**컴파일러의 기초: Project 1,lexical analyzer (Due: 10/9)**

이 프로젝트의 목적은 lex utility를 이용하여 (C의 feature를 가감한; subc라 부른다)에 대한 lexical analyzer를 만드는 데 있다. lex manual을 참고하여 lex input인 subc.l을 만들고, lex를 이용하여 lexical analyzer를 만든 뒤, subc로 작성된 program에서 모든 token들을 찾아내서 output으로 print 한다.

**Lexical Structure of subc**

* **Definitions**  
  letter A-Z, a-z, \_  
  alnum A-Z, a-z, 0-9, \_  
  digit 0-9  
  white space space, tab(\t), newline(\n)  
  integer-constant   
  float-constant
* **Operators**  
  ( ), [ ], { }, ->, ., ,, .., !, ~, ++, --, \*, /, %, +, -,   
  <<, >>, <, <=, >, >=, ==, !=, &, ^, |, &&, ||, ?, :, ;,   
  =, +=, -=, \*=, /=, %=, &=, ^=, |=, <<=, >>=
* **Comments**  
  /\* ... \*/ (Nesting을 허용하므로 /\*Hong/\*lastname\*/Kil–Dong\*/ is OK)
* **Keywords**  
  ‘auto’,‘break’,‘case’,‘char’,‘continue’,‘default’,‘do’,‘double’,‘else’, ‘extern’,‘float’,‘for’,‘goto’,‘if’,‘int’,‘long’,‘register’,‘return’,‘short’ ‘sizeof’,‘static’,‘struct’,‘switch’,‘typedef’,‘union’,‘unsigned’,‘void’,‘while’

**A Few Notes**

**1. Identifier and Key words**

* Identifier와 Keyword에 대하여 같은 pattern을 사용하여 recognize 한다.
* 이 pattern에 맞는 lexeme을 읽게 되면 Hash Table을 찾아 본다.
* Hash Table에 그 lexeme이 존재하지 않으면 새로운 entry를 만들어 insert하고 그 entry의 tokentype에 Identifier 임을 나타낸다. 만약 Hash Table에 존재한다면 그 entry의 tokentype을 통해 Identifier인지 혹은 Keyword인지를 확인한다. 각 entry는 reference count를 가지고 있어 그 lexeme이 reference된 count를 기록한다.
* 따라서 모든 keyword들은 lexer가 읽기를 시작하기 전에 이미 Hash Table에 insert되어 있어야 한다.
* Hash Table로의 interface는 enter(flag, string of lexeme, length of the lexeme)을 통해서 프로그램 한다.(flag: id or keyword)

**2. Comments**

C의 comment는 nesting이 허락되지 않으나 이번 숙제에는 nesting이 가능한 comment를 recognize 해야 한다. nested comments에 대한 regular expression을 작성할 수 없으므로 다른 방법을 사용한다. lex의 start condition을 이용하여 normal mode와 comment mode를 구별하고 normal mode로 lex가 시작된다. Input 중에 “/\*” 를 읽는 순간 comment mode가 시작되면서 comment mode에서는 세가지 동작이 이루어진다: (1) 또 다른 “/\*”를 읽게 되면 depth count를 증가시키고 (2) “\*/” 를 읽게 되면 depth count를 감소시키며 만약 depth count가 0이 되면 normal mode가 시작된다. (3) 그 외의 모든 character들은 무시된다. Lex가 처음 시작될 때 normal mode에서 시작되기 위해 main()을 lex input의 procedure section에 넣고 main()에서 start mode를 set 한다. (main()에는 yylex() call도 있어야 함)

**3. DOTDOT and Float Const**

우리의 float const의 정의는 C의 정의와 약간 다르다. (.325를 허락 안함) 1..2는 “float (1.)” “.” “integer” 로 이해되면 안되고 “integer” “..” “integer”로 해석되어 range를 나타낼 수 있어야 한다. 이를 위해 lex의 lookahead operator(/) 를 이용한다.

**4. Output**

빠르고 정확한 채점을 위해서 출력 형식을 맞추도록 한다. 만약에 출력 기준을 어길 시, 프로그램이 잘 동작 하더라도 감점(10%)을 당하게 되니 주의하도록 한다. 출력 형식은 다음과 같다.

{Token의 종류}\t{token의 값}\t{프로그램에서 나타난 횟수(ID와 KEY에 대해서만 출력)}

**Your Lex Program**

주어진 input subc 프로그램에서 위의 모든 token들을 recognize하여 standard out에 다음과 같이 print한다. “token type”, “lexeme”, “reference count” (“reference count”는 identifier나 keyword인 경우에만 필요하다)

**1) Makefile template**

subc: lex.yy.o hash.o

gcc -o subc lex.yy.o hash.o -ll

lex.yy.o: lex.yy.c

gcc -c -g lex.yy.c

lex.yy.c: subc.l

flex subc.l

hash.o: hash.c hash.h subc.h

gcc -c -g hash.c

clean:

rm –f lex.yy.c

rm –f \*.o

rm –f subc

**2) subc.l template**

%{

#include “subc.h”

int commentdepth = 0; /\* depth of comment nesting \*/

%}

...

%%

..

%%

main()

{

...

yylex();

...

}

**3) subc.h template**

#include <stdio.h>

#include <strings.h>

typedef struct id {

int tokenType;

char \*name;

int count;

} id;

id \*enter(int tokenType, char\* lexeme, int length);

**4) hash.c**

#include "subc.h"

#define HASH\_TABLE\_SIZE 101

typedef struct nlist {

struct nlist \*next;

id \*data;

} nlist;

static nlist \*hashTable[HASH\_TABLE\_SIZE];

id \*enter(int tokenType, char \*name, int length) {

/\* implementation is given here \*/

}

hash table의 구현에 대한 제약사항은 없으며 전체적인 프로그램의 동작에 지장을 주지 않는 범위 내에서 자유롭게 구현이 가능하다.

**Example**

**입력**

|  |
| --- |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  /\* nested comments\*/  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  struct \_point {  float x, y, z;  int color;  } point[20];  struct \_line {  struct \_point \*p[2];  int color;  float meter = 0.5;  } line[20];  1..50 |

**출력**

|  |  |
| --- | --- |
| KEY struct 1  ID \_point 1  OP {  KEY float 1  ID x 1  OP ,  ID y 1  OP ,  ID z 1  OP ;  KEY int 1  ID color 1  OP ;  OP }  ID point 1  OP [  INT 20  OP ]  OP ;  KEY struct 2  ID \_line 1  OP {  KEY struct 3  ID \_point 2 | OP \*  ID p 1  OP [  INT 2  OP ]  OP ;  KEY int 2  ID color 2  OP ;  KEY float 2  ID meter 1  OP =  F 0.5  OP ;  OP }  ID line 1  OP [  INT 20  OP ]  OP ;  INT 1  OP ..  INT 50 |

token의 종류(OP, INT, …)와 값(\*, 1, …), 횟수는 반드시 tab으로 구분한다.