Project 1 Documentation

Thomas Trask

COP4620

Spring 2013

# Project Description

The purpose of project 1 is to develop a lexical analyzer that will effectively tokenize source code written in the C- language as outlined in the Compilers book (pp491-495).

# Project Specifications

For the purpose of this exercise, a token is defined as any string that will eventually be processed as an individual “letter” of a compiler grammar. The lexical analyzer should do the following:

* Read a file, line by line, as provided by the first CLI argument for the program.
* Process each line within the file individually.
* Split each line into tokens as follows:
  + Recognize the following case-sensitive **keywords** as their own tokens:
    - float, Int, float, else, if, return, while
  + Recognize the following **special characters** as their own tokens:
    - +, -, \*, /, <, >, ;, ., <= >, +., ==, !=, ), (, }, {, [,]
  + Detect the following **comment** special characters:
    - /\*, \*/, //
  + Recognize both float values and integer values as **numbers** and treat them as tokens.
  + Detect **symbol** tokens as defined below:
    - A symbol must not contain any special characters or keyword (as defined above)
    - A symbol must start with a letter.
    - A symbol must only contain letters and numbers.
* Detect invalid characters within a **token**.
* Display the each line and the tokens that it contains.

# 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

./p1 <testfilename>

Java Proj1 <testfilename>

# How the program works

The program starts by reading the test file specified in, line by line. For each line, the program splits the line into separate tokens, originally by spaces based on the keywords, special characters and specifications specified previously. If there are no spaces separating each token, spaces are added and the token set is reprocessed.

# Special Notes

## Code block depth

Code blocks are parts of valid code grouped within curly brackets. When a “{“ token is encountered, the code block depth is incremented by 1; conversely, when a “}” token Is encountered, the code depth is decremented by 1. The code depth for all tokens is tracked.

## Symbols & the Symbol Table

Symbols will eventually represent the variables within the source code. At the moment, they’re little more than tokens flagged as identifiers. After the input file is parsed, symbol tokens are gathered and added to a symbol table on the condition that their combined name and code depth are unique. This symbol table is then displayed at the end of the output of the program, along with the code depth of each symbol.

## Floating Numbers

As outlined in the project specification, a float is a number that conforms to the following specification:

* Any number of digits.
* An optional decimal (.) along with any number of digits after the decimal, if provided.
* Either a lower-case or upper-case E
* An optional (+/-)
* Any number of digits

Valid Floats are:

* 3E5
* 3.2E5
* 5E-2
* .5E1
* 4E+3

Invalid Floats are:

* 3.E5
* 0.
* 4E5.2
* 4E4.

## Comments

Comment blocks represent source code that should not be tokenized by the lexical analyzer. Comments can be either single-line or multi-line. Multi-line comments are started with the “/\*” string and ended with the “\*/” string. Multiline comments can be nested. For instance, in the following string “/\* /\* return \*/ \*/”, return is within the second nested **depth** of the comment. Single-line comments are started by the “//” string and, unless already located in a comment, anything after the “//” on the line is not tokenized.

### Comment depth

As noted above, comments can be nested, increasing the global comment depth. Until the comment depth reaches 0, no code will be analyzed.

### Special note about starting comments

To nest comments within each other the start-comment strings (“/\*”) must be separated by any number of characters greater than 0. For example, the following code “/\*/\*” will not process correctly but the code “/\* /\*” will. This is not true for the closing-comment string, “\*/”. The code “\*/\*/” is completely valid and, if the comment depth is greater than 2, will be processed as “comment-end” tokens. This was a late-stage design decision in lieu of the non-determinism with the code “/\*/\*\*/\*/”. This should be fixed in the next release.

# Sample Input

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*return/\* /\* \*/\*/\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* /\*

i = 333; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ \*/

//iiii = 3@33;

Float f5 = 3

float f = 5e5; //proper form.

inti 5;

int g 4 cd (int u, int v) {

/\*

if(v == >= 0) return/\*a comment\*/\*/v+ u;

return /\*a comment\*/ u;

return /\*acomment u;

else ret\_urn gc\*/d(vxxxxxxvvvvv, u-u/v\*v);

/\* u-u/v\*v == u mod v\*/

{

val x=3;

int v=4;

}

!

}

j[v] = 4; j[test^2] =5

JamesMay[test2] =5

j{test} = 3;

testtwo[test] = 2;

[e.g., int parse(int x, int y){ x=4;}]

ftwo = 5.0E-1

ftwo = 5.E

float 1direction;

0.5E4

0.3

.4e5.0

# Sample Output

Project Started

Attempting to tokenize Input File: test1

INPUT: /\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INPUT: /\*return/\* /\* \*/\*/\*/

INPUT: /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* /\*

INPUT: i = 333; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ \*/

INPUT: //iiii = 3@33;

INPUT: Float f5 = 3

ID: Float

ID: f5

=

Num: 3

INPUT: float f = 5e5; //proper form.

Keyword: float

ID: f

=

Num: 5e5

;

INPUT: inti 5;

Keyword: int

ID: i

Num: 5

;

INPUT: int g 4 cd (int u, int v) {

Keyword: int

ID: g

Num: 4

ID: cd

(

Keyword: int

ID: u

,

Keyword: int

ID: v

)

{

INPUT: /\*

INPUT: if(v == >= 0) return/\*a comment\*/\*/v+ u;

ID: v

+

ID: u

;

INPUT: return /\*a comment\*/ u;

Keyword: return

ID: u

;

INPUT: return /\*acomment u;

Keyword: return

INPUT: else ret\_urn gc\*/d(vxxxxxxvvvvv, u-u/v\*v);

ID: d

(

ID: vxxxxxxvvvvv

,

ID: u

-

ID: u

/

ID: v

\*

ID: v

)

;

INPUT: /\* u-u/v\*v == u mod v\*/

INPUT: {

{

INPUT: val x=3;

ID: val

ID: x

=

Num: 3

;

INPUT: int v=4;

Keyword: int

ID: v

=

Num: 4

;

INPUT: }

}

INPUT: !

Error: !

INPUT: }

}

INPUT: j[v] = 4; j[test^2] =5

ID: j

[

ID: v

]

=

Num: 4

;

ID: j

[

Error: test^2

]

=

Num: 5

INPUT: JamesMay[test2] =5

ID: JamesMay

[

ID: test2

]

=

Num: 5

INPUT: j{test} = 3;

ID: j

{

ID: test

}

=

Num: 3

;

INPUT: testtwo[test] = 2;

ID: testtwo

[

ID: test

]

=

Num: 2

;

INPUT: [e.g., int parse(int x, int y){ x=4;}]

[

Error: e.g.

,

Keyword: int

ID: parse

(

Keyword: int

ID: x

,

Keyword: int

ID: y

)

{

ID: x

=

Num: 4

;

}

]

INPUT: ftwo = 5.0E-1

ID: ftwo

=

Num: 5.0E-1

INPUT: ftwo = 5.E

ID: ftwo

=

Error: 5.E

INPUT: float 1direction;

Keyword: float

Error: 1direction

;

INPUT: 0.5E4

Num: 0.5E4

INPUT: 0.3

Num: 0.3

INPUT: .4e5.0

Error: .4e5.0

Symbol Table

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Name Depth

Float 0

f5 0

f 0

i 0

g 0

cd 0

u 0

v 0

v 1

u 1

d 1

vxxxxxxvvvvv 1

val 2

x 2

v 2

j 0

JamesMay 0

test2 0

test 1

testtwo 0

test 0

parse 0

x 0

y 0

x 1

ftwo 0