# μC: A Simple C Programming Language

## Programming Assignment I

## Lexical Definition

Due Date: 23:59, April 14, 2017

Your assignment is to write a scanner for the ***μC*** language in **lex**. This document gives the lexical definition of the language, while the syntactic definition and code generation will follow in subsequent assignments.

Your programming assignments are based around this division and later assignments will use the parts of the system you have built in the earlier assignments. That is, in the first assignment you will implement the scanner using **lex**, in the second assignment you will implement the syntactic definition in **yacc**, and in the last assignment you will generate assembly code for the Java Virtual Machine by augmenting your **yacc** parser.

**This definition is subject to modification as the semester progresses**. You should take care in implementation that the programs you write are well-structured and easily changed.

# 1. Lexical Definitions

Tokens are divided into two classes:

- tokens that will be passed to the parser, and
- tokens that will be discarded by the scanner (i.e. recognized but not passed to the parser)

## 1.1 Tokens that will be passed to the parser

The following tokens will be recognized by the scanner and will be eventually passed to the parser:

### Delimiters

Each of these delimiters should be passed back to the parser as a token.

| | |
|---|---|
| comma | , |
| semicolon | ; |
| parentheses | ( ) |
| quotation | " " |
| Next line | \n |

## Arithmetic, Relational, and Logical Operators

| | |
|---|---|
| arithmetic | + - * / |
| exponentiation | ^ |
| remainder | % |
| relational | < <= >= > == != |
| assignment | = |

Each of these operators should be passed back to the parser as a token.

## Keywords

The following keywords are reversed words of μC :

**int double print if else while**

Each of these keywords should be passed back to the parser as a token.

## Identifiers

An identifier is a string of letters and digits beginning with a letter. (Case of letters is relevant, i.e., ident, Ident, and IDENT are not the same identifier. Note that keywords are not identifiers.)

## Integer Constants

A sequence of one or more digits.

## String Constants

A string constant is a sequence of zero or more ASCII characters appearing between double-quote (") delimiters. A double-quote appearing with a string must be written after a ". (i.e. "abc" 、"Hello world").

## 1.2 Tokens that will be discarded

The following tokens will be recognized by the scanner, but should be discarded rather than passing back to the parser.

## Whitespace

A sequence of blanks (spaces), tabs, and newlines.

### Comments (Advanced features)

Comments can be denoted in several ways:

- C-*style* is text surrounded by "*/**" and "***/*" delimiters, which may span more than one line;
- C++-*style* comments are a text following a "*//*" delimiter running up to the end of the line.

Whichever comment style is encountered first remains in effect until the appropriate comment close is encountered. For example,

***// this is a comment // line */ /* with /* delimiters */ before the end***

and

***/* this is a comment // line with some /* and C delimiters */***

are both valid comments.

### Other characters

The undefined characters or strings should be discarded by your scanner during parsing.

## 2. Symbol Tables

You must implement symbol tables to store all identifiers. Symbols tables should be designed for efficient insertion and retrieval operations, and hence they are usually organized as hash tables. In order to create and manage the tables, at least the following functions should be provided:

**create( )**: Creates a symbol table. Declare a data structure and assign the memory space to store variable.

**insert(s)**: Inserts s into a new entry of the symbol table.

**lookup(s)**: Returns index of the entry for string s, or nil if s is not found.

**dump( )**: Dumps all entries of the symbol table.

| *Symbol Table example* | | |
|---|---|---|
| **Index** | **ID** | **Type** |
| 0 | Cat | int |
| 1 | Dog | double |

## 3. What should your scanner do?

Your scanner are expected to offer the **basic features**.

To get **bonus points**, your scanner should be able to provide the **advanced features**.

### Basic features (100pt)

- Print the recognized token on a separate line and discard whitespace and undefined character sets. (60pt)
  (The token could be delimiter, operator, keyword, identifier, or constant.)
- Recognize the type of scanned values (***int*** and ***double***). (10pt)
- Create and insert symbol tables. (20pt)
  (Your *create* and *insert* functions need to print out related messages.)
- Dump and lookup function for the symbol table. (10pt)
  (At the end of the file scan, your scanner should print out the contents in the symbol table.)

The example input code and the corresponding output that we expect your scanner to generate are shown in the next page.

### Advanced features (30pt)

- Allow case-insensitive keywords, e.g., IF = if = iF. (10pt)
- Discard comment. (20pt)
  (You need to print out the contents of the comments and ignore them during parsing.)

## Example input code and the expected output from your scanner:

**Input**:

```
int a = 3;
double b = 7.0;
DOUBLE sum = a + b;
print(sum);
while(a < b)
  a++;
```

**Output**:

a int TYPE VAR
Create a symbol table
Insert a symbol: a
= ASSIGN
3 NUMBER
; SEMICOLON
b double TYPE VAR
Insert a symbol: b
= ASSIGN
7.0 FLOATNUMBER
; SEMICOLON
sum DOUBLE TYPE VAR
Insert a symbol: sum
= ASSIGN
a ID
+ OPERATOR
b ID
; SEMICOLON
print PRINT FUNCTION
( LB
sum ID
) RB
; SEMICOLON
while WHILE FUNCTION
( LB
a ID
< RELATIONAL
b ID
) RB
a ID
+ OPERATOR
+ OPERATOR
; SEMICOLON

Parse over, the line number is 6.

The symbol table dump:
1 a int
2 b double
3 sum DOUBLE

# 4. lex Template

You can download this template file on the Moodle.

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
%}

letter [a-zA-Z]
digit [0-9]
id {letter}+({letter}|{digit})*
number                {digit}+

%%

";"                             {printf("%s is a sem \n",yytext); }
{id}                            {printf("%s is a ID \n",yytext); }
{digit}+                        {printf("%s is a number \n",yytext);}
"="                             {printf("%s is an assign \n",yytext);}
"+"                             {printf("%s is a operator \n",yytext);}
[\n]                            {}

%%

main(int argc,char *argv[]){

        yyin = fopen(argv[1],"r");
        yylex();
}

yywrap(void) {
    return 1;
}
```