Department of Computer Science and Engineering
Indian Institute of Technology Guwahati

CS348

Assignment - 6: Lexer for micro C

1 Preamble – micro C

This assignment follows the lexical specification of C language from the International Standard **ISO/IEC 9899:1999 (E)** with some minor modifications. To keep the assignment simple, we have chosen a subset of the specification as given below. We shall refer to this language as micro C.

The lexical specification quoted here is written using a precise yet compact notation typically used for writing language specifications. We first outline the notation and then present the Lexical Grammar that we shall work with.

2 Notation

In the syntax notation used here, syntactic categories (non-terminals) are indicated by *italic type*, and literal words and character set members (terminals) by **bold type**. A colon (*:*) following a non-terminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words "one of". An optional symbol is indicated by the subscript "opt", so that the following indicates an optional expression enclosed in braces.

{ $expression_{opt}$ }

3 Lexical Grammar of micro C

1. **Lexical Elements**

   token:
   - *keyword*
   - *identifier*
   - *constant*
   - *string-literal*
   - *punctuator*

2. **Keywords**

   keyword: one of

return	**void**	**float**	**integer**
char	**for**	**const**	**while**
bool	**if**	**do**	**else**
begin	**end**		

3. **Identifiers**

   identifier:
   - *identifier-nondigit*
   - *identifier identifier-nondigit*
   - *identifier digit*

   identifier-nondigit: one of

_	a	b	c	d	e	f	g	h	i	j	k	l	m
	n	o	p	q	r	s	t	u	v	w	x	y	z
	A	B	C	D	E	F	G	H	I	J	K	L	M
	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

   digit: one of

   0 1 2 3 4 5 6 7 8 9

4. Constants

constant:
  integer-constant
  floating-constant
  enumeration-constant
  character-constant
integer-constant:
  nonzero-digit
  integer-constant digit
nonzero-digit: one of
  1 2 3 4 5 6 7 8 9
floating-constant:
  fractional-constant exponent-part$_{opt}$
  digit-sequence exponent-part
fractional-constant:
  digit-sequence$_{opt}$ **.** *digit-sequence*
  digit-sequence **.**
exponent-part:
  e *sign*$_{opt}$ *digit-sequence*
  E *sign*$_{opt}$ *digit-sequence*
sign: one of
  $+$ $-$
digit-sequence:
  digit
  digit-sequence digit

  ' *c-char-sequence* **'**
c-char-sequence:
  c-char
  c-char-sequence c-char
c-char:
  any member of the source character set except
    the single-quote ', backslash \, or new-line character
  escape-sequence
escape-sequence: one of
  **\' \" \? **
  \a \b \f \n \r \t \v

5. String literals

string-literal:
  " *s-char-sequence*$_{opt}$ **"**
s-char-sequence:
  s-char
  s-char-sequence s-char
s-char:
  any member of the source character set except
    the double-quote ", backslash \, or new-line character
  escape-sequence

6. **Punctuators**

   punctuator: one of

   ```
   [  ]  (  )  ->  ++  --  &  *  +  -  !
   /  %  <<  >>  <  >  <=  >=  ==  !=  ^  |  &&  ||
   ?  :  ;  =  ,
   ```

7. **Comments**

   (a) *Multi-line Comment*

   Except within a character constant, a string literal, or a comment, the characters `/*` introduce a comment. The contents of such a comment are examined only to identify multibyte characters and to find the characters `*/` that terminate it. Thus, `/* ... */` comments do not nest.

   (b) *Single-line Comment*

   Except within a character constant, a string literal, or a comment, the characters `//` introduce a comment that includes all multibyte characters up to, but not including, the next new-line character. The contents of such a comment are examined only to identify multibyte characters and to find the terminating new-line character.

4 Changes with respect to C

1. keywords are specified differently
2. Instead of braces { and }, we are using begin and end respectively.

5 The Assignment

1. Write a flex specification in both iterative and non-iterative ways for the language of micro C using the above lexical grammar. The name of your file should be a6*it_roll*.l (for iterative implementation) and a6*nit_roll*.l (for non-iterative implementation).
2. Write your `main()` (in a same .l) to test your lexer.
3. Your code should also print a symbol table.
4. Prepare a Makefile to compile the specifications and generate the lexer.
5. Prepare a test input file a6_*roll*_test.mc that will test all the lexical rules that you have coded.
6. Prepare a zip file with the name a6_*roll* containing all the above files and upload to Moodle.

6 Example for iterative and non-iterative implementation

6.1 Iterative

```
    %{
#include <stdio.h>
#define INT 1
#define NUM 2
#define ID 3
#define ASSIGN 4
%}

%%

int     { return INT; }
```

```
[0-9]+  { return NUM; }
[a-zA-Z_][a-zA-Z0-9_]* { return ID; }
= { return ASSIGN;}
; {printf("SPECIAL: ;\n");}

%%

int main()
{
 int token;
 while(token=yylex())
 {
  switch(token)
  {
  case INT:
   printf("<KEYWORD,%d, %s>",token, yytext);   break;

  case ID:
   printf("<IDENTIFIER,%d, %s>",token, yytext);  break;

  case NUM:
   printf("<INTEGER_CONSTANT, %s>", yytext); break;

      case ASSIGN:
   printf("<OPERATOR, %s>", yytext); break;

//  case WS: break;
  default:
   printf("<INVALID_TOKEN, %s>", yytext);   break;
  }

printf("\n");
 }
  return 0;
}
```

6.2 Non-Iterative

```
    %{
#include <stdio.h>
%}
INT int
letters [_A-Za-z]

%%

{INT}     { printf("Keyword: INT\n"); }
[0-9]+  { printf("Integer: %s\n", yytext); }
{letters}[a-zA-Z0-9_]* { printf("Identifier: %s\n", yytext); }
= { printf("OPERATOR: =\n");}
; {printf("SPECIAL: ;\n");}

%%

int main() {
  yylex();
  return 0;
}
```