Generatori di analizzatori lessicali e sintattici

Analizzatori lessicali

- Analisi lessicale: riconoscere nella stringa di ingresso gruppi di simboli che corrispondono a specifiche categorie sintattiche.
- La stringa di ingresso è trasformata in una sequenza di simboli astratti: i **token**, che sono passati all'analizzatore sintattico.
- Token: coppia
  - nome: simbolo astratto che rappresenta una categoria sintattica
  - valore: una sequenza di simboli del testo di ingresso

Token	⟨Nome,Valore⟩	⟨`IDE`, **x**1⟩
Nome	Informazione che identifica una classe di token	`IDE`
Valore	Informazione che identifica uno specifico token	**x**1
Pattern	Descrizione generale della forma dei valori di un token (espressione regolare)	(**x**|**y**)(**x**|**y**|0|1)*
Lessema	Una stringa **istanza** di un pattern	**x**1

Analizzatori lessicali

- Esempi di token
  - keywords (IF,…)
  - segni di punteggiatura (;…)
  - operatori costituiti di caratteri singoli e multipli (=, ==, <=, …)
  - identificatori (stringhe che iniziano con una lettera e continuano con lettere e cifre: token IDE)
  - numeri (interi, reali)
  - commenti
- Nella stringa C

```
if (x==0) printf("Zero")
```

potrebbero essere riconosciuti i token:

⟨IF⟩, ⟨(⟩, ⟨IDE, x⟩, ⟨OPREL, ==⟩, ⟨CONST-NUM, 0⟩, ⟨)⟩, ⟨IDE, printf⟩, ⟨(⟩, ⟨CONST-STRING, Zero⟩, ⟨)⟩

3

Generatori di analizzatori lessicali

- *Input*: insieme di pattern e corrispondente *azione* da eseguire
- *Output*: **programma** che riconosce i pattern nella sua stringa d' ingresso
- LEX: noto tool di Unix per la generazione di analizzatori lessicali (scanner)
- Lex costruisce uno scanner in C da un insieme di espressioni regolari che definiscono i token e un insieme corrispondente di azioni espresse come frammenti di programmi in linguaggio C

4

Lex: Sorgente

- Struttura

  Definizioni (opzionale)

  %%

  Regole (coppie: espressione regolare azione)

  %%

  Funzioni ausiliarie (opzionale: se la sezione è vuota, il precedente separatore '%%' viene omesso)

- La prima e la terza sezione contengono rispettivamente tutte le dichiarazioni e le procedure utili alle Regole e sono ricopiate nel file C generato da Lex

5

Lex: Sorgente - Definizioni

- **Definizioni regolari**

  I_1 espressione regolare 1

  I_2 espressione regolare 2

  …

  Il nome $\{I_k\}$ potrà essere usato successivamente per denotare la k-esima espressione regolare

- **Codice C delimitato dai simboli speciali %{ e %}**
  dichiarazioni di variabili e costanti, #include, prototipi di funzioni, macro Lex

6

Lex: Sorgente – Espressioni regolari

- Sequenze di caratteri ASCII che utilizzano gli operatori:
  “\[]^-?.*+|()$/{}%<>
- Lettere e numeri del testo di ingresso sono descritti mediante loro stessi
- I caratteri non alfabetici sono descritti facendoli precedere dal carattere **\: xyz\+\+** rappresenta la sequenza **‘x’ ‘y’ ‘z’ ‘+’ ‘+’**
- Classi di caratteri: **[0123456789], [0-9]**
- Negazione di classi di caratteri: **[^0-9]**
- Tutti i caratteri eccetto fine riga: **.**
- Fine riga: **\n**; tabulazione: **\t**

7

Lex: Sorgente – Espressioni regolari

- espressione precedente opzionale: **a*b*?c**
- espressione precedente ripetuta 0 o + volte: **a*b**c**
- espressione precedente ripetuta 1 o + volte: **a*b*+c**
- espressione precedente ripetuta tra n e m volte: **a{2,5}**
- alternativa tra due espressioni: **ab|cd**
- priorità tra operatori (parentesi): **(ab|cd+)?ef** indica sequenze tipo **ef, abef, cdddef**
- espressione a inizio/fine linea: **^a / a$**
- **lookahead**: **ab/cd** indica la stringa “ab”, ma solo se seguita da “cd”; cd però non farà parte del testo riconosciuto

8

Lex: Sorgente – Azioni

- Se il codice C delle azioni comprende più di una istruzione o occupa più di una linea deve essere racchiuso tra {}
- Le azioni devono iniziare sulla stessa riga in cui termina l'espressione regolare e ne sono separate tramite spazi o tabulazioni
- carattere **;** : azione nulla (ignora il testo)
- Il testo riconosciuto viene accumulato nella variabile **yytext**, array di caratteri di lunghezza **yyleng** (int)
- Il testo non descritto da nessuna espressione regolare viene ricopiato in uscita

Lex: Sorgente – Azioni

- Funzioni
  - **yymore()**: invocabile dentro un'azione per indicare che la prossima porzione di input che farà match sarà aggiunta allo yytext corrente (che sarebbe invece normalmente sovrascritto)
  - **yyless(n):** alcuni caratteri che hanno fatto match non devono essere utilizzati; n = numero di caratteri di yytext che devono essere *conservati*
  - **input():** restituisce il prossimo carattere dell'input
  - **output(c):** scrive **c** sull'output

Lex: Sorgente – Ambiguità lessicali

- La parte iniziale di una sequenza di caratteri riconosciuta da un'espressione regolare è riconosciuta anche da una seconda espressione regolare → Lex sceglie il **match più lungo**
- La stessa sequenza di caratteri è riconosciuta da due espressioni regolari distinte → Lex sceglie la **regola dichiarata per prima**

Lex: Output

- Il file **lex.yy.c** prodotto da Lex è privo di main() e il punto di accesso è dato dalla funzione **int yylex()**
- Implementa un automa a stati finiti deterministico, riconoscitore delle espressioni regolari definite nelle regole
- Al termine di ogni azione l'automa si ricolloca sullo stato iniziale pronto a riconoscere nuovi simboli
- Legge dal file **yyin** (di default stdin) e scrive sul file **yyout** (di default stdout); è possibile specificare file di input/output

Lex: Output

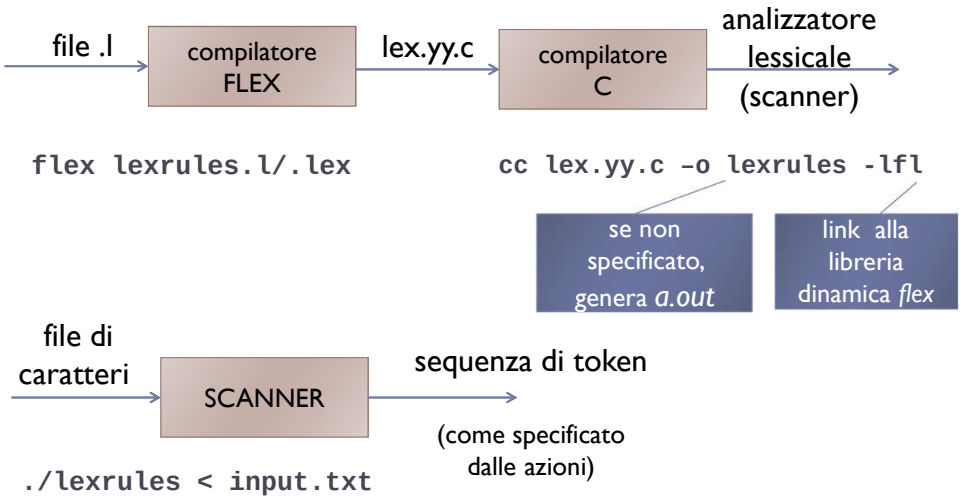
- Se non specificato diversamente nelle azioni (tramite l'istruzione `return`), tale funzione termina solo quando l'intero file di ingresso è stato analizzato
- **yylex()** chiama **input()** per leggere il (singolo) carattere di input successivo
- quando incontra EOF, chiama **yywrap()**
  - Se yywrap ritorna 1, yylex restituisce il token 0 per indicare end of file
  - Se yywrap ritorna 0, indica ulteriore input
    - yyin deve associarsi ad un altro file

FLEX (Fast LEXical analyzer generator)

- Originariamente scritto in C da Vern Paxson nel 1987, è un free software alternativo a Lex e rappresenta una versione più recente e più veloce di Lex.
- Sviluppato da M.E. Lesk e E. Schmidt presso AT&T Bell Labs
- Flex++: scanner per C++

- **http://flex.sourceforge.net/**

FLEX (Fast LEXical analyzer generator)



Lex: Esempio 1

- Il seguente sorgente Lex

```
%{
#include <stdio.h>
%}
%%
[0-9]          printf("*");
[a-z]+         printf("%c", yytext[yyleng-1]);
```

descrive 2 pattern a cui sono associate azioni di stampa. Lex genera un programma C lex.yy.c che, compilato ed eseguito sulla stringa d'ingresso

```
12caso45casa2
```

restituisce

```
**o**a*
```

Lex: Esempio 2 (ambiguità lessicale)

- Dato il file

```
%%
for       {return FOR_CMD;}
format    {return FORMAT_CMD;}
[a-z]+    {return GENERIC_ID;}
```

- e la stringa d'ingresso *format*, la funzione yylex() ritorna il valore FORMAT_CMD, preferendo la seconda regola alla prima - perché descrive una sequenza più lunga - e la seconda regola alla terza - perché definita prima nel file sorgente.

Esercizio 1

- Dato l'input:

```
pigreco := 3.14;                (:= seguito da 1 spazio)
temp1  := pigreco*(temp1+2); (:= seguito da tab)
temp2  := 6.0;
```

- scrivere un programma Lex (Definizioni e Regole) che produca in output la lista di token:

```
<IDE,pigreco>_<:=>_<FRACT,3.14><;>_
<IDE,temp1>_<:=>_<IDE,pigreco><OP2,*>
<(><IDE,temp1><OP2,+><NUM,2><)><;>_
<IDE,temp2>_<:=>_<FRACT,6.0><;>
```

Soluzione (*lexrules.l*)

```
cifra              [0-9]
cifre              [0-9]+
ide                [a-zA-Z][a-zA-Z0-9]*
separatore         [ \t\n]*
razionale          {cifre}"."{cifre}
bin                [+*]
%%
{cifre}            {printf("<NUM,%s>",yytext);}
{razionale}        {printf("<FRACT,%s>",yytext);}
":="|";"|while|"("|")"|"{"|"}"  {printf("<%s>",yytext);}
{ide}              {printf("<IDE,%s>", yytext);}
{separatore}       {printf("_");}
{bin}|"-"          {printf("<OP2,%s>", yytext);}
```

definizioni

regole

Soluzione

input.txt

```
pigreco := 3.14;
temp1 := pigreco*(temp1+2);
temp2 := 6.0;
```

Esecuzione

```
flex lexrules.l
cc lex.yy.c -o espr_aritm –lfl
./espr_aritm < input.txt
```

Esercizio 2

- Scrivere un programma Lex che elimina spazi o tab presenti *a fine riga*, e trasforma le rimanenti stringhe (interne) composte di più spazi o tab in un singolo spazio.

Soluzione (*lexrules.l*)

```
%%
[ \t]+$   ;                  //fine riga
[ \t]+    printf(" ");       //spazi/tab interni
```

Esercizio 3

- Scrivere un programma Lex che conta il numero di caratteri e di linee dell'input e riporta in output solo i conteggi.
  - inserire la stampa di riepilogo dei conteggi nella funzione (ausiliaria) main(), dopo aver invocato yylex().

Soluzione (*lexrules.l*)

```
%{
int num_lines = 0, num_chars = 0;
%}
%%
\n    {++num_lines; ++num_chars;}
.      ++num_chars;  //qualsiasi carattere eccetto fine riga
%%
main()   {
          yylex();
          printf("# of lines = %d, # of chars = %d\n",
            num_lines, num_chars);
          }
```

funzioni ausiliarie

Esercizio 4

- Scrivere un programma Lex che produca in output le linee lette in input precedute dal numero d'ordine.

Soluzione (*lexrules.l*)

```
%{
#include <stdio.h>
int l=1;
%}
linea     .*\n  //0 o più caratteri terminati da a capo
%%
{linea}   {printf("%d %s", l++, yytext);}
```

Esercizio 5

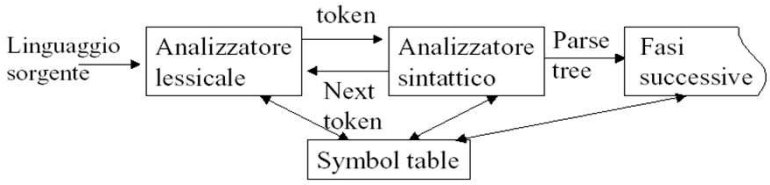
- Scrivere un programma Lex che traduca numeri in input da notazione decimale ad esadecimale e stampi il numero di sostituzioni *effettive*.

Soluzione (*lexrules.l*)

```
%{
#include <stdio.h>
#include <stdlib.h>
int cont=0;
%}
cifra  [0-9]
num    {cifra}+
%%
{num}  {int n = atoi(yytext);  //conversione stringa→intero
        printf("%x",n);        //formato esadecimale
        if (n>9) cont++; }  //conta sostituzioni effettive(da 10)
%%
main() {
       yylex();
       printf("Tot sostituzioni = %d\n",cont);
}
```

Analizzatori sintattici

- Analisi sintattica: data una sequenza *s* di token (in cui il programma sorgente è stato tradotto dall'analizzatore lessicale), si verifica se la sequenza appartenga al linguaggio sorgente del compilatore specificato dalla grammatica *G*
- L'analizzatore sintattico cerca di costruire l'albero sintattico (albero di derivazione) di *s*
  - in caso positivo, restituisce in uscita l'albero per s nella grammatica G
  - in caso negativo, restituisce un errore (errore sintattico)



29

Analizzatori sintattici

- La *sintassi* è costituita da un insieme di regole che:
  - definiscono le frasi formalmente corrette
  - permettono di assegnare ad esse una struttura (albero sintattico) che ne indica la decomposizione nei costituenti immediati
- **Grammatiche context-free (CFG)** per la descrizione dei linguaggi di programmazione
  - (+) semplici
  - (+) tabelle di parsing relativamente compatte
  - (+) algoritmi deterministici di riconoscimento delle frasi veloci (complessità lineare) e facili da realizzare
  - (-) alcune regole del linguaggio non possono essere espresse

30

Generatori di analizzatori sintattici

- *Input*: descrizione di una grammatica
- *Output*: **analizzatore sintattico per** quella **grammatica**
- YACC: *Yet Another Compiler Compiler,* tool per la generazione di analizzatori sintattici (o parser) esistente in numerose versioni
- Yacc costruisce un programma C che è un parser LALR(1) dalla descrizione di una grammatica
  - L: (Left to Right): input analizzato da sinistra a destra
  - R: derivazione rightmost
  - LA(1): 1 simbolo usato per il lookahead

31

Yacc: Sorgente

- Struttura

%{ Prologo %}	(opzionale)
Definizioni	(opzionale)
%%	
Regole	(coppie produzione azione semantica)
%%	
Funzioni ausiliarie	(opzionale: se la sezione è vuota, il separatore '%%' viene omesso)

32

Yacc: Sorgente

- **Simbolo terminale**
  - stringhe dichiarate come token nella sezione Definizioni
  - singoli caratteri racchiusi tra apici (es. '+')
- **Simbolo non-terminale**: ogni stringa alfanumerica che appare nelle produzioni e che non è stata dichiarata come token. Ogni simbolo non-terminale deve apparire dalla parte sinistra di almeno una regola
- I valori dei non-terminali sono calcolati dalle azioni semantiche;
- I valori dei token (cioè i lessemi) sono comunicati a Yacc dall'analizzatore lessicale attraverso la variabile globale **yylval** di tipo int

Yacc: Sorgente – Prologo e Definizioni

- %{ Prologo %}
  - la dichiarazione della funzione che eseguirà l'analisi lessicale, macro, #include, dichiarazioni di variabili e funzioni usate nella sezione Regole e Funzioni ausiliarie
- Definizioni
  - token della grammatica che saranno usati nella sezione Regole
    - `%token DIGIT`
  - precedenza e associatività terminali: `%left, %right,...`
  - simbolo di start: `%start <S>`. Se assente, Yacc assume per default come start symbol il non-terminale sinistro (testa) della prima produzione listata nella sezione Regole

Yacc: Sorgente – Prologo e Definizioni

- Definizioni
  - i valori restituiti dall'analizzatore lessicale sono di default interi
  - **%union** dichiara che la variabile yylval ha tipo corrispondente a quello specificato nel body di %union
    - Esempio analisi testi: `%union { char val; }`
    - Esempio manipolazione numeri reali: `%union { double val; }`
  - le keywords %left, %token, %nonass specificheranno il nuovo tipo:
    - `%token <val> DIGIT`
  - `%type <val> nonterm` specifica il tipo dei non-terminali

35

Yacc: Sorgente – Regole

- Una produzione della grammatica

$$\texttt{nonterm} \rightarrow \texttt{corpo}_1 | \ldots | \texttt{corpo}_k$$

viene espressa con le regole

```
nonterm : corpo1 {az. semantica1}
          ...
        | corpok {az. semantica_k}
        ;
```

- Produzioni con la stessa testa sono raggruppate in un unico blocco, usando la barra verticale | per separarle
- Il corpo può essere vuoto

36

Yacc: Sorgente – Regole

- **Azione semantica**: codice C racchiuso tra {} che il parser generato da Yacc deve eseguire tutte le volte che esegue una riduzione secondo la produzione associata.
- Spesso calcola il valore semantico della testa in funzione dei valori semantici dei simboli del corpo
  - `$$`: rappresenta il valore associato alla testa (non-terminale sinistro)
  - `$i`: valore dell'i-esimo terminale o non-terminale (destro) del corpo

37

Yacc: Sorgente – Esempio Regole

- La produzione: E → E + number
  in formato Yacc diventa: `e : e '+' NUMBER ;`
  NUMBER viene considerato token (se supponiamo che ce ne sia la dichiarazione nella I° sezione), + viene considerato come terminale, infine e viene considerato per esclusione un non-terminale.

- La produzione che specifica il costrutto dell'addizione può essere completata dalla seguente azione che ne calcola il valore parziale del risultato:

```
e : e '+' NUMBER { $$ = $1 + $3;};
```

38

Yacc: Sorgente – Regole

- Per terminare correttamente la sua esecuzione, Yacc si aspetta l'istruzione `return(0)`

Yacc: Sorgente – Funzioni ausiliarie

- routine di supporto utili al corretto funzionamento del parser
- routine obbligatorie:

1. **funzione main()**: fa partire il procedimento di lettura, parsing e traduzione I/O di un input. All'interno del programma Yacc si può usare semplicemente:

```
int main() {  yyparse(); }
```

dove yyparse() è il parser generato da Yacc che invoca ripetutamente yylex() innescando l'interazione parser – scanner

2. funzione **yyerror():** gestione errori sintattici. Per semplicità, può essere usata la funzione provvista da Yacc che stampa "syntax error"

```
void yyerror(char *s) {
        fprintf(stderr, "%s\n", s);
}
```

Yacc: Sorgente – Integrazione con Lex

- Altra routine obbligatoria è l'**analizzatore lessicale yylex()**: ritorna al parser i nomi dei token che devono essere stati prima dichiarati nella sezione Definizioni e ne comunica il valore (lessema) tramite **yylval**
  - yylex() può essere scritta esplicitamente in questa sezione
  - yylex() può essere generata precedentemente da Lex → inserire nella sezione Definizioni del programma Yacc la direttiva

    `#include "lex.yy.c"`
- viceversa, nella sezione Definizioni di un programma Lex, quando lo scanner viene usato in combinazione con il parser, va inserita la `#include "y.tab.h"`, il file generato da Yacc insieme a *.tab.c (se invocato con l'opzione –d) che contiene la definizione dei token

Esempio: calcolatrice

```
%{    /* PROLOGO */
  #define YYSTYPE double
  #include <math.h>
  #include <stdio.h>
%}

/* DEFINIZIONI  */
%token NUM

%left '-' '+'
%left '*' '/'
%left NEG     /* meno unario */

%% /* REGOLE E AZIONI SEMANTICHE  */
input:    /* empty */
        | input line
;

line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;

exp:      NUM                { $$ = $1;         }
        | exp '+' exp        { $$ = $1 + $3;    }
        | exp '-' exp        { $$ = $1 - $3;    }
        | exp '*' exp        { $$ = $1 * $3;    }
        | exp '/' exp        { $$ = $1 / $3;    }
        | '-' exp  %prec NEG { $$ = -$2;        }
        | '(' exp ')'        { $$ = $2;         }
;

%%  /* FUNZIONI AUSILIARIE */

#include "lex.yy.c"
```

YYSTYPE: tipo della pila interna di Yacc per i valori semantici

NUM sarà restituito da Lex

Gli operatori binari associano a sinistra
Token sulla stessa linea hanno stessa precedenza
Un token ha precedenza su tutti i token delle linee precedenti

start symbol: *input*:. Stringa vuota o sequenza finita di *line* (ricorsione sinistra)

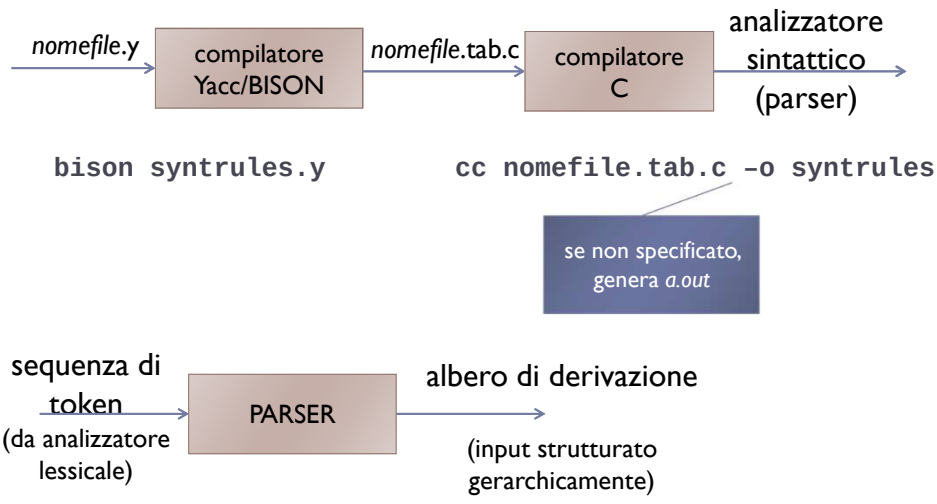
Il '-' unario ha la stessa precedenza di NEG

BISON

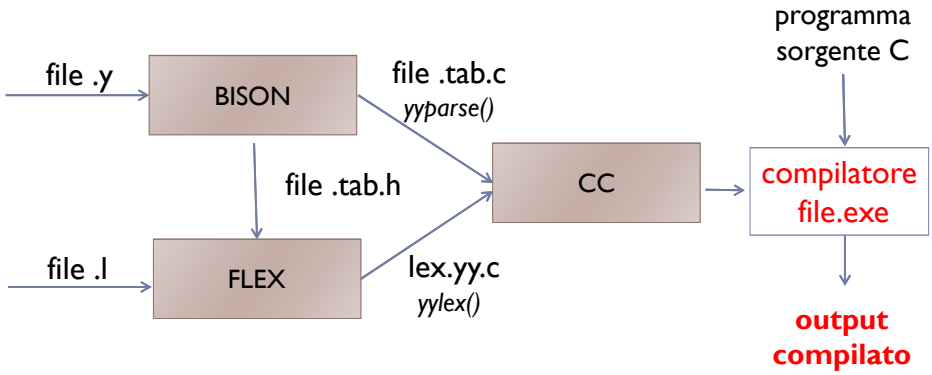
- Popolare generatore di parser LALR in C, C++ o Java
- Scritto da Robert Corbett nel 1988
- Parte del software GNU distribuito da Free Software Foundation: http://www.gnu.org/software/bison/

- Compatibile con Yacc
- Usato spesso in combinazione con FLEX

BISON



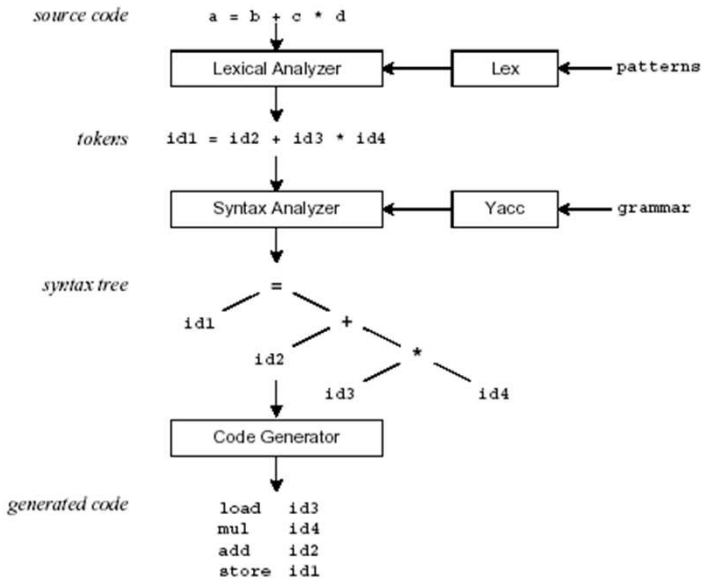
Il compilatore finale: uso combinato di FLEX & BISON




```
bison -d file.y      # crea *.tab.h (definizioni dei %token e valore
                       variabile yylval) e *.tab.c
flex file.l          # include .tab.h, crea lex.yy.c
cc -c lex.yy.c nomefile.tab.c
cc lex.yy.o nomefile.tab.o -o file.exe
```

45

Esempio: uso combinato di FLEX & BISON



46

Esercizio

- Dato il programma Yacc per la calcolatrice, modificarlo:
  - aggiungendo la funzionalità di calcolo della potenza
  - inserendo il costrutto %union per gestire i floating point
- scrivere il corrispondente programma Lex
- generare l'eseguibile finale
- testare il funzionamento della calcolatrice

Soluzione: calc2.1

```
 /* calculator  */
%{
   #include "y.tab.h"
   #include <stdlib.h>
%}

%%

[0-9.]+      {
              yylval.dval = atof(yytext); //converte da stringa a double
              return (NUM); //ritorna il token definito nel parser
             }

[-+()=/*^.\n]     { return *yytext; }
//…
```

```
//…(continua)
[ \t]   ;       /* skip whitespace */

%%

int yywrap(void) {
   return 1;
}
```

Soluzione: calc2.y

```
%{
   #include <stdio.h>
   #include <math.h>          //per usare pow()
   void yyerror(char *);
   int yylex(void);
   double atof(char *);
%}
%union {double dval;}
%token <dval> NUM             //token rappresentante un numero reale (double)
%type <dval> expression
%left '+' '-'                 //precedenza operatori
%left '*' '/'
%left NEG
%right '^'                     //esponente
%%
```


Soluzione: calc2.y

```
program:
        program statement '\n'
        |     /* NULL */
        ;

statement:
         expression                { printf("%.10g\n", $1);  /*stampa il risultato*/}
        ;

expression:
         NUM
        | expression '+' expression     { $$ = $1 + $3;}
        | expression '-' expression     { $$ = $1 - $3;}
        | expression '*' expression     { $$ = $1 * $3;}
        | expression '/' expression     { $$ = $1 / $3;}
```

Soluzione: calc2.y

```
        | '-' expression %prec NEG      { $$ = -$2; }
        | expression '^' expression      { $$ = pow($1, $3); }
        | '(' expression ')'             { $$ = $2; }
        ;

%%
void yyerror(char *s) {          //stampa errori sintattici
    fprintf(stderr, "%s\n", s);
}

int main(void) {
    yyparse();
}
```


Soluzione: Esecuzione

- `bison -y -d calc2.y` (-y genera y.tab.c, -d genera il tab.h)
- `flex calc2.l` (genera lex.yy.c)
- `gcc -c y.tab.c lex.yy.c` (genera lo scanner e il parser)
- `gcc y.tab.o lex.yy.o -o calc2.exe`
- `./calc2.exe`
- `4+4.5*2`
- `13`
- `3^4`
- `81`
- `ctrl+c per uscire`