UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA

A SIMPLE

{C}

LEXER

Corso di Compilatori

Anno accademico

2015/2016

Supervisor

Prof. Gennaro Costagliola

Studenti

Rosario Di Florio

Vincenzo Venosi

INDICE

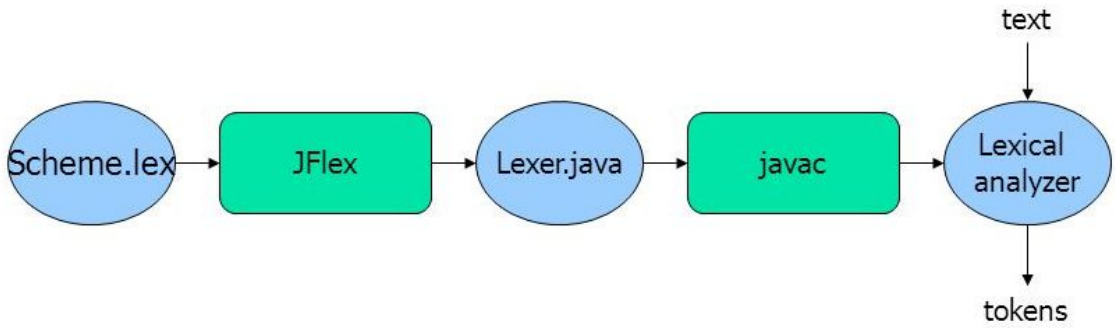
Obiettivi

L' obiettivo di questo lavoro è stato quello di creare un Lexer che fosse in grado di effettuare l'analisi lessicale di un qualsiasi file C conforme allo standard ANSI C89. Da premettere che nel nostro lavoro l'attività svolta dal Pre-processore C non è stata presa in considerazione e quindi i test sono stati effettuati su codice C non avente le direttive del Pre-processore (#include, etc.).

Progettazione e implementazione

Realizzazione del Lexer

Per realizzare il Lexer è stato usato il tool JFlex, un generatore di analizzatori lessicali che permette di realizzare un analizzatore lessicale integrabile in un qualunque programma Java.



Esso prende in input un file di specifiche e genera in output il lexer vero e proprio.

Il Reference Manual

Il primo passo consiste nel recuperare e analizzare il reference manual del linguaggio C.

https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

Per la realizzazione del Lexer la nostra attenzione si è focalizzata quindi sugli elementi lessicali del linguaggio descritti all'interno del manuale, il quale descrive gli elementi lessicali che compongono il codice sorgente C dopo la fase di pre-processing.
Nel GNU reference manual troviamo i seguenti elementi lessicali:

Struttura del file .jflex

Classi importate

Il progetto prevede l'utilizzo della libreria **java_cup.runtime**, per poter iniziare a utilizzare il file dei simboli *sym.java*, questo file di solito, viene automaticamente generato dalla specifica .cup in cui vengono definiti terminali e non terminali della grammatica che definiranno i token a cui poter assegnare un id da poter utilizzare in seguito.
Al momento di questa realizzazione non è stato utilizzata una grammatica e quindi non è stato creato il file .cup per questioni di semplicità di testing, i token al momento vengono definiti direttamente all'interno del file sym.java.

La classe **java.util.Hashtable**, che realizza l'implementazione Java della struttura dati HashMap, e che viene utilizzata per realizzare la tabella dei simboli

Direttive JFlex Utilizzate

%class CLexer
Usata per definire il nome della classe Java che implementa il Lexer.
%unicode
Usata per la compatibilità con i caratteri unicode.
%debug
Utilizzata per stampare l'azione che il Lexer avrebbe compiuto se non fosse stato in modalità debug.
%line e %column
Usate per il conteggio automatico della posizione dell'input corrente all'interno del file. Jflex salverà il conteggio della riga e della colonna corrente nelle variabili yyline e yycolumn.
%cup passa alla modalità di compatibilità CUP per interfacciarsi con un parser CUP.

Definizioni regolari

Sono state costruite le seguenti definizioni regolari:

OctDigit	=	[0-7]
HexDigit	=	[0-9a-fA-F]
letter	=	[A-Za-z]
digit	=	[0-9]
alphanumeric	=	{letter}|{digit}
other_id_char	=	[_]
memID	=	&{identifier}
HexLiteral	=	0 [xX] 0* {HexDigit} {1,8}
OctLiteral	=	0+ [1-3]? {OctDigit} {1,15}
integer	=	{digit}*
real	=	{integer}\.{integer}
char	=	'.'
identifier	=	({other_id_char}|{letter})({alphanumeric}|{other_id_char})*
nonrightbrace	=	[^}]
slash	=	\/
newline	=	\\n
comment_line	=	{slash}{slash}+
whitespace	=	[\n\t]
ILLEGAL	=	[^\n \t\f\x0b\rA-Za-z0-9+/\-*=<.~„:;()@\}\{\"_]

Keywords

Per quanto riguarda le keyword, sono state inserite tutte le keywords che rispettano lo standard ANSI 89:

auto break case char const continue default do double else enum extern
float for goto if int long register return short signed sizeof static
struct switch typedef union unsigned void volatile while

Stati

Gli stati usati nel file jflex sono:

- Lo stato COMMENT che gestisce i commenti su una linea.
- Lo stato LINE_COMMENT che gestisce i commenti su più linee e i commenti innestati.
- Lo stato STRING che gestisce le stringhe.

Tabelle

Per mantenere i valori o i nomi degli identificatori trovati dal Lexer sono state utilizzate tre tabelle. Queste tabelle sono state inserite con il solo scopo di mostrare il funzionamento di queste ultime nella fase di analisi lessicale. Nelle fasi successive, come ad esempio nell' analisi semantica, queste strutture si rivelano necessarie, però la loro complessità dovrebbe essere molto più elevata di quelle inserite da noi:

- symTable: la tabella in cui vengono messi i nomi degli identificatori
- stringTable: la tabella in cui viene messo il contenuto di una string costant
- numTable: la tabella in cui viene messo il valore di un qualsiasi numero trovato, sia esso esadecimale, reale, intero, etc.

Nel file ci sono tre contatori, uno per ogni tabella. Quando viene trovato un lessema da inserire nella tabella viene utilizzato il contatore come indirizzo della tabella. Ogni tabella è una Hashtable.

Es:

"Ciao Mondo!"

diventa

<STRING,1>

dove "1" nella stringTable è la chiave per avere la stringa "Ciao Mondo!".

Principali difficoltà incontrate

Le principali difficoltà incontrate durante la creazione del lexer sono state:

- Gestione delle stringhe. In generale in c una stringa non può contenere virgolette, in quanto le virgolette vengono utilizzate per racchiudere la stringa. Per includere il carattere di virgolette in una stringa, bisogna utilizzare una sequenza di escape (\ ). È possibile utilizzare una qualsiasi delle sequenze di escape che possono essere utilizzate come costanti di caratteri nelle stringhe Ecco alcuni esempi:

/ * Questa è una singola stringa costante. * /
"tutti frutti ice cream"

/ * Queste costanti stringa saranno concatenati, come sopra. * /
"tutti " "frutti" " ice " "cream"

/ * Questo utilizza due sequenze di escape. * /
"*"hello, world!*""

Se una stringa è troppo lunga per essere contenuto in una sola riga, è possibile utilizzare un backslash \ per disgregare la stringa su righe separate.

Es:

*"Today's special is a pastrami sandwich on rye bread with *
a potato knish and a cherry soda."

Due stringhe su due linee adiacenti verranno contatenate:
Es:

"Tomorrow's special is a corned beef sandwich on "
"pumpernickel bread with a kasha knish and seltzer water."

equivale a

*"Tomorrow's special is a corned beef sandwich on *
pumpernickel bread with a kasha knish and seltzer water."

Programma di Test

La classe di Test (CLexerTest.java) è formata da 3 metodi

- Il **main** non fa altro che inizializzare i file di input e output.
- Il metodo **initializeTokenNames** si occupa di generare una tabella contenente la coppia <ID, NomeToken>, cosicché l'output di test possa

essere letto da un umano senza dover controllare a mano la corrispondenza degli ID con i token nella classe sym.java.

- Il metodo **scan** controlla se il file passatogli in input esiste effettivamente e, dopo aver inizializzato la tabella dei token utilizzato il metodo initializeTokenNames, va a costruire un'istanza del CLexer che utilizza per scandire il file lessema per lessema e va a stampare sul file di input i vari token incontrati durante l'analisi.

Risultati test

Di seguito verranno riportati alcuni dei test fatti per provare che le regole del lexer da noi creato rispetta quelle del linguaggio C. I test riportati sono solo quelli con poche linee di codice in modo tale che ne sia più facile la comprensione.

Test 1

Uno dei test è stato effettuato su un semplice script nel quale non vi sono errori lessicali.

script

```
/*test c con caratteri di escape messi correttamente in  una stringa
*       compreso il carattere '\' per andare a capo con la stringa
*/
//commento su linea
int main(){
        int i = 0;
        int b = 0x12f;
        float num = 0.2;
/* ciao  /* questo è un commento innestato */  */
        char *p = "questa è \'davvero\' una \
 \"prova\" \n stringa? ";

}
```

output

```
comment_body
comment line
<INT_TYPE>  at line 4, column 0
```

```
<ID,0>  at line 4, column 4
<LPAREN>  at line 4, column 8
<RPAREN>  at line 4, column 9
<LBRACE>  at line 4, column 10
<INT_TYPE>  at line 5, column 1
<ID,1>  at line 5, column 5
<EQ>  at line 5, column 7
<INT_CONSTANT,0>  at line 5, column 9
<SEMICOLON>  at line 5, column 10
<INT_TYPE>  at line 6, column 1
<ID,2>  at line 6, column 5
<EQ>  at line 6, column 7
<HEX_CONSTANT,1>  at line 6, column 9
<SEMICOLON>  at line 6, column 14
<FLOAT_TYPE>  at line 7, column 1
<ID,3>  at line 7, column 7
<EQ>  at line 7, column 11
<REAL_CONSTANT,2>  at line 7, column 13
<SEMICOLON>  at line 7, column 16
comment_body
<MULT>  at line 8, column 47
<DIV>  at line 8, column 48
<CHAR_TYPE>  at line 9, column 1
<MULT>  at line 9, column 6
<ID,4>  at line 9, column 7
<EQ>  at line 9, column 9
buffer: questa è 'davvero' una  "prova"
 stringa?
<STRING,0>  at line 10, column 23
<SEMICOLON>  at line 10, column 24
<RBRACE>  at line 12, column 0
```

descrizione

Il test da noi sottoposto al lexer mostra che quest'ultimo è in grado di riconoscere i caratteri di escape in una stringa e salvarla correttamente nella tabella delle stringhe.
Nell' output la stampa del "buffer" mostra il contenuto della stringa formattata.
Nello script quindi:

```
"questa è \'davvero\' una \
 \"prova\" \n stringa? ";
```

è diventato

```
questa è 'davvero' una  "prova"
 stringa?
```

Test 2

Uno dei test è stato effettuato su un semplice script nel quale non è stata chiusa una stringa con le virgolette.

script

```
//test c con stringa non chiusa
int main(){
	int i = 0;
	int j;
	float num = 0.2;
/* ciao  /* questo è un commento innestato */   */
	char *p = "prova stringa;

}
```

output

```
comment line
<INT_TYPE>  at line 1, column 0
<ID,0>  at line 1, column 4
<LPAREN>  at line 1, column 8
<RPAREN>  at line 1, column 9
<LBRACE>  at line 1, column 10
<INT_TYPE>  at line 2, column 1
<ID,1>  at line 2, column 5
<EQ>  at line 2, column 7
<INT_CONSTANT,0>  at line 2, column 9
<SEMICOLON>  at line 2, column 10
<INT_TYPE>  at line 3, column 1
<ID,2>  at line 3, column 5
<SEMICOLON>  at line 3, column 6
<FLOAT_TYPE>  at line 4, column 1
<ID,3>  at line 4, column 7
<EQ>  at line 4, column 11
<REAL_CONSTANT,1>  at line 4, column 13
<SEMICOLON>  at line 4, column 16
comment_body
<MULT>  at line 5, column 48
<DIV>  at line 5, column 49
<CHAR_TYPE>  at line 6, column 1
<MULT>  at line 6, column 6
<ID,4>  at line 6, column 7
<EQ>  at line 6, column 9
<ERROR,Unterminated string constant>  at line 6, column 27
<ERROR,Unterminated string constant>  at line 7, column 2
<ERROR,Unterminated string constant>  at line 8, column 1
```

<ERROR,Unterminated string constant> at line 9, column 0
<ERROR,Unterminated string constant> at line 10, column 0

descrizione
Il test da noi sottoposto al lexer mostra che quest'ultimo è in grado di riconoscere
quando non viene chiusa una stringa in un programma.

<ERROR,EOF in string constant> at line 10, column 0

Test 3

Un altro test interessante è quello seguente, in cui è stato dato al lexer uno script con un carattere di escape non valido nella stringa e un commento non chiuso.

script

```
//test c con carattere escape non valido in una stringa
int main(){
        int i = 0;
        int j;
        float num = 0.2;
/* ciao  /* questo è un commento innestato */  */
        char *p = "prova \ stringa";
                /*
}
```

output
comment line
<INT_TYPE> at line 1, column 0
<ID,0> at line 1, column 4
<LPAREN> at line 1, column 8

```
<RPAREN>  at line 1, column 9
<LBRACE>  at line 1, column 10
<INT_TYPE>  at line 2, column 1
<ID,1>  at line 2, column 5
<EQ>  at line 2, column 7
<INT_CONSTANT,0>  at line 2, column 9
<SEMICOLON>  at line 2, column 10
<INT_TYPE>  at line 3, column 1
<ID,2>  at line 3, column 5
<SEMICOLON>  at line 3, column 6
<FLOAT_TYPE>  at line 4, column 1
<ID,3>  at line 4, column 7
<EQ>  at line 4, column 11
<REAL_CONSTANT,1>  at line 4, column 13
<SEMICOLON>  at line 4, column 16
comment_body
<MULT>  at line 5, column 47
<DIV>  at line 5, column 48
<CHAR_TYPE>  at line 6, column 1
<MULT>  at line 6, column 6
<ID,4>  at line 6, column 7
<EQ>  at line 6, column 9
<ERROR,Illegal escape sequence "\ ">  at line 6, column 18
buffer: prova stringa
<STRING,0>  at line 6, column 27
<SEMICOLON>  at line 6, column 28
comment_body
<ERROR,EOF in comment>  at line 10, column 0
```

descrizione

Il test da noi sottoposto al lexer mostra che quest'ultimo è in grado di riconoscere
il problema del carattere di escape non valido nella stringa:

```
<ERROR,Illegal escape sequence "\ ">  at line 6, column 18
```

e che il lexer è in grado di riconoscere e restituire un errore in caso un commento non sia stato chiuso.

```
<ERROR,EOF in comment>  at line 10, column 0
```

Test 4

Un test è stato effettuato sottoponendo al lexer una serie di raw-token, ossia di parole sulla quali il lexer avrebbe dovuto parsare e trovare un matching con le proprie regole.

script

```
/* This file is just a bunch of C tokens.  The lexer should recognize them
  in this form just fine. */
break
byte
case
catch
char
const
continue
do
double
else
*
+
-
/
;
,
(
)
[
]
=
<
>
<=
>=
!=
:
:=
.
a
b
```

```
c
abc
The_quick_brown_fox_jumped_over_the_lazy_dogs_0123456789
0
1
2
3
01
1234567890
1.0
1.23
1234567890.0123456789
000001.0000000
'a'
'b'
'c'
'!'
```

output

```
comment_body
<BREAK>  at line 2, column 0
<ID,0>  at line 3, column 0
<CASE>  at line 4, column 0
<ID,1>  at line 5, column 0
<CHAR_TYPE>  at line 6, column 0
<CONST>  at line 7, column 0
<CONTINUE>  at line 8, column 0
<DO>  at line 9, column 0
<DOUBLE_TYPE>  at line 10, column 0
<ELSE>  at line 11, column 0
<MULT>  at line 12, column 0
<PLUS>  at line 13, column 0
<MINUS>  at line 14, column 0
<DIV>  at line 15, column 0
<SEMICOLON>  at line 16, column 0
<COMMA>  at line 17, column 0
<LPAREN>  at line 18, column 0
<RPAREN>  at line 19, column 0
<LBRACK>  at line 20, column 0
<RBRACK>  at line 21, column 0
<EQ>  at line 22, column 0
<LT>  at line 23, column 0
<GT>  at line 24, column 0
```

```
<LTEQ>  at line 25, column 0
<GTEQ>  at line 26, column 0
<NOTEQ>  at line 27, column 0
<COLON>  at line 28, column 0
<COLON>  at line 29, column 0
<EQ>  at line 29, column 1
<DOT>  at line 30, column 0
<ID,2>  at line 31, column 0
<ID,3>  at line 32, column 0
<ID,4>  at line 33, column 0
<ID,5>  at line 34, column 0
<ID,6>  at line 35, column 0
<INT_CONSTANT,0>  at line 36, column 0
<INT_CONSTANT,1>  at line 37, column 0
<INT_CONSTANT,2>  at line 38, column 0
<INT_CONSTANT,3>  at line 39, column 0
<OCT_CONSTANT,4>  at line 40, column 0
<INT_CONSTANT,5>  at line 41, column 0
<REAL_CONSTANT,6>  at line 42, column 0
<REAL_CONSTANT,7>  at line 43, column 0
<REAL_CONSTANT,8>  at line 44, column 0
<REAL_CONSTANT,9>  at line 45, column 0
<CHAR_CONSTANT,10>  at line 46, column 0
<CHAR_CONSTANT,11>  at line 47, column 0
<CHAR_CONSTANT,12>  at line 48, column 0
<CHAR_CONSTANT,13>  at line 49, column 0
```

descrizione
Il lexer è riuscito a codificare come ID tutti gli ID presenti nel file ed ha fatto stesso con tutti gli altri lessemi dando così il risultato atteso.

Conclusioni

Seguendo le direttive per manuale di riferimento siamo riusciti a creare un lexer per il linguaggio C che corrisponde correttamente ad un lexer C completo per la specifica c89.

Il nostro lexer è in grado di analizzare un qualsiasi file di testo con all'interno codice C e riconoscere i lessemi contenuti all'interno generando un file contente tutti i token associati ai lessemi e generando errore nel caso in cui si incontri un carattere non appartenente alla specifica.
Per quanto riguarda alcune considerazioni riguardo al linguaggio C possiamo affermare che nonostante l'estrema libertà che lascia al programmatore il linguaggio in sé è abbastanza lineare da analizzare e non presenta meccaniche articolate come in altri linguaggi ad esempio di scripting come python.