**CS4013**

**Project 1**

A Lexical Analyzer for a subset of the Pascal programming language

By Carson Ellsworth

0976

**Part 1)** **Introduction:**

For the first project I had been assigned to construct a working Lexical Analyzer, referred to from on out as LA. LAs are programs that scan files line by line looking for tokens in the source program. A correctly working LA should be capable of generating tokens, creating a listing file, and detecting and printing error messages.

The LA we are implementing is a machine that is built off of sub machines, these machines are Non-Deterministic Finite Automata with epsilon transitions, known as NFAe, where e represents epsilon, creating a LA by linking these machines together with epsilon transitions, referred to from on out as e-transitions, generalizes our LA to a level that is manageable, granting us the ability to move between machines without having to consume a token along each machine change due to the nature of the e-transitions. While NFAes are extremely useful, they also will require the most amount of backtracking of the three types of machines, though this is a fairly trivial problem to solve in software.

**Part 2) Methodology:**

The LA works off of two files, a source program file, which is the file of the language that is being analyzed, and the reserved word file, a list of words that are special to the language. The LA reads in the source file line by line into a 72-character buffer, per special restrictions, and process the character stream. This stream is then processed by the LA till all tokens and errors are properly handled till a new-line character is found, which the process then repeats itself, working down the source file till the EOF symbol is found.

There are a few restrictions that need to be kept in mind when working on this project as per project guidelines. First and previously mentioned is that the character buffer has a max size of seventy-two characters, in reality though that means a source code line can only be 70 characters long because each stream of characters must have a newline character and termination character. Second, any identifier can only be ten characters long at max and must start with a letter. Third, all integers have a max size of ten digits and cannot possess a leading zero, and finally, Reals have a max before dot length of five where a leading zero can exist if and only if it is the only before the dot character, a max after dot length of five with no trailing zeros, and if they contain an exponent it can only be two digits.

**Part 3) Implementation:**

Each token recognizer was first designed on paper as an NFAe, through multiple iterations of machine designing, I was able to get the hang of being able to visualize and create the machines on paper and then transcribe them into approximate C pseudo-code. This repetitive process allowed me to get the program flow of the machines engrained in my head before going gung-ho in coding.

The tokens generated are then handled one of few ways, if the token is a reserved word, then the reserved word lexeme is returned and processed by printing its information to the token file. Now if the token is an identifier, the process is similar but there are a few more hoops to jump through, this is because we need to make sure the identifier is not a reserved word and that it is not already been previously identified. To do this I have implemented a simple symbol table that keeps track of what identifiers have already been created in the source program, this symbol table works as a linked list, so anytime I need to check if a word is an identifier I just traverse the list and check each element. Given that an element is not in the list, then simply create it, otherwise that element will then reference the already created element in the table.

Integers and reals work in similar ways, if a number is recognized that is not a part of an identifier name, then the real machine will run first, this follows the rule of checking subsets before supersets. Given that the numbers do not break any of the special restriction rules then they are printed out to the listing and token files. However, they are going to need to be dealt with in the later projects.

A relop and catchall machine were also going to be needed for identifying the many symbols of the pascal language. These machines were by far the easiest to create and only required nesting and chaining together if and else statements in the correct order so the machines could search the text for the associative token. The relop machine was designed to handle the Boolean comparison symbols such as <> (not equals) < (less than) and others while the catchall machines were built to catch all of the other symbols of the language as well as identify errors such as unknown symbols.

**Part 4) Discussion and Conclusions:**

This is the first major program I have constructed in C and it has been a real challenge learning how the language features interact with one another, I have learned a great deal about the programming language and have come to realize how powerful of a language it is due it its nature of being a bridge between low and high level languages.

Before completing the code for project 1, I had no idea how intricate the software has to be to make such a machine work. Now I feel like writing a compiler is equated to creating a piece of art, where every line of code as a unique and critical purpose for making the machine function as a whole. This project is the first step in creating a one-pass compiler. Before taking this course, compilers and programming language making as just been pure magic, but now I am starting to see what is happening behind the scenes and I am finding this whole experience to be really amazing.

**Part 5) References:**

Aho, A., Sethi R., Ullman J. (1986) Compilers Principles, Techniques, and Tools. Reading, MA: Addison-Wesly

**Appendix I: Sample Inputs and Outputs**

1. Program with no lexical errors

program example(input, output);

var x, y: integer;

function gcd(a, b: integer): integer;

begin

if b = 100 then gcd := a

else gcd := gcd(b, a mod b)

end;

begin

read(x, y);

write(gcd(x, y))

end.

1. Program with no lexical errors

program example(input, output);

var x, y: integer;

function gcd(a, b: integer): integer;

begin

if b <><><><> = := 100 then gcd := a

else gcd := gcd(b, a mod b);;;;;

end;

begin

read(x, y)))));

write(gcd(x, y))

end.

1. Program with lexical errors

program example(input, output);

var x, y: integer;

function gcd(a, b: integer): integer;

begin

if b !@#$%^& 100 then gcd := a

else gcd := gcd(b, a mod b);

end;

b:=10000000000000

c:=1.22222E333

e:=1.33333333

f:=1111111.03

g:=00001.0

h:=1.000

begin

read(x, y);

write(gcd(x, y))

end.

1. Program with lexical errors

program example(input, output);

var x, y: integer;

function gcd(a, b: integer): integer;

begin

if b := 100 then gcd := a

else gcd := gcd(b, a mod b);

end;

abcdefghijklmnopqrstuvwxyz:=26

begin

read(x, y);

write(gcd(x, y))

end.

**Appendix II: Program Listings**