Provide a sample screenshot for the parse tree you generated via CANVAS.

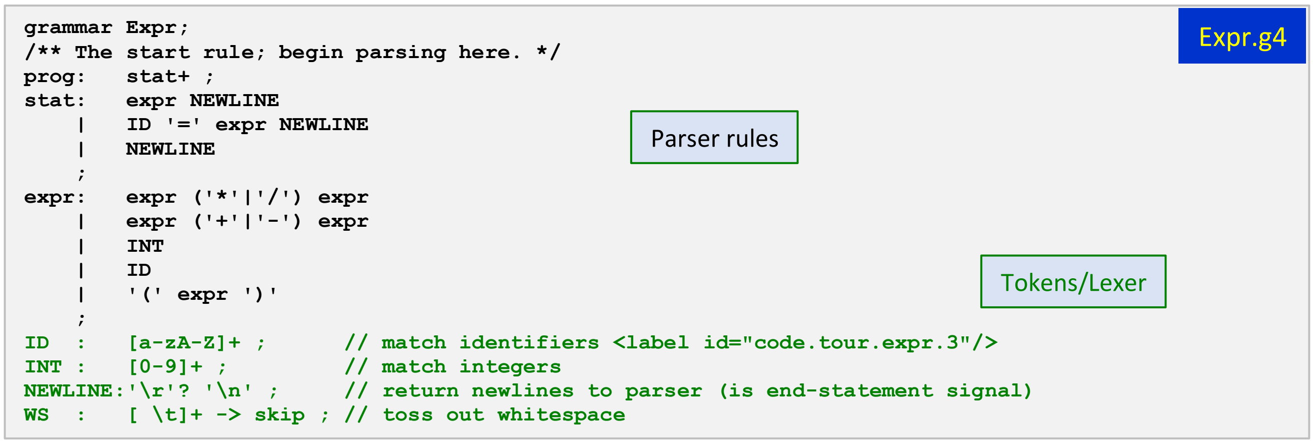
How it will be graded

To get a full grade you must provide a correct grammar that generate an AST same as the provided one. Your grammar shouldn't be ambiguous. For every missing or incorrect define rule some points will be deducted.

Useful Tips

1. Remember ANTLR resolves an ambiguity by choosing the first alternative in the grammar, so the order of the rules does matter.
2. Steps to run ANTRL and have the GUI parse tree
   a. In your cmd wind navigate to the file where you have the grammar (.g4) file and the Program.txt
   b. use the antlr4 program to generate the files that your program will actually use, such as the lexer and the parser
      antlr4 <options> <grammar-file-g4>
   c. compile lexer and parser
      javac <grammar-file>*.java
   d. You can optionally test your grammar using a little utility named TestRig (although, as we have seen, it's usually aliased to grun).
      grun <grammar-name> <rule-to-test> <input-filename(s)>

Example ANTLR Grammar File

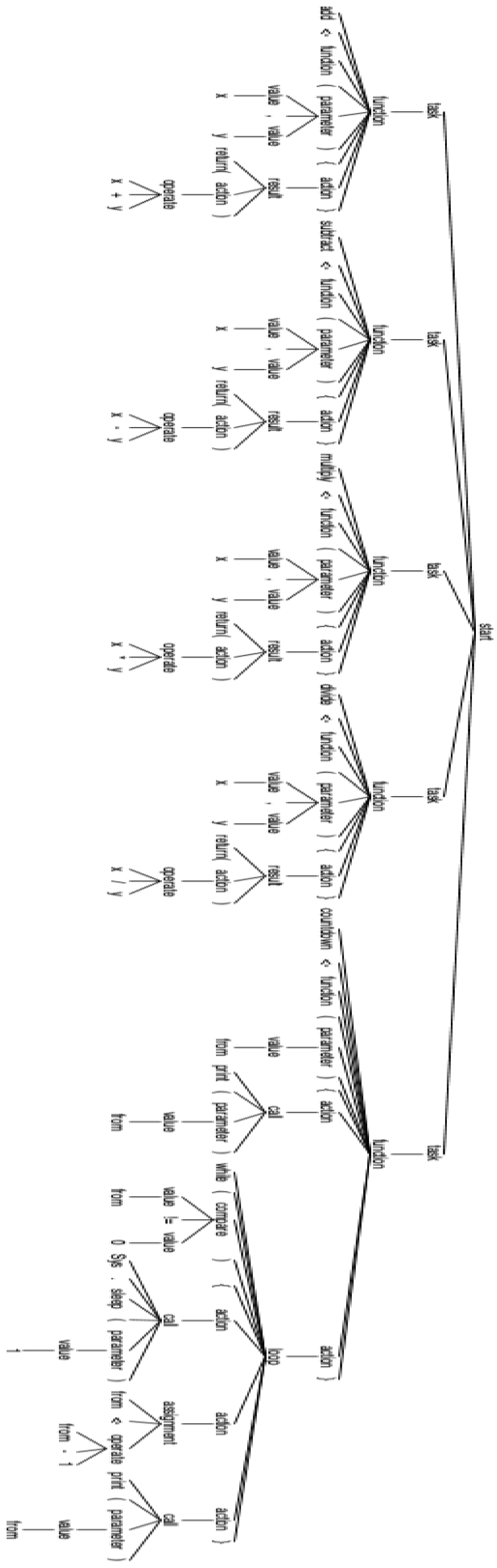


```
grammar Expr;
/** The start rule; begin parsing here. */
prog:   stat+ ;
stat:   expr NEWLINE
    |   ID '=' expr NEWLINE
    |   NEWLINE
    ;
expr:   expr ('*'|'/') expr
    |   expr ('+'|'-') expr
    |   INT
    |   ID
    |   '(' expr ')'
    ;
ID  :   [a-zA-Z]+ ;      // match identifiers <label id="code.tour.expr.3"/>
INT :   [0-9]+ ;         // match integers
NEWLINE:'\r'? '\n' ;     // return newlines to parser (is end-statement signal)
WS  :   [ \t]+ -> skip ; // toss out whitespace
```

You probably noticed that there's a mix of lower- and upper-case words in the grammar. **Lower cased** words are parser expressions, **UPPER CASE** words are lexer expressions**.** ***It is important to understand that the whole parsing is happening in two phases;***

1. the lexer creates the token stream and the parser
2. then builds the AST from this token stream.

   If your grammar is ambiguous the lexer might interpret a token wrong and then you'll see the parser complain about receiving an unexpected token while in fact it was the lexing stage where this went wrong.


start
task
function
add <- function (parameter) { action }
value , value
x y
result
return(action)
operate
x + y
subtract <- function (parameter) { action }
x - y
multiply <- function (parameter) { action }
x * y
divide <- function (parameter) { action }
x / y
countdown <- function (parameter) { action action }
value
from
call
print (parameter)
loop
while (compare) { action action action }
value != value
from 0
Sys . sleep (parameter)
1
assignment
from <- operate
from - 1
print (parameter)