GENERATORI AUTOMATICI DI SCANNER: FLEX

GENERATORI DI SCANNER

Un generatore automatico di scanner prende in input un file che specifica il lessico di un certo linguaggio, solitamente nella forma di espressioni regolari, includendo altre funzioni ausiliarie, definizioni di token, etc., e produce in output un codice (scritto in un certo linguaggio di programmazione) che implementa lo scanner. Ci risparmia quindi la fatica di costruire a mano l'automa a partire dalle espressioni regolari.

Esitono molti generatori automatici:

- **lex, flex, scangen,** generano un codice in ***C***
- **jlex, sable, cup** generano un codice in **java**

ESEMPI DI ANALIZZATORI LESSICALI

Lex, il primo generatore di scanner, inventato da Mike Lesk e Eric Shmidt (AT&T Bell Lab) nel 1975.

Uno dei generatori di scanner più conosciuti ed attualmente usati è **flex** – fast lexical analyser generator (introdotta da Vern Paxson intorno al 1987 per risolvere problemi di efficienza del lex).

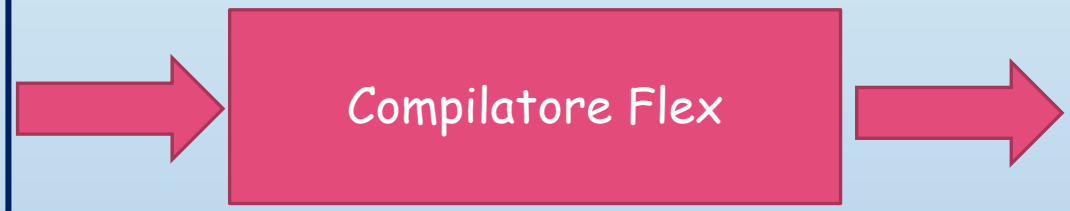
Flex è un free software. Esso è frequentemente usato con Bison, un parser generator alternativo a Yacc, ma è comunque utilizzabile come generatore di programmi «stand alone» per fare un'analisi lessicale del testo a prescindere dal parsing. Pur non essendo un software GNU, il GNU project ne distribuisce un manuale.

- http://flex.Sourceforge.Net/ (linux)
- http://gnuwin32.Sourceforge.Net/packages/flex.htm (windows)

USO DI FLEX

Primo step

INPUT: File in **formato flex** contenente la descrizione dei pattern dei lessemi mediante **espressioni regolari** e **azioni** da effettuare ad ogni match con una RE. Può contenere anche altre funzioni ausiliarie, definizioni di token…



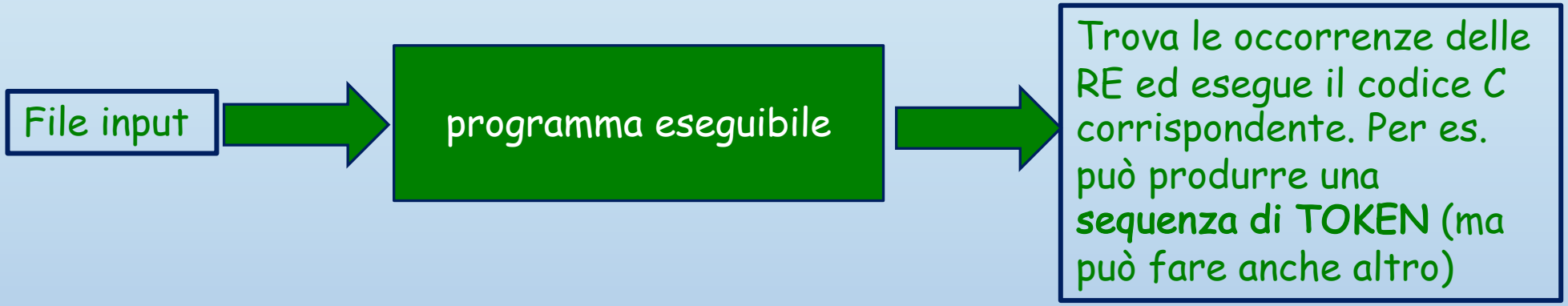
OUTPUT: file in c **lex.yy.c** contenente codice in C. Viene definita la funzione **yylex()** che è un'implementazione basata su tabella del DFA che riconosce il linguaggio definito dalle RE

USO DI FLEX

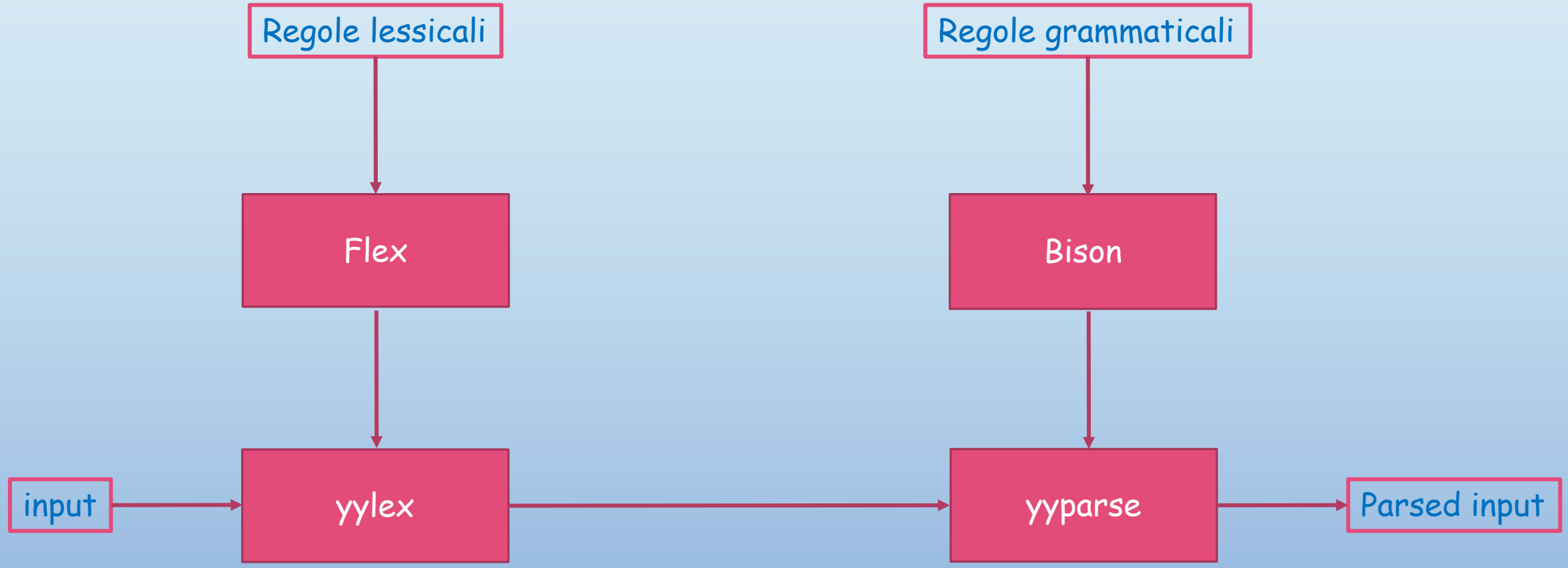
Secondo step

USO DI FLEX

Terzo step



USO DI FLEX INSIEME A BISON



FLEX-FILE.FL

nomefile.fl: file sorgente in formato flex, che serve per avere una descrizione dello scanner da generare. La descrizione è nella forma di coppie comprendenti:

- **espressione regolare**: definiscono rigorosamente gli elementi che devono essere riconosciuti nel flusso di dati.
- **Codice C**: azioni da effettuare al verificarsi del match con una RE.

Le coppie, RE e codice C, formano la sezione REGOLE DI TRADUZIONE del file nomefile.fl

LEX.YY.C

E' l'output del programma flex.

lex.yy.c è un file di codice sorgente C, che definisce la funzione **'yylex()'**. Questa funzione può essere utilizzata direttamente nella funzione main, se lo scanner è utilizzato da solo, o, in associazione con un parser, per il recupero dei token e non ritorna al chiamante fino a quando non ha esaurito i suoi dati in lettura.

YYLEX()

La funzione, yylex(), scandisce l'input file yyin e ritorna il prossimo token, ricopiando sul file yyout il testo non riconosciuto.

Per default, i file yyin e yyout sono inizializzati rispettivamente a stdin e stdout. Il token riconosciuto (una costante, un numero, un'istruzione, ecc…), sarà fornito, eventualmente con l'indicazione del tipo, al parser (bison), ogni volta che questo la richiamerà.

Il parser in base alle informazioni ricevute, applicherà le regole opportune. L'implementazione di questa funzione si basa su un DFA che rappresenta le RE. Al termine di ogni azione l'automa si ricolloca sullo stato iniziale, pronto a riconoscere nuovi simboli.

PROGRAMMA ESEGUIBILE

Compilando il file lex.yy.c tramite un compilatore C, si ottiene il programma eseguibile.

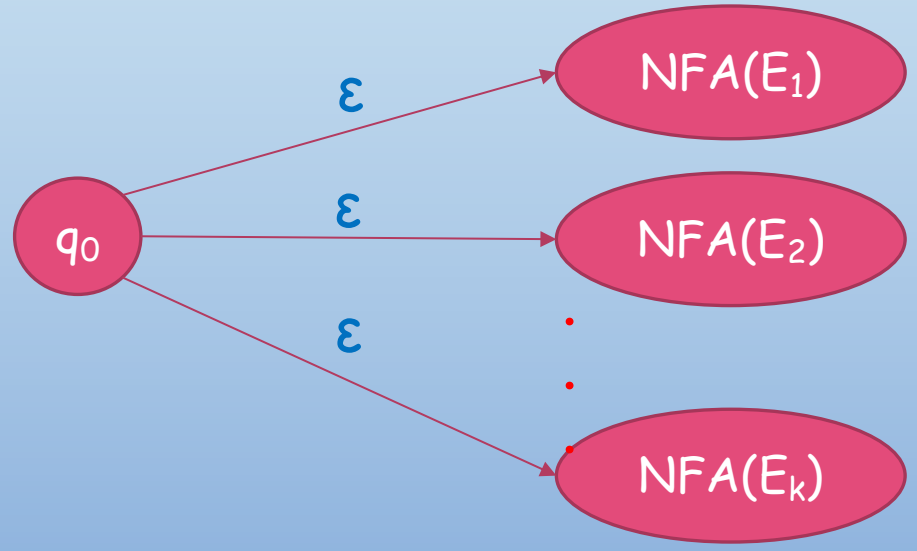
Quando l'eseguibile viene lanciato, analizza il proprio ingresso in cerca di occorrenze delle espressioni regolari. Ogni volta che ne individua una, viene eseguito il corrispondente codice C.

L'input viene dato sotto forma di file testo, e contiene il testo che deve essere analizzato dall'analizzatore lessicale

COME FUNZIONA FLEX?

PRIMO PASSO

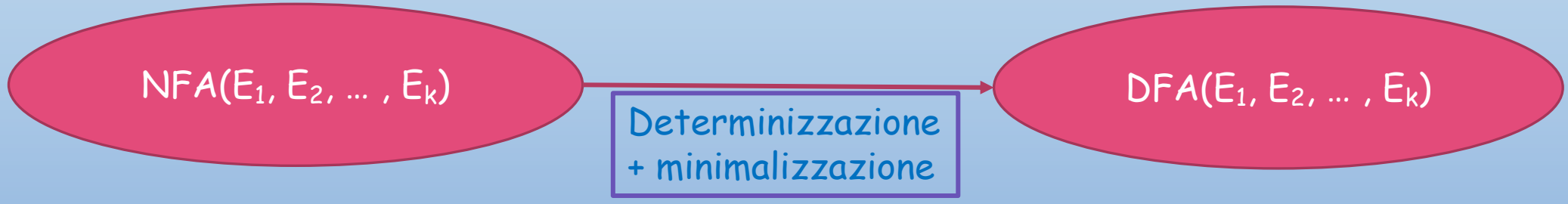
- Si dà in input un file in cui si descrivono tutte le espressioni regolari che definiscono i token
- Flex converte le espressioni regolari in automi non deterministici
- Combina tutti i NFA's in un unico NFA



COME FUNZIONA FLEX

SECONDO PASSO

- Flex converte l'NFA in un automa deterministico (DFA).
- Ottimizza l'automa deterministico
- Produce codice per simulare l'azione del dfa



FORMATO DI UN PROGRAMMA IN FLEX

Un file sorgente in formato flex consiste di 3 sezioni, separate da %%:

Dichiarazioni (opzionale)

%% → Obbligatorio, anche se la sezione precedente è vuota

Regole di traduzione (obbligatoria)

%% → Possiamo ometterlo se la sezione seguente è vuota

Funzioni ausiliarie (opzionale)

SCRIVERE UN PROGRAMMA IN FLEX

- **DEFINIZIONI** (opzionale) contiene:
  - Definizioni di variabili, costanti simboliche o definizioni regolari
  - Segmento di codice C, indentato oppure delimitato da %{ e %}, che deve comparire nel file di output
- **REGOLE** che contiene una sequenza di regole contenenti:
  - I pattern espressi mediante RE
  - Codice C da eseguire in corrispondenza del match con un certo pattern
- **FUNZIONI AUSILIARIE** (opzionale) che contiene codice C che deve essere copiato nel file di output

Anche nella sezione REGOLE è possibile inserire codice tra %{ e %}. Ciò deve avvenire prima della prima regola e mai dove dovrebbe essere definito un nuovo pattern. Può servire per esempio per dichiarare variabili locali usate nella routine di scanning.

Possono anche essere introdotti commenti tra /* e */, che appariranno nel file di output

FLEX : SEZIONE DICHIARAZIONI - DEFINIZIONI REGOLARI

Questa sezione contiene

DEFINIZIONI REGOLARI

ogni definizione è del tipo:

nome definizione

Esempi di definizione regolare:

```
DIGIT          [0-9]
LETTER         [a-zA-Z]
ID
        {LETTER}({LETTER}|{DIGIT})*
```

il nome sarà lo stesso che, nella sezione regole di traduzione, potrà essere utilizzato per riferirsi alla specifica definizione, attraverso la dicitura {nome}

Hanno lo scopo di semplificare la scrittura delle espressioni regolari nella sezione delle regole

FLEX : SEZIONE DICHIARAZIONI - OPZIONI

Flex permette il passaggio di opzioni che altrimenti si dovrebbero passare attraverso la riga di comando. Le opzioni vengono passate attraverso la direttiva: %option nomeopzione

esempi di opzioni:

%option main: flex fornisce una routine main() di default

%option noyywrap: questa direttiva comporta che non venga chiamata la funzione yywrap() dopo che è stato raggiunto un carattere di end-of-file (fine file) e si assume, quindi, che non esistano più file da analizzare. La funzione yywrap() dovrebbe impostare correttamente la variabile globale yyin e ritornare il valore zero se esistono altri file da analizzare, oppure ritornare un valore non zero nel caso in cui il lavoro da compiere sia giunto al termine (e, quindi, ottenere anche la terminazione nell'esecuzione dello scanner).

FLEX : SEZIONE DICHIARAZIONI - SEGMENTO DI CODICE C

Il codice C, delimitato da

%{ %}

Sarà ricopiato parola per parola nel file di output lex.yy.c

Esempio di segmento C:

```
%{
#include <stdio.h>
#define PAP 1
#define PCH 2
#define SUM 3
%}
```

In questo segmento di programma si possono per esempio includere librerie e definire costanti.

FLEX : SEZIONE REGOLE DI TRADUZIONE REGOLE E AZIONI

Questa sezione è l'unica obbligatoria ed è la più importante del file flex, perché è qui che vengono specificate:

- Le regole di matching (riconoscimento dei pattern)
- Le azioni da intraprendere per ogni pattern riconosciuto

Anche in questa sezione è possibile inserire codice tra %{ e %}. Ciò deve avvenire prima della prima regola. Può servire per esempio per dichiarare variabili locali usate nella routine di scanning.

FLEX : SEZIONE REGOLE DI TRADUZIONE

La sezione regole di traduzione avrà la seguente struttura:

Pattern **{action}**

Pattern sono espressioni regolari che permettono di avere un riscontro con determinate stringhe di caratteri del linguaggio sorgente. Questi pattern possono essere resi più compatti richiamando in essi i nomi associati alle diverse definizioni come descritto nella sezione dichiarazioni.

Action la parte action può:

- essere vuota (contiene solo ;), al match del pattern non succede nulla.
- Contenere un segmento di codice C, racchiuso tra {} , associato ad uno specifico pattern e solo in presenza di esso sarà eseguito integralmente.
- Contenere solo | indica che quell'azione è definita nella stesso modo della regola che segue.

FLEX : SEZIONE REGOLE DI TRADUZIONE

Esempi di regole di traduzione:

```
{ID}          { printf("identificatore\n"); }
[a-z]+        {ECHO;}
\t            |
\n            ;
```

Direttive speciali che possono essere incluse in un'azione sono:

ECHO: copia yytext (ossia il lessema riconosciuto) nell'output dello scanner

REJECT: cerca un'alternativa al match corrente

FLEX : SEZIONE FUNZIONI AUSILIARIE

Quest'ultima sezione contiene semplicemente codice C che sarà copiato esattamente come è scritto nel file di output lex.yy.c

Questa sezione è opzionale e viene spesso usata per contenere funzioni richiamate dallo scanner come supporto all'elaborazione.

Di fatto, qua possono essere riportate tutte quelle funzioni utili che sono richiamate all'interno da qualche azione nella sezione delle regole di traduzione.

Se si vuole generare uno scanner che possa vivere di vita propria, bisogna fornirlo di una funzione main, tale funzione va inserita proprio in questa sezione.

FLEX : OSSERVAZIONI

- Sia nella sezione delle dichiarazioni che in quella delle regole di traduzione tutto ciò che si trova all'interno della coppia di identificatori %{ e %} viene copiato direttamente nel file di output lex.yy.c senza alcuna modifica.

  %{ e %} devono apparire a inizio di linea (non indentati).

- Nella sezione regole di traduzione è possibile inserire codice tra %{ e %}, basta che ciò avvenga avvenga prima della prima regola.

- In flex qualsiasi cosa tra /* e */ è considerato commento e copiato direttamente nel file di output lex.yy.c però ci sono due eccezioni:

  1. Nella sezione dichiarazioni i commenti non possono apparire sulla linea in cui si trova la direttiva %option

  2. Nella sezione regole di traduzione i commenti non possono apparire all'inizio di una linea o immediatamente dopo una lista di stati dello scanner.

PROGRAMMA IN FLEX

N.B. Sia nella sezione delle definizioni che in quella delle regole:

- Ogni testo indentato o racchiuso tra %{ e %} viene copiato parola per parola sull'output.

- **%{** e **%}** devono apparire a inizio di linea (non indentati).

- Nella sezione delle definizioni, un **commento non-indentato** che comincia con **/*** e finisce con ***/** diventa un **commento nel file di output**

Esiste la regola di default: **il testo dell'input che non combacia con nulla viene ricopiato sull'output.**

IL PROGRAMMA FINALE GENERATO DA FLEX:

- Legge il suo input un carattere alla volta da sinistra a destra finché trova il più lungo prefisso di input (quello sarà il corrente lessema) che fa match con uno dei pattern della sezione delle regole di traduzione.
- Il testo corrispondente al match viene reso disponibile attraverso la variabile globale yytext e la sua lunghezza viene memorizzata in yyleng.
- Vengono quindi eseguite le azioni corrispondenti al pattern per il quale avviene il match.
- Prosegue la scansione dell'input alla ricerca di altri match.

ESEMPIO

Il programma più semplice:

Primo.fl: 

Genera uno scanner che semplicemente copia il suo input (un carattere per volta) nel suo output.

Come si usa:

1. `flex primo.fl`

Costruisce il codice C per la costruzione dell'analizzatore lessicale a partire dalle espressioni regolari. Genera il file lex.yy.c

2. `gcc lex.yy.c -o primo`

Compila in C il codice generato da flex. Genera il file primo.exe (o a.exe se il nome dell'output non è specificato)

3. `primo < prova.txt >output.txt`

Applica il programma eseguibile generato al passo precedente al file prova.txt e fa scrivere l'output nel file output.txt (o lo stampa a video se l'output non è specificato)

COME DEVE ESSERE ESPRESSO IL FILE DI INPUT?

Mostriamo come deve essere scritto il file primo.Fl

```
%option main
%%
```

/* E' un opzione che serve nel caso si voglia utilizzare la funzione main generata di default */

In alternativa nella sezione del codice va inserito un segmento di codice del tipo:

```
int main (int argc, char** argv)
{
    --argc; ++argv;
  if(argc > 0)
    yyin = fopen(argv[0], "r");
  else yyin = stdin;
  yylex();
  return 0;
}
```

OPZIONI

%option può essere seguito da

main: flex fornisce una routine main() di default

noyywrap: lo scanner non chiamerà la routine yywrap che serve per la gestione di più file di input

COME DEFINIRE I PATTERN NELLA SEZIONE DELLE REGOLE?

x match con il carattere'x'

. qualsiasi carattere escluso il newline

[xyz] una "classe di caratteri"; in questo caso, il pattern ha un match o con 'x', 'y', o 'z'

[abj-oz] una "classe di caratteri" contenente un range; ha un match con 'a', 'b', una qualsiasi lettera da 'j' a 'o', oppure 'z'

[^A-Z] una "classe negata di caratteri", i.e., qualsiasi carattere escluso quelli della classe. In questo caso, un qualsiasi carattere escluso le lettere maiuscole.

[^A-Z\n] qualsiasi carattere tranne le maiuscole e il newline

- Note: dentro le parentesi quadre solo \, - e ^ sono caratteri speciali; per includere il carattere – questo deve apparire come primo o ultimo carattere.

COME DEFINIRE I PATTERN NELLA SEZIONE DELLE REGOLE?

r*	zero o più espressioni r
r+	una o più espressioni r
r?	zero o una r
r{2,5}	da due a cinque r
r{2,}	due o più r
r{4}	esattamente 4 r
{name}	espansione della definizione “name”
"[xyz]\"foo"	la stringa: [xyz]"foo

- Note: il carattere \ ha il ruolo di carattere di escape.

COME DEFINIRE I PATTERN NELLA SEZIONE DELLE REGOLE?

\x se x è 'a', 'b', 'f', 'n', 'r', 't', o 'v', allora \x ha la stessa interpretazione che per ANSI-C. Altrimenti, sta per il carattere x. (Usato per l'escape di operatori come '*')

\0 carattere NULL (ASCII code 0)

\123 carattere con valore ottale 123

\x2a carattere con valore esadecimale 2a

(r) espressione r; le parentesi servono per imporre delle precedenze

rs espressione r seguita da s (concatenazione)

r|s r oppure s (unione)

r/s una r ma solo se seguita da s. Il testo che è ricoperto da s è considerato incluso per effetto del "longest match", ma è restituito all'input prima che venga eseguita l'azione. Questo tipo di pattern è chiamato "trailing context".

- Nota: valgono le regole di precedenza per le espressioni regolari; ad es. Ciao|bu* è equivalente a (ciao)|(b(u)*)

COME DEFINIRE I PATTERN NELLA SEZIONE DELLE REGOLE?

^r r, ma solo all'inizio di una linea (prefisso)

r$ r, ma solo alla fine di una linea. Equivalente a "r/\n". (suffisso)

<s>r r, ma solo nella start condition s

<s1,s2,s3>r r stessa cosa, ma in una qualsiasi start condition s1, s2, o s3

<*>r r in una qualsiasi start condition, anche in una esclusiva.

<<Eof>> end-of-file

<s1,s2><<eof>> end-of-file in start condition s1 o s2

[:digit:] indica tutti i caratteri per cui "isdigit" restituisce true

COME AVVIENE IL MATCH?

Quando viene determinato un match, il testo corrispondente al match viene inserito nella variabile globale **yytext** e la sua lunghezza viene memorizzata in **yyleng**.

Vengono quindi eseguite le azioni corrispondenti al pattern per il quale avviene il match.

Prosegue la scansione dell'input alla ricerca di altri match.

Se non viene trovato alcun match, viene eseguita la default rule: il carattere dell'input viene considerato un match e copiato nell'output

Nota: la variabile **yytext** può essere definita come char pointer o array; E' possibile controllare tale definizione attraverso la direttiva %pointer o %array nella sezione delle definizioni. Se si usa l'opzione -l (compatibilità con lex), yytext è un array. %pointer consente scansioni più veloci

COME DEFINIRE LE AZIONI?

Pattern azione

Se l'azione è vuota (contiene solo ;), al match del pattern non succede nulla.

Se l'azione contiene un parentesi graffa aperta {, allora verranno eseguite le azioni fino al raggiungimento di }.

Un'azione che consiste solo di | indica che è definita nella stesso modo della regola che segue.

Esempio 1:

```
[ ]  |
\t   |
\n   ;
```

Equivale ad avere ovunque ;

ESEMPIO 2:

```
[a-z]+ {printf("%s",yytext);}
```

È equivalente a

```
[a-z]+ {echo;}
```

NOTA: echo è una direttiva speciale che copia yytext nell'output dello scanner

ESEMPIO

```
/* nella sezione definitions i commenti (come questo)
vengono copiati nel file di output*/
%{
int num_lines = 0, num_chars = 0;
%}
%option noyywrap
%%
\n {++num_lines; ++num_chars;}
.  {++num_chars;}
%%
Int main(int argc, char *argv[])
{
--argc; /*salta il nome del programma*/
if ( argc > 0 )
        yyin = fopen(argv[1], "r" );
else
        yyin = stdin;
yylex();
printf( "# di linee = %d, # di caratteri=%d\n",
num_lines, num_chars );
return 0;
}
```

Commenti tra /* e */

Costruire uno scanner che conta il numero di caratteri e il numero di linee dell'input:

Definizione e inizializzazione delle variabili num_lines e num_chars

Se viene incontrata una newline, si incrementano sia il numero delle linee che il numero dei caratteri

Se si incontra un qualunque carattere si incrementa solo il numero dei caratteri

Il numero di linee e il numero di caratteri sono stampati al video

VARIABILI E ROUTINE DISPONIBILI ALL'UTENTE

yylex()	routine di scanning
yytext	stringa con la quale avviene il match
yyin	input file (default: stdin)
yyout	output file (default: stdout)
yyleng	lunghezza di yytext
input()	legge il carattere successivo

...

ESEMPIO

```
%{
int conta_3=0, conta_5=0, num;
%}
%option noyywrap
nat 0|[1-9][0-9]*
%%
{nat} {num=atoi(yytext);
          if (num%3==0) conta_3++;
          if (num%5==0) conta_5++;}
\n        ;
.         ;
%%
int main(int argc, char *argv[])
{
--argc; /*salta il nome del programma*/
if ( argc > 0 )
          yyin = fopen(argv[1], "r" );
else
          yyin = stdin;
yylex();
printf("il numero degli interi multipli di 3 e': %d",conta_3);
printf("\n");
printf("il numero degli interi multipli di 5 e': %d",conta_5);
return 0;
}
```

Lo scanner per sua natura riconosce tutto come testo. Se vogliamo interpretarlo come numero dobbiamo fare una conversione

Scrivere un programma in flex che conti le occorrenze dei numeri multipli di 3 e dei multipli di 5 in un testo

ESERCIZI

Scrivere un programma in flex che :

1. Raddoppi tutte le occorrenze delle vocali in un file

2. Comprima sequenze di spazi e tab in un unico singolo blank, e rimuova quelli alla fine di una linea.

3. Elimini il testo inserito tra { e }

4. Elimini il testo inserito tra { e } su un'unica linea

5. Mantenga solo le linee che finiscono o cominciano con una consonante, cancellando le altre.

6. Scrivere un programma in flex che converta in un file tutte le lettere maiuscole in minuscole e tutte le lettere maiuscole in minuscole

7. Restituisca il più grande dei numeri naturali presenti nel testo

8. Conti le occorrenze dei numeri pari e di quelli dispari

ESERCIZIO 1

Scrivere un programma in flex che raddoppi tutte le occorrenze delle vocali in un file

```
%option main
%%
[aeiouAEIOU]  {ECHO;ECHO;}
```

```
vocale   [aeiouAEIOU]
%option main
%%
{vocale}   {ECHO,ECHO}
```

ESERCIZIO 2

Scrivere un programma in flex che comprima gli spazi e i tab in un unico singolo blank, e rimuova quelli alla fine di una linea.

```
%option main
%%
[ \t]+     {printf(" ");}
[ \t]+$    ;
```

ESERCIZIO 3

Scrivere un programma in flex che elimini il testo inserito tra { e }

```
%option main
%%
\{[^}]*\} ;
```

Il simbolo di escape \ è necessario perché la parentesi graffa è un simbolo speciale

Intervallo di simboli diversi da }

ESERCIZIO 4

Scrivere un programma in flex che elimini il testo inserito tra { e } che si trova su un'unica linea

```
%option main
%%
\{[^}\n]*\}
        ;
```

ESERCIZIO 5

Scrivere un programma in flex che mantenga solo le linee che finiscono o cominciano con una consonante, cancellando le altre.

```
cons [b-df-hl-np-tvz]
begin_cons ^{cons}.*$
end_cons ^.*{cons}$
%%
{begin_cons} {ECHO;printf("secondo caso\n");}
{end_cons} {ECHO; printf("primo caso\n");}
.*\n ;
%%
int main(int argc,char *argv[])
{
--argc; /* skip over program name */
if ( argc > 0 )
yyin = fopen( argv[1], "r" );
else yyin = stdin;
yylex();
return 0;
}
```

```
cons [b-df-hl-np-tvz]
begin_cons ^{cons}.*$
end_cons ^.*{cons}$
%option main
%%
{begin_cons} {ECHO;printf("secondo
caso\n");}
{end_cons} {ECHO; printf("primo
caso\n");}
.*\n ;
```

ESERCIZIO 6

Scrivere un programma in flex che converta in un file tutte le lettere maiuscole in minuscole e tutte le lettere maiuscole in minuscole

```
alfa [a-zA-Z]
%option main
%%
[A-Z] {printf("%c", *yytext+32);};
[a-z] {printf("%c", *yytext-32);}
```

ESERCIZIO 7

Scrivere un programma in flex che restituisca il più grande dei numeri naturali presenti nel testo

```
%{
int max=0;
%}
nat 0|[1-9][0-9]*
%option noyywrap
%%
{nat}   {if (atoi(yytext)>max)
        max=atoi(yytext);}
. ;
%%
int main(int argc,char *argv[])
{
--argc; /* salta il nome del programma */
if ( argc > 0 )
yyin = fopen( argv[1], "r" );
else yyin = stdin;
yylex();
printf("il valore massimo contenuto nel testo e'
%d", max);
return 0;
}
```

ESERCIZIO 8

```
%{
int contapari=0, contadispari=0;
%}
nat 0|[1-9][0-9]*
%option noyywrap
%%
{nat} {if (atoi(yytext)%2 ==0) ++contapari;
              else ++contadispari;}
. ;
%%
int main(int argc,char *argv[])
{
--argc; /* skip over program name */
if ( argc > 0 )
yyin = fopen( argv[1], "r" );
else yyin = stdin;
yylex();
printf("il numero dei pari contenuti nel testo e' %d\n", contapari);
printf("Il numero dei dispari contenuti nel testo e' %d", contadispari);
return 0;
}
```

Scrivere un programma in FLEX conti in un testo le occorrenze dei numeri pari e di quelli dispari

ESERCIZIO 9

Scrivere un programma in flex che riconosca e restituisca tutte le righe costituite da lettere minuscole che contengono tutte le 5 vocali in ordinecrescente. Se ci sono più occorrenze della stessa vocale esse devono apparire tutte prima che compaia la vocale successiva. Per esempio : zfaehipojksuj; o anche sdfgadfgartyelkjfepkjiqwrvbinnoomnjuruudfguuqw

```
spaces                          [ \t\n]
cons                            [b-df-hj-np-
tv-z]
a
        ({cons}*a)+
e
        ({cons}*e)+
i
        ({cons}*i)+
o
        ({cons}*o)+
u
        ({cons}*u)+
%%
```

Se vogliamo che ogni vocale appaia solo una volta definiamo a {cons}*a e così via

ESERCIZIO 10

Scrivere un programma in FLEX che fattorizzi un testo in maniera tale che ogni fattore sia costituito da lettere maiuscole scritte in ordine crescente. Si tratta quindi di visualizzare il testo in output inserendo uno spazio fra ogni coppia di tali fattori. Esempio: BDFGLNINORTUZTX si fattorizza come BDFGLN INORTUZ TX

Come modificare il programma in maniera tale che le lettere possano essere maiuscole o minuscole, e le maiuscole e minuscole abbiano lo stesso ordine di priorità?

Come modificarlo in maniera tale che l'ordine delle lettere in ogni fattore sia non decrescente?

```
%option main
words
        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?
%%
{words}         {ECHO; printf(" ");}
%%
```

ESERCIZIO 11

Scrivere un programma in FLEX che riconosca e restituisca tutte le righe del file costituite da lettere maiuscole in cui tutte le lettere compaiono in ordine alfabetico crescente. Per esempio:

AFHMNQSYZ

ABCDZ

FGIDOR

DGKY

BDEAZ

AFHMNQSYZ

ABCDZ

DGKY

```
 #include <stdio.h>
%option main
words
      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?
%%
{words}\n            {ECHO;}
[A-Z]+         ;
[^ \t\n]       ;
.                    ;
```

ESERCIZIO 12

Scrivere uno scanner che inserisce uno start-marker (#) e un end-marker ($) per ogni parola separata da spazi, newline o tab o punteggiatura

```
space [ \t]
newline \n
punct [;:,.!?]
letter [a-zA-Z]
%option main
%%
{letter}/({space}|{punct}|{newline})     { printf("%s$", yytext); }
{space}/{letter}                                                  {
printf("%s#", yytext); }
^{letter}
        { printf("#%s", yytext); }
.|\n
```

Lettera seguita da spazio, punteggiatura o nuova linea

Spazio seguito da lettera

Inizio riga

Per tutti gli altri simboli e newline

Aggiungi endmarker alla fine della lettera

Aggiungi startmarker alla fine dello spazio

Aggiungi startmarker prima della stringa

Riscrivi il carattere

ESERCIZIO 13

Scrivere un programma in flex che converta un file attraverso il cifrario di Cesare.

Il cifrario di Cesare è uno dei più antichi algoritmi crittografici di cui si abbia traccia storica. È un cifrario a sostituzione monoalfabetica in cui ogni lettera del testo in chiaro è sostituita nel testo cifrato dalla lettera che si trova un certo numero di posizioni dopo nell'alfabeto. Nel nostro caso useremo uno spostamento di 6 posizioni. Ovvero 'A' sarà sostituito con 'G', 'B' con 'H' e così via. La cifratura è ciclica, nel senso che 'U' è sostituito con 'A', 'V' con 'B' e 'W' con 'C' e così via.

SOLUZIONE ESERCIZIO 13

```
%{
#include <stdio.h>
 int offset=6;
%}
%option noyywrap
%%
[a-z] {printf("%c", (*yytext - 97 + offset) % 26 + 97);}
[A-Z] {printf("%c", (*yytext - 65 + offset) % 26 + 65);}
%%
int main(int argc,char *argv[])
{--argc; /* skip over program name */
if ( argc > 0 )
yyin = fopen( argv[1], "r" );
else yyin = stdin;
yylex();
return 0;
}
```

Offset fissato dal programmatore

ESERCIZI PER CASA

1. Scrivere un programma in FLEX che calcoli la lunghezza di ogni riga
2. Scrivere un programma in FLEX che sommi gli interi presenti nel testo
3. Scrivere un programma in FLEX che conti il numero delle occorrenze della parola «casa» in un testo

ESEMPIO PIÙ COMPLESSO

Si vuole costruire un analizzatore lessicale che permette la tokenizzazione di un programma relativo alla grammatica

stmt-> **if** *expr* **then** *stmt*

| **if** *expr* **then** *stmt* **else** *stmt*

| ε

expr-> *term* **relop** *term*

| *term*

term-> **id**

| **number**

relop sta per operatore relazionale

I terminali della grammatica sono **if**, **then else**, **relop**, **id** e **number** e costituiscono i nomi dei token e sono descritti dalle seguenti espressioni regolari

```
digit   -> [0-9]
digits -> [0-9]*
number -> digits (.digits)? (E[+-]? digits)?
letter -> [A-Za-z]
id      -> letter (letter|digit)*
if      -> if
then    -> then
else    -> else
relop   -> < | > | <= | >= | = | <>
```

Definizioni regolari ausiliarie

Inoltre poiché l'analizzatore lessicale dovrà riconoscere ed eliminare gli spazi bianchi ci servirà riconoscere il token

```
ws ->(blank | tab |newline)
```

Obiettivo dell'analizzatore lessicale è quello di costruire la seguente tabella che rappresenta per ogni lessema, il suo token e il suo attributo

Lessema	Nome del token	Valore dell'attributo
qualsiasi ws	---	---
if	if	---
then	then	---
else	else	---
qualsiasi id	id	Puntatore alla tabella dei simboli
qualsiasi number	number	Puntatore alla tabella dei simboli
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

COSTRUZIONE DELL'ANALIZZATORE LESSICALE

SEZIONE DEFINIZIONI

```
%{/*definizione delle costanti simboliche LT, LE, EQ, NE, GT, GE,
IF, THEN, ELSE*/
%}
/*regular definitions*/
ws              [ \t\n]*
letter          [A-Za-z]
digit           [0-9]
id              {letter}({letter}|{digit})*
number          {digit}*(\.{digit}+)? (E[+|-]?(digit)+)?
RELOP           {LT}|{LE}|{EQ}|{NE}|{GT}|{GE}
```

SEZIONE REGOLE

Token restituito allo scanner

installID e instalkNum sono procedure che inseriscono il lessema (rispettivamente di tipo id o di tipo numerico) nella tabella dei simboli. Il valore dell'indirizzo in cui viene inserito viene salvato nella variabile globale yylval, restituita da InstallID o InstallNumb

```
{ws}        ;
if                  {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = (int) installID(); return (ID);}
{number}    {yylval= (int) installNum(); return (NUMBER)}
"<"         {yylval = LT; return(RELOP)}
"<="        {yylval = LE; return(RELOP)}
"="         {yylval = EQ; return(RELOP)}
"<>"        {yylval = NE; return(RELOP)}
">"         {yylval = GT; return(RELOP)}
">="        {yylval = GE; return(RELOP)}
```

SEZIONE FUNZIONI AUSILIARIE

```
Int installid()     /* funzione per istallare nella tabella dei simboli
                          il lessema il cui primo carattere è puntato
                    da
                          yytext, e la cui lunghezza è yyleng, nella
                          tabella dei simboli e restituisce un
                    puntatore
                          a quell'indirizzo*/
Int installNum()     /* funzione simile a installID, ma mette costanti
                          numeriche in una tabella separata*/
```

È un esempio di utilizzo di un analizzatore lessicale per la tokenizzazione per un analizzatore sintattico