Project 2 Documentation

Thomas Trask

COP4620

Spring 2013

# Project Description

The purpose of project 2 is to consume the tokenized source code created by Project 1 and develop a Lexical analyzer to check source code written in the C- language as outlined in the Compilers book (pp491-495).

# Project Specifications

The project should do the following things:

* Read a file, line by line, as provided by the first CLI argument for the program.
* Split the file into a series of Tokens, as outlined by Project 1.
* Traverse the series of tokens linearly and determine whether or not the source file specified adheres to the C- language specifications.
  + If the file adheres to the specification, the user should receive a “VALID FILE” message.
  + If the parser detects any error in the file, it exits and notifies the user of the first error it encounters.
    - NOTE: For the special case of a source file not being complete (i.e., it’s missing closing brackets for functions or something) it will display a “null” error. This error will be more verbose in Project 3.

# Execution

## Command line arguments

The application takes 1 command line argument: the input file that the program should process.

## Executing the application

The application can be run automatically with the following command

./p2 <testfilename>

java Proj2 <testfilename>

# How the program works

When the program is started, it tokenizes the input file specified when the application was started, as outlined in the Project 1 specification. Once it has tokenized the input file, it checks the token list’s syntax as specified in the C- Language specification (p491). It does this by passing the token set to a lexical analyzer which parses the tokens by using a series of rules as followed:

A-> B  
B->C B’  
C->E id ;  
E->int|void|float

\* This is not indiciative of the rule set that was used in the project. It’s just an example.

Whithin the Lexical Analyzer, each rule is represented by a function. The set of tokens that the analyzer uses is a global variable, as is the current token. The Analyzer pops the first token off of the token stack and calls A(). A() calls B() and so on until function E() is encountered. When E() is called and the current token is found to pass the rule of E (is it “int”, “void”, or “float”?), the token is “consumed” and the current token becomes the next token in the set. The analyzer continues like this until all tokens are consumed and the analyzer is in an accepting state (a.k.a. we end up back at A() with nothing left to do). At this point, the analyzer returns a “This file is valid” message to the consuming analyzer.

# Special Notes

## On Dangling Else statements

As you know, in C-like languages, there is an issue with if/else statements that causes ambiguity within the language. This program parses Else statements as they should logically be parsed. For example, the following file passes Syntactic analysis.

int main(void) {

int x;

int y;

int j;

if(x==4) if(j==4) j=3; else x=3;

if(x==4) if(j==4) j=3; else x=3; else y=3;

if(x==4){

if(j==4)

j=3;

else x=3;

}

else y=3;

return func(1, fnc(4,x));

}