LFC Lab (Linguaggi Formali e Compilatori) - Note del Corso

Edoardo Lenzi

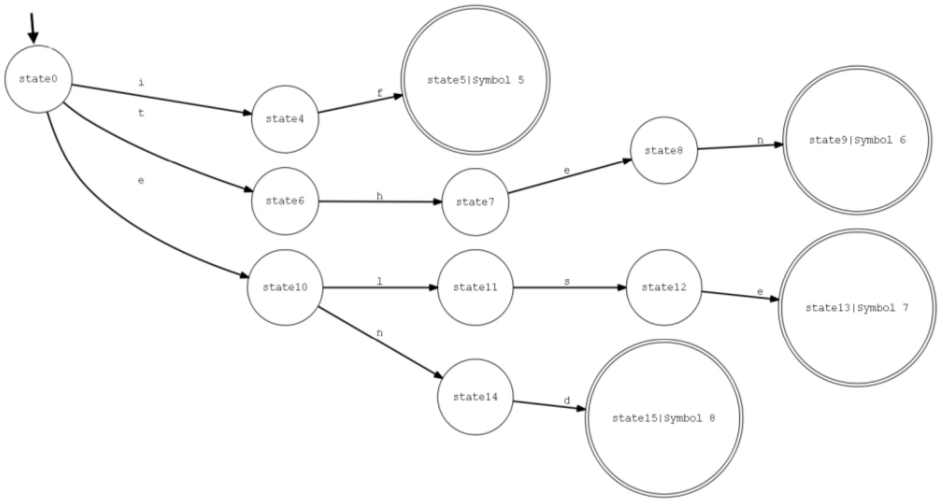
January 24, 2018

Contents

Chapter 1

Introduzione

Un Lexer λ é un DFA, ogni stato finale ϕ_i é associato ad una regular expression α_i. Una stringa $s \in L(\lambda)$, il pattern matching termina in uno stato finale ϕ_j ed s é associata ad una regular expression α_j; token: (s, α_j).



- (if, 5)
- (then, 6)
- (else, 7)
- (end, 8)

Data la descrizione del linguaggio che vuoi matchare lui ti genera il codice C per farlo.

1.1 Get started

Windows	https://sourceforge.net/projects/winflexbison/
Linux	sudo apt-get install bison flex gcc
Mac	XCode

Chapter 2

FLEX (Fast Lex)

Posso annotare una regular expression con istruzioni C.

2.1 Lex matching character

2.1.1 Operatori

$\alpha \cdot \beta$	concatenazione
$\alpha \vert \beta$	alternazione
$\alpha+$	1+ ripetizioni
$\alpha*$	0+ ripetizioni
$\alpha?$	0 o 1 ripetizione
.	un carattere qualsiasi (tranne \n)
$\alpha\{n, m\}$	$(n \leq m)$ matches α from n to m times
$[a-z]$	intervallo di caratteri da 'a ' a 'z '
$[0-9a-z\_]*$	(intervalli multipli) considerati 0+ volte [tutti gli id]
$a\$$	'a' a fine di una linea
$\hat{}a$	'a' all'inizio di una linea
$<< EOF >>$	
a/b	'a' iif 'b' segue
$[\hat{}C]$	complementare
$[\hat{}CB]$	$= [\hat{}C\hat{}B]$ matcha bat ma non Bat o Cat
"$stringa$"	
."at"	matches words: "cat", "rat", etc.

2.2 Template Lex

```
%{
  /* Dichiarazioni iniziali in codice C (librerie e variabili globali) */
%}

/* " Named " regular expressions here . */

%%
/* " Anonymous " Regular expressions here . */
%%

/* This section is copied verbatim into the lexer 's source . */
```

Nel primo blocco posso ficcare istruzioni C che verranno copiate verbatim (pari pari) nel sorgente C; ad esempio posso dichiarare variabili che poi uso nelle **semantic actions** delle regex.

Nel secondo blocco posso dare un nome ad una regex (al uso nella sezione dopo con l'operatore {}):

```
%{ /* Copied verbatim in lexer 's source . */ %}

Type ( " int " | " float " )
%%
{ Type }   { /* A Semantic action . */ }
```

```
[a - z ]+  { /* Another one . */ }
%%

int yywrap () {
   return 1;  //if I find eof should stop lexing?
}

int main ( int iArgC , char ** lpszArgV ) {
   yylex ();  // Starts lexing .
}
```

2.3 Compilare

```
> flex Input.l
> gcc lex.yy.c -o Lexer.out -std=c99
> Lexer.out < In.txt
```

2.4 Variabili generate da Lex

```
FILE * yyin ;   /* Default value is stdin . */
FILE * yyout ;  /* Default value is stdout . */
int yyleng ;    /* Number of characters read . */
char * yytext ;    /* The buffer on which characters are copied during pattern matching*/
```

2.4.1 Esempi Regex

```
%{
   #include <stdlib.h>
   int iWords = 0;
%}

word  [^ \t\r\n]
Declaration        "public"[ \t]+("static"[ \t]+)?"void"[ \t]+"main"

%%

{word}        {iWords+=1;}
{Declaration}  {printf("siamo nel main feghio")}
.|\n       {printf(" %d ", n++); /*mi stampa il numero di caratteri*/}
.|\n       {ECHO; /* mi stampa l'ultima parola per intero */}

"if"     |
"else"     |
"for"      {fprintf(stdout, "This is a keyword: %s\n", yytext);}

[\t \n\r]+    {/*Do nothing.*/;}
.|\n       {ECHO; fprintf(stdout, " is not a keyword.\n");}

<<EOF>>       {printf("words: %d \n", iWords); return yytext[10];}

%%

int yywrap() {
   return 1; //true per input multipli
}

int main() {
   printf("%d", yylex()); //10
}
```

2.4.2 Yywrap Function

When the scanner receives an end-of-file indication from YY_INPUT, it then checks the yywrap() function. If yywrap() returns false (zero), then it is assumed that the function has gone ahead and set up yyin to point to another input file, and scanning continues. If it returns true (non-zero), then the scanner terminates, returning 0 to its caller. Note that in either case, the start condition remains unchanged; it does not revert to INITIAL.

2.4.3 Input Multipli

```
%{
   #include <stdlib.h>
   int iCurrentFile = 0;
   int iFiles = 0;
   char**lpszFileName = NULL;
%}
%%
.|\n  {printf("ciao"):}
%%
int yywrap() {
   FILE*fInput = NULL;
   if (iCurrentFile<iFiles) {
      printf("Frase finale");
      //reset variables

      fclose(yyin);

      iCurrentFile+=1;

      fInput = fopen(lpszFileName[iCurrentFile], "r");
      if (fInput!=NULL) {
         yyin = fInput; //yyin il file da parsare
      }
   }
   return fInput!=NULL ? 0 : 1;
}

int main(int iArgC, char**lpszArgV) {
   if (iArgC>1){
      iFiles=iArgC;
      lpszFileName = &lpszArgV[1];

      FILE*fInput = fopen(lpszFileName[iCurrentFile], "r");
      if (fInput!=NULL) {
         yyin = fInput;
         yylex(); //start parsing yyin
      }
   }
}
```

2.5 Context Sensitivity (CS)

In base al token letto devo comportarmi in modo diverso. Ho delle **start conditions** (o start state), solo una é attiva. Il primo start state é l'**initial**. Uno start state puó essere **inclusivo** o **esclusivo**.

Da uno start state esclusivo solo regex correlate ad esso sono raggiungibili.

Da uno start state inclusivo tutte le regex correlate ad esso ed anche quelle non correlate agli altri start state sono raggiungibili.

```
%{ /* */ %}

% x String  // Exclusive start states
% s Cond   // Inclusive start states

%%
...
```

```
%%
```

2.6 Associare regole a stati

Per assegnare uno start state ad una re uso <**NOME SS**>. Per entrare in uno stato uso il comando **BEGIN**. All inizio il lexer é nello stato **INITIAL**.

```
%{ /* */ %}

% x Cond

%%
" \" "          { /* Read " char . */ BEGIN Cond ;}
< Cond >(^[ " ])+ { /* Consume string content . */ }
< Cond > " \" "    { /* Read " char . */ BEGIN INITIAL ;}
%%
```

2.6.1 Stati inclusivi ed esclusivi

```
%{ /* EXCLUSIVE START STATE*/ %}

%x String Unreacheable

%%
" \" "             { /* Read " char . */ BEGIN String ;}
< String > (^[ " ])+ { /* Consume string content . */ }
< String > " \" "  { /*Read " char . */ BEGIN INITIAL ;}
< Unreacheable > "end" { /* Will never be matched . */ }
%%

%{ /* INCLUSIVE START STATE */ %}

%s String

%%
" \" "          { /* Read " char . */ BEGIN String ;}
< String >(^[ " ])+ { /* Consume string content . */ }
< String > " \" "  { /* Read " char . */ BEGIN INITIAL ;}
" end "         { /* Will be matched . */ }
%%
```

Gli start state sono realizzati tramite uno stack, ho tre funzioni per manipolarli:

- void yy push state(int NewState) //pusho in cima allo stack, equivalente a BEGIN NewState;
- void yy pop state() //fa una pop, equivalente a BEGIN;
- int yy top state() //fa la top, non esistono metacomandi per farla;

2.6.2 Esempio

```
%{
    #include <stdlib.h>

    int iLine = 0;
    int iColumn = 0;
%}

%s FirstDate
%x SecondDate Motivation Amount Info
```

```
%%

"#".+   {iColumn+=yyleng;}

"$"     {iColumn+=1;
        fprintf(yyout, "INSERT INTO Movement(SDate, EDate, Desc, Amount, Info) VALUES (");
        BEGIN FirstDate;}

([0-9]{2}"/"?){3}          {iColumn+=yyleng;
                           fprintf(yyout, "to_date('%s', 'DD/MM/YY'), ", yytext);
                           BEGIN SecondDate;}

<SecondDate>([0-9]{2}"/"?){3} {iColumn+=yyleng;
                           fprintf(yyout, "to_date('%s', 'DD/MM/YY'), ", yytext);
                           BEGIN Motivation;}

<Motivation>[^-0-9]+            {iColumn+=yyleng;
                                fprintf(yyout, "'%s', ", yytext); BEGIN Amount;}

<Amount>"-"?[0-9\.]+","[0-9]{2}[ \t]+"EUR"   {int i=0, j=0;
                                               for (;i<yyleng; i+=1, j+=1) {
                                                   if (yytext[i]=='.') {
                                                       i+=1;
                                                   }
                                                   if (i!=j) {
                                                       yytext[j]=yytext[i];
                                                   }
                                               }
                                               yytext[j]='\0';
                                               for (int i=0; i<yyleng; i+=1) {
                                                   if (yytext[i]==','){yytext[i]='.';}
                                                   if (yytext[i]==' '){
                                                       yytext[i]='\0'; break;
                                                   }
                                               }
                                               fprintf(yyout, "%s, ", yytext);
                                               BEGIN Info;}

<Info>[^$]*                     {for (int i=0; i<yyleng; i+=1) {
                                       if (yytext[i]=='\n'||yytext[i]=='\r') {
                                           yytext[i]=' ';
                                       }
                                   }
                                   fprintf(yyout, "'%s');\n", yytext);
                                   BEGIN INITIAL;}

[ \t]+                          {iColumn+=yyleng;}
[\r\n|\r|\n]                    {iColumn=0; iLine+=1;}
.                               {printf("ERROR in line: %d, column: %d.", iLine, iColumn);
                           yyterminate();}
%%

int yywrap(){return 1;}

int main(int iArgC, char**lpszArgV) {
    FILE*fInput = fopen(lpszArgV[1], "r");
    FILE*fOutput = fopen(lpszArgV[2], "w");
    if (fInput!=NULL && fOutput!=NULL) {
        yyin = fInput;
        yyout = fOutput;

        fprintf(yyout, "DROP TABLE IF EXISTS Movement;");
      fprintf(yyout, "CREATE TABLE Movement(\n\tStartDate DATE,\n\tEndingDate DATE, ...;\n");
        yylex();
    }
}
```

2.7 Ambiguous Specifications

Dati α e β regular expressions taliché $L(\alpha) \subset L(\beta)$, devo stare attento all'ordine in cui le scrivo, quelle piú in alto hanno precedenza su quelle piú in basso.

> Se con molte regular expression posso arrivare in un nodo, vince quella a prioritá maggiore. Regular expressions generating constant finite languages must be placed first, or they will be obscured.

Una volta che una stringa é matchata viene **consumata**, le altre re non la potranno piú matchare. Posso anche usare **REJECT** per darla in pasto alla seconda re in ordine di prioritá.

```
%{ int iCounter = 0; %}

%%
" abcde "  { iCounter +=1; REJECT ;}
" abcd " { iCounter +=1; REJECT ;}
" abc "    { iCounter +=1; REJECT ;}
" ab "     { iCounter +=1; REJECT ;}
"a"        { iCounter +=1; }
.       { /* Consumes the rest . */ }
%%

int main (){
   yylex ();
   /* iCounter = 5 when input is " abcde ". */
}
```

Chapter 3

Exercises

3.1 Lex

Devise a lexer accepting a Windows file path.
Devise a lexer accepting a Linux file path.
Devise a lexer accepting a non regular language.

3.2 Yacc

Design an unambiguous grammar for arithmetic expressions whose operations are: difference, sum, product, division, exponentiation.
Remember to encode precedence and associativity: Difference and sum are left associative.
Product, division and exponentiation are right associative.

Chapter 4

Acronimi

w	Word (2 byte short integer)
dw	DWord (4 byte integer)
p	untyped pointer
lp	generic long pointer
lpsz	long pointer to a null-terminated string

Chapter 5

Yacc (Yet Another Compiler Compiler)

É un PDA (Personal Digital Assistant) basato sull'**automa LALR(1)**; runna l'algoritmo shift/reduce per decidere se una stringa di simboli appartiene o meno a L(G). Yacc é stato sostituito da Bison; é un **parser generator** per grammatiche libere. Posso associare una semantic action ad ogni statement.

Quando ho un match con il body di una produzione applico la riduzione ed eseguo la semantica action associata.

Flex e Yacc sono stati pensati per lavorare assime anche se nulla vieta di usarli separatamente.

5.1 Lexer/Parser communication

L'algoritmo shift reduce coinvolge il lexer per capire quale terminale sto leggendo. Si crea un rapporto master slave fra lexer e parser.

```
int iToken = Lexer.GetNextToken();
```

5.2 Cosa puó fare Yacc

Data una grammatica G mi dice se é o meno LALR(1). Date delle grammatiche non LALR(1) mi da dei tool per parsarle. Copre anche grammatiche IELR(1) e GLR.

5.3 Yacc Grammar Rules

Il template yacc é uguale a quello di flex con dei comandi aggiuntivi:

```
//Lexer .l
%{
    #include "y.tab.h"
    int yylex();
    void yyerror(char*s);
%}

%option noyywrap

%%

"a"    {printf("\n Lex %c \n", yytext[0]); return yytext[0];}
.    {printf("Error\n");}

%%

void yyerror(char*s) {
    printf(s);
}

//Parser .y
%{
```

```
    #include <stdio.h>
    FILE*yyin;
%}

%start S

%%

S: A    { printf("Parsed S.\n");}
| D    { printf("Parsed S.\n");} ;

%%

int main() {
    do {
        yyparse();
    } while(!feof(yyin));
    return 0;
}
```

Una produzione $A \rightarrow B$ viene specificata come $A : B$. I non terminali vanno fra apici ($E \rightarrow (E)+T$ diventa E: '('E') ' '+' T)

5.4 Compilare

```
flex input.l
bison -d input.y -o y.tab.c
gcc lex.yy.c y.tab.c -o Lexer.out -std=c99
./Parser.out < input.txt
```

5.5 Yacc directives

5.5.1 %start

Posso dichiarare lo start symbol con la direttiva %start, se non lo setto viene automaticamente settato il primo simbolo.

5.5.2 %token

```
//Lexer
"(a)"  {return A_TOKEN;}
//Parser
%token TerminalName_1 TerminaleName_2 ... TerminalName_n
%%
A: A_TOKEN B_TOKEN C { printf("Parsed A.\n");} ;
```

In pratica nel .l ritorno un enum che é dichiarato nell'interstazione del .y con l'istruzione %token.

5.5.3 %union

Declare a union of types and identifiers.

```
//Parser
%{
    #include <stdio.h>
    FILE*yyin;
%}

%token LOWER_WORD_TOKEN UPPER_WORD_TOKEN NUMBERS_TOKEN

%union {
    char*lpszLowerWord;
    char*lpszUpperWord;
```

```
    char*lpszNumbers;
}

%type <lpszLowerWord> LOWER_WORD_TOKEN
%type <lpszUpperWord> UPPER_WORD_TOKEN
%type <lpszNumbers> NUMBERS_TOKEN

%%

//$2 e' il secondo simbolo nel body della produzione
S: S LOWER_WORD_TOKEN { printf("Lower case word: %s.\n", $2); }
| S UPPER_WORD_TOKEN { printf("Upper case word: %s.\n", $2); }
| S NUMBERS_TOKEN    { printf("Numbers: %s.\n", $2); };
|
%%

int main() {
    do {
        yyparse();
    } while(!feof(yyin));
    return 0;
}

//Lexer
[a-z]+ {yylval.lpszLowerWord = StringDuplicate(yytext, yyleng); return LOWER_WORD_TOKEN;}
[A-Z]+ {yylval.lpszUpperWord = StringDuplicate(yytext, yyleng);return UPPER_WORD_TOKEN;}
[0-9]+ {yylval.lpszNumbers = StringDuplicate(yytext, yyleng); return NUMBERS_TOKEN;}
```

In pratica con la union definisco variabili accessibili dal lexer tramite l'oggetto yylval. Con %type $< Identifier >$ $Symbol_1$ ... $Symbol_n$ attribuisco un tipo ai nodi (stati) nelle semantic actions.

Quando Yacc incontra la direttiva %token definisce in y.tab.h un set di interi per ogni terminale associato.

5.6 C/C++ Union

```
union { /*list of valid C identifiers */ }
```

Associa alla stessa area di memoria molte variabili; pertanto la grandezza della union é la grandezza della variabile con max sizeOf(). Qualsiasi variabile puó essere referenziata ed editata. Il risultato corretto dipende dall'interpretazione dei bytes.

```
struct Test {
    union {
        int iFirst ;
        char cSecond ;
        char cThird [2];
        void * pPointer ;
    };
};

struct Test * a = ... /* Allocate empty Test . */ ;
a - > pPointer = 0 x000000DD ;
a - > iFirst += 0 xFF00CC00 ;
a - > cSecond = 0 xAA ;
a - > cThird [1] = 0 xBB ;
// a = 0xAABBCCDD!
```

Con la direttiva union creo un ponte fra lexer e parser

5.7 Indexing tree's data structure

Vedo le produzioni come un albero, $$ rappresenta il padre, $1 rappresenta il primo simbolo del body, $2 il secondo e cosí via.

5.8 Operazioni aritmetiche

```
// Lexer
%}
Decimal      [0-9]+
Hexadecimal     "0x"[0-9A-Fa-f]+
Binary       "b"(0|1)+
Real      ([0-9]+)?"."([0-9]+)
%%

"+"  | "-" | "*" | "/" | "(" | ")" | "^" { return yytext[0];}

{Decimal}      {
                      int iIntegerValue = 0;
                      for (int i=0; i<yyleng; i+=1) {
                          iIntegerValue*=10; iIntegerValue+=yytext[i]-'0';
                      }
                      yylval.iInteger = iIntegerValue;
                      return INTEGER;
                  }
{Hexadecimal}  {
                      int iHexadecimalValue = 0;
                      for (int i=0; i<yyleng; i+=1) {
                          if ((yytext[i]>='A' && yytext[i]<='F') || (yytext[i]>='a' &&
                              yytext[i]<='f')) {
                              iHexadecimalValue*=10; iHexadecimalValue+= 10 +
                                  yytext[i]-'A';
                          } else {
                              iHexadecimalValue*=10; iHexadecimalValue+=yytext[i]-'0';
                          }
                      }
                      yylval.iInteger = iHexadecimalValue;
                      return INTEGER;
                  }
{Binary}       {
                      int iBinaryValue = 0;
                      for (int i=0; i<yyleng; i+=1) {
                          iBinaryValue*=2; iBinaryValue+=yytext[i]-'0';
                      }
                      yylval.iInteger = iBinaryValue;
                      return INTEGER;
                  }

// Parser
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

int yylex();
int yyparse();
FILE *yyin;
void yyerror(char *s);

%}

%union{
    int iInteger;
}

%token INTEGER

%type  <iInteger>  Expression INTEGER

%left '+' '-'
%left '*' '/'
```

```
%right '^'
%nonassoc SpecialSymbol

%%
S: Expression           {printf("%d", $1);}

Expression:  Expression '+'     Expression {$$= $1+$3;}
    |      Expression '-'     Expression {$$= $1-$3;}
    |      Expression '/'     Expression {$$= $1/$3;}
    |      Expression '*'     Expression {$$= $1*$3;}
/*  |      Expression '^'  Expression {$$= pow($1, $3);} */
    |      '-'   Expression           {$$= -$2;}      %prec SpecialSymbol
    |      '('   Expression ')'       {$$= $2;}
    |      INTEGER                 {$$= $1;}
    ;

%%
```