06 – Flex

06 – A Introduzione

GNU flex è un software open source, ispirato al noto **Lex**, che genera **analizzatori lessicali** (i cosiddetti "**scanner**").

Lex è stato originariamente scritto da **Mike Lesk** ed è divenuto l'analizzatore lessicale standard nei sistemi Unix, venendo incluso anche nello standard POSIX.
Lex è ovviamente un software proprietario, a pagamento.

Flex, al contrario, è gratuito e a codice aperto, pertanto nel nostro corso useremo quest'ultimo per i nostri esempi. **Flex** ha anche il pregio di essere compatibile quasi al 100% con **Lex**.

Analisi lessicale
Come abbiamo già accennato, l'analisi lessicale:
- **riconosce campioni (pattern)**, ad esempio
  - "una sequenza di cifre"
  - "una sequenza di lettere"
  - una parola specifica, ad esempio "while"
  - un simbolo, ad esempio "<"

- **associa campioni a token**, ad esempio BEGIN, END, NUMERO.

- se necessario, **associa attributi a token**, ad esempio per il pattern 309 il valore 309 è associato al token NUMERO.

Cosa fa flex in dettaglio?

Flex legge in input un file che specifica le regole dell'analizzatore lessicale e produce in output un file che costituisce una implementazione in linguaggio C dell'analizzatore lessicale in questione.

Tale file può in seguito essere compilato con un normale compilatore C per ottenere un eseguibile in grado di comportarsi da analizzatore lessicale.

Vediamo una figura per capire meglio:

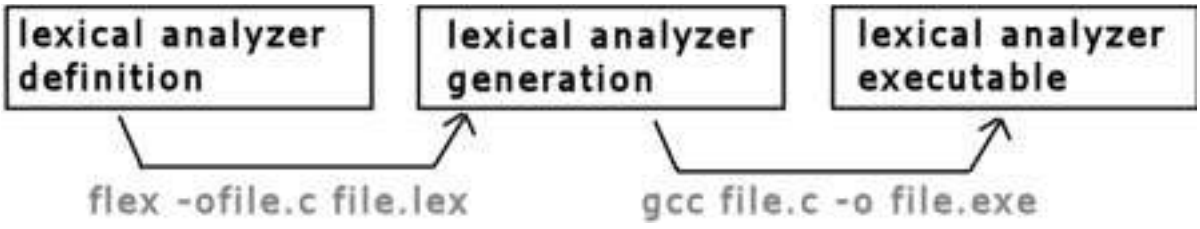

figura 06.01

06 – B GNU flex

Partiamo, come sempre, da alcune definizioni per poi chiarire tutto con un esempio.

Il file che utilizziamo per descrivere il nostro scanner può essere nominato con estensione **.flex**, ad esempio **file.flex** ; tale file è suddiviso in tre sezioni:

- **definition section**:
  qui vengono definite macro utilizzando **regex**, e vengono importati file di header scritti in linguaggio C.

- **rules section**:
  qui vengono associate regole a comandi in linguaggio C. Quando lex trova un pattern nel suo input che uguaglia una certa regola qui definita, esegue il codice C ad essa associato.
  Le regole sono semplicemente delle espressioni regolari, eventualmente contenenti le macro definite nella sezione precedente.

- **C code section**:
  Qui viene inserito il codice C dell'utente che viene copiato direttamente nel file generato da flex. Generalmente questa sezione contiene codice richiamato da alcune regole nella rules section.
  In programmi di grandi dimensioni conviene inserire questo codice in un file separato e linkarlo in fase di compilazione.

Vediamo subito un esempio di un file sorgente .flex :

```
/* file.flex */
/*////////////////////////////////////////////
// definition section                        */
%{
#define NPARI 1001
#define NDISPARI 1002
#define SALUTO 1010
%}
PARI [1-9]*[02468]
DISPARI [1-9]*[13579]
%option main
/*////////////////////////////////////////////
// rules section                             */

%%
[ ] ;
{PARI}       { printf("Trovato pari.\n") ; return(NPARI);}
{DISPARI}    { printf("Trovato dispari.\n") ; return(NDISPARI);}
"CIAO"|"AOO" { printf("Trovato saluto.\n"); return(SALUTO);}
.            { printf("Riconoscimento fallito.\n") ; exit(1) ;}
%%
/*////////////////////////////////////////////
// C code section                            */
```

Nelle prime due sezioni è possibile aggiungere direttamente codice C, inserendolo tra "**%{**" e "**%}**" oppure **indentandolo** (con un TAB, ad esempio).

E' ovviamente possibile inserire commenti con: **/* commento */** .
Attenzione: il modo di commentare (poco chiaro nel **man**) può variare con le varie versioni di flex; accertatevi del modo corretto di commentare prima di procedere.
In generale, nella **rules section** di flex è bene non posizionare i commenti ad inizio riga, mentre nelle sezioni in linguaggio C non dovrebbero esserci problemi.

Valgono le seguenti regole di priorità:

1) **longest match**: uguaglianze più lunghe hanno la precedenza: prendendo due pattern, **viva** e **vivaio**, quando trovo nell'input la parola vivaio, uguaglio il secondo pattern, non il primo.

2) **First rule**: a parità di precedenza, le regole inserite per prime nel file .flex vengono preferite.

3) **Standard action**: quando nessuna regola viene uguagliata, l'input viene passato direttamente in output.

Quando viene lanciato flex, è possibile usare l'opzione "**-l**" per forzare la piena compatibilità con lex, oppure "-s" per sopprimere la regola 3.

Nella definition section troviamo **%option main**, è il modo di aggiungere delle opzioni al comportamento di flex. "**main**" indica che flex deve provvedere una funzione **main( )** in linguaggio C per lo scanner, che quindi **non** deve essere fornita da noi.

Nella rules section troviamo regole per lo spazio [ ] , per i numeri pari o dispari, oppure per le due stringhe CIAO e AOO; infine, ogni altro carattere "." produce la stampa a schermo dell'avvertimento corrispondente.

Nella rules section le **istruzioni C** devono essere racchiuse tra **parentesi graffe**, a meno che non si tratti di una azione unica, nel cui caso le graffe non servono.

Proviamo ad eseguire i vari passaggi per la creazione dell'eseguibile:

```
sim@debian:~/lab-comp/flex$ flex -ofile.c file.flex
sim@debian:~/lab-comp/flex$ gcc file.c -o scanner1.exe
sim@debian:~/lab-comp/flex$ ./scanner1.exe
56
Trovato numero pari.
sim@debian:~/lab-comp/flex$ ./scanner1.exe
rtt
Riconoscimento fallito.
sim@debian:~/lab-comp/flex$ ./scanner1.exe
AOO
Trovato saluto.
sim@debian:~/lab-comp/flex$ ./scanner1.exe
Aoo
Riconoscimento fallito.
sim@debian:~/lab-comp/flex$
```

Nel file **.flex** è possibile "giocare" con tutta una serie di funzioni proprie di flex:

```
/* file.flex */
/*////////////////////////////////////////////
// definition section                        */
%{
%}
%option main
/*////////////////////////////////////////////
// rules section                             */

%%
[ ] ;
"ciao"   ECHO; yymore();
. { printf("Riconoscimento fallito.\n") ; exit(1) ;}
%%
/*////////////////////////////////////////////
// C code section                            */
```

Se compiliamo questo file come prima, una volta lanciato lo scanner otteniamo la stampa a schermo della parola "ciao" per due volte:

```
sim@debian:~/lab-comp/flex$ ./scanner1.exe
56
Riconoscimento fallito.
sim@debian:~/lab-comp/flex$ ./scanner1.exe
ciao
ciaociao
sim@debian:~/lab-comp/flex$
```

06 – C Alcuni usi avanzati di flex

Esempio 1
Questo esempio stampa a schermo il nome utente:

```
%option main
%%
[ ] ;
username    { printf("%s", getlogin() );}
.           { printf("Riconoscimento fallito.\n");}
%%
```

In fase di compilazione con **gcc** questo file flex dovrebbe restituire degli **warnings** (avvertimenti), il che può capitare spesso.

Più in generale, nel caso di veri e propri **errori**, invece, se non è sufficiente sistemare il file flex, occorre mettere le mani sul sorgente C generato da flex stesso; quest'ultima operazione è spesso piuttosto **complicata**.

Vediamone l'esecuzione (**[^C]** indica la combinazione di tasti **Ctrl-C**):

```
sim@debian:~/lab-comp/flex$ flex -o2.c 2-username.flex
sim@debian:~/lab-comp/flex$ gcc 2.c -o 2.exe
sim@debian:~/lab-comp/flex$ ./2.exe
ciao
Riconoscimento fallito.
username
sim
[^C]
sim@debian:~/lab-comp/flex$
```

Esempio 2
Vediamo un altro esempio, nel quale la funzione **main()** viene scritta da noi.

```
          int n_linee = 0, n_caratteri = 0;
%option noyywrap
%%
\n        n_linee++;n_caratteri++;
.         n_caratteri++;
"esci"    return (1);
%%
main ()   {
          yylex ();
          printf ("linee:%d  caratteri:%d\n",n_linee, n_caratteri);
          }
```

La opzione **noyywrap** indica che la corrispondente funzione **yywrap** non viene fornita dall'utente, ma deve essere creata da flex stesso. Vediamone l'esecuzione:

```
sim@debian:~/lab-comp/flex$ flex -o3.c 3-contaparole.flex
sim@debian:~/lab-comp/flex$ gcc 3.c -o 3.exe
sim@debian:~/lab-comp/flex$ ./3.exe
ciao a tutti
bella giornata, vero?
esci
linee: 2  caratteri: 35
sim@debian:~/lab-comp/flex$
```

Esempio 3

Ora passiamo ad un altro esempio in cui, invece che leggere da standard input, leggiamo da file:

```
%option noyywrap
%%
[aA]+      printf("e");
[eE]+      printf("i");
[iI]+      printf("o");
[oO]+      printf("u");
[uU]+      printf("a");
.         ;
%%
main (argc, argv)
          int argc; char **argv;
          {
          ++argv, --argc; /* così ignoriamo il nome del programma lanciato */
          if (argc > 0) {yyin = fopen (argv[0], "r");}
          else {printf ("Leggo da tastiera!\n"); yyin = stdin;}
          yylex ();
          }
```

Immaginiamo che il nostro file **4-esempio.txt** sia questo:

```
Una donna alta non e' mai banale...
sarà per lo sguardo necessariamente superiore.
[Flavio Giurato, 'Il tuffatore']

Le religioni sono come le lucciole:
per splendere hanno bisogno delle tenebre.
[A. Schopenhauer]
```

Vediamo compilazione ed esecuzione in presenza o meno di un file passato da riga di comando:

```
sim@debian:~/lab-comp/flex$ flex -o4.c 4-leggi-da-file.flex
sim@debian:~/lab-comp/flex$ gcc 4.c -o 4.exe
sim@debian:~/lab-comp/flex$ ./4.exe 4-esempio.txt
aeueeeuieoeei
eiuaeuiieoeiiaioui
eouoaeuoaeui

iioououuuiiaoui
iiiieuouuiiiii
euieai

sim@debian:~/lab-comp/flex$ ./4.exe
Leggo da tastiera!

[varie]

sim@debian:~/lab-comp/flex$
```

Esempio 4
Molto semplicemente utilizziamo la funzione C "**putchar**" per sostituire i TAB con lo spazio singolo, fatta eccezione per i TAB a fine riga:

```
%option main
%%
[ \t]+        putchar( ' ' );
[ \t]+$       /* ignora il token TAB a fine riga */
%%
```

Esempio 5
Qui incontriamo **yytext**, che contiene le stringhe lette dallo scanner; l'istruzione **ECHO** stampa a schermo la stringa letta, cosa che può essere ottenuta anche utilizzando una **printf** e passando **yytext** come stringa.
Come esempio, sia per parole minuscole, sia per le maiuscole, il comportamento sarà di stampare a schermo la stringa letta.

Il simbolo pipe "**|**" indica che l'azione per quel pattern sarà uguale all'azione per il pattern sottostante: per il pattern "banano" l'azione corrispondente sarà quella indicata per il pattern "maleducato":

```
%option main
%%
"antoni[o]+"     ECHO;
"giampier[o]+"   printf ("Che te rode?");
"banano"         |
"maleducato"     {printf ("%s sarai tu!", yytext);}
[A-Z ]+          printf ("%s? Hmmm...", yytext);}
[a-z ]+          ECHO;
%%
```

Vediamo la relativa esecuzione:

```
sim@debian:~/lab-comp/flex$ flex -o5.c 5-maleducato.flex
sim@debian:~/lab-comp/flex$ gcc 5.c -o 5.exe
sim@debian:~/lab-comp/flex$ ./5.exe
ciao!
ciao!
antonioooooo
antonioooooo
giampieroo
Che te rode?
maleducato
maleducato sarai tu!
banano!
banano sarai tu!!
ANDIAMO?
ANDIAMO? hmmm...?

sim@debian:~/lab-comp/flex$
```

Da notare che il punto esclamativo dopo la parola "banano" viene messo in fondo alla frase "banano sarai tu!".

Esempio 6
In flex possiamo leggere la variabile **yyleng** che indica la lunghezza, in caratteri, dell'ultimo "match" (corrispondenza, uguaglianza). Possiamo anche accedere a yytext come vettore, e usare yyleng per "giocare" un po' con l'ultima stringa uguagliata. Vediamo un esempio, e la relativa esecuzione:

```
%option main
%%
[a-zA-Z ]+  { int i;
              for (i = 1; i <= yyleng; i++)
              printf ("%c", yytext[yyleng-i]); }
%%
```

```
sim@debian:~/lab-comp/flex$ flex -o6.c 6-yyleng.flex
sim@debian:~/lab-comp/flex$ gcc 6.c -o 6.exe
sim@debian:~/lab-comp/flex$ ./6.exe
simone
enomis
brunozzi
izzonurb
maleducato
otacudelam

sim@debian:~/lab-comp/flex$
```

Esempio 7

Vediamo un esempio più consistente.

```
/* scanner per un linguaggio simil-Pascal */
%{
#include <math.h>
%}
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
%option noyywrap
%%
{DIGIT}+    {
            printf( "Numero intero: %s (%d)\n", yytext,
                    atoi( yytext ) );
            }
{DIGIT}+"."{DIGIT}*        {
            printf( "Numero reale: %s (%g)\n", yytext,
                    atof( yytext ) );
            }

if|then|begin|end|procedure|function        {
            printf( "Parola chiave: %s\n", yytext );
            }

{ID}        printf( "Identificatore: %s\n", yytext );

"+"|"-"|"*"|"/"   printf( "Operatore: %s\n", yytext );

"{"[^}\n]*"}"     /* elimina i commenti */

[ \t\n]+          /* elimina gli spazi inutili a fine riga */

.           printf( "Carattere sconosciuto: %s\n", yytext );

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

06 – D Approfondimenti su flex

Il file di output generato da flex a partire dal sorgente da noi scritto contiene una routine chiamata **yylex()**, un certo numero di tabelle usate per uguagliare token, e un certo numero di routine ausiliarie. Di default, **yylex()** viene dichiarato come segue:

```
int yylex()
    {
    ... varie definizioni ed azioni ...
    }
```

Quando **yylex()** viene invocata, esegue la scansione di token dal file globale di input chiamato **yyin**, che legge di default dallo standard input.
La funzione **yylex()** continua ad eseguire la scansione finchè non raggiunge un **EOF** (End Of File) oppure una delle azioni collegate ad un token letto eseguono un **return**.
Quando **yylex** raggiunge un **EOF**, eventuali successive chiamate ad essa sono indefinite, a meno che **yyin** non venga puntato ad un nuovo file di input, oppure venga invocata la funzione **yyrestart()**; tale funzione prende come argomento un puntatore a file, **'FILE *'**, e inizializza **yyin** per effettuare la scansione da quel file.
La maniera in cui lo scanner legge caratteri dall'input viene definita dalla macro **YY_INPUT**, che di default legge dal puntatore di file globale **yyin**.

Quando lo scanner riceve un **EOF** da **YY_INPUT**, esegue la funzione **yywrap()**; nel caso in cui essa restituisca **false** (zero), si assume che tale funzione abbia proseguito e fatto puntare **yyin** ad un altro file di input; in tal caso la scansione continua.
Se invece **yywrap()** restituisce **true** (non-zero), allora lo scanner termina la sua esecuzione.
Nel caso in cui non venga fornita una propria versione di **yywrap()**, è necesario aggiungere l'opzione seguente:

```
%option noyywrap
```

Vediamo ora di spiegare l'utilizzo delle start conditions.

Condizioni inziali (start conditions)

In flex è possibile utilizzare un meccanismo per **attivare regole secondo certe condizioni**; tale meccanismo usufruisce delle start conditions.

L'utilità di questo meccanismo è data dal poter utilizzare **regole differenti a seconda della porzione di input analizzata**; un esempio banale è dato da un codice sorgente in un linguaggio di programmazione, che possiamo considerare suddiviso in **codice** e **commenti**.

Ogni **regola** il cui **pattern** venga preceduto da **<sc>** sarà attiva solo quando lo scanner è nella start condition denominata **sc**.

Le start conditions vengono dichiarate nella **definition section**, utilizzando linee senza indentazione che iniziano per **%s** oppure per **%x**, seguite da una lista di nomi. Il **%s** indica delle condizioni **inclusive**, il **%x** delle condizioni **exclusive**.

Una start condition è attivata usando l'azione **BEGIN**; finchè la successiva azione **BEGIN** non viene eseguita, le regole con la start condition scelta sono **attive**, e quelle con altre start condition sono **inattive**.
Se la start condition è **inclusiva**, allora le regole **prive** di start condition saranno attive anch'esse; se la start condition è **esclusiva**, allora solo le regole qualificate con una start condition saranno attive.
Chiariamo subito con un esempio per capire la differenza tra **%s** e **%x**, mostrando due codici **equivalenti nel comportamento**:

```
%s condiz1
%%

<condiz1>ciao          printf(“pippo”);
bye                    printf(“pluto”);
```

```
%x condiz1
%%

<condiz1>ciao          printf(“pippo”);
<INITIAL,condiz1>bye   printf(“pluto”);
```

La parte **<INITIAL,condiz1>** è necessaria perchè, usando semplicemente **<condiz1>** per qualificare la stringa “**bye**”, le relative regole sarebbero attive solo nella condizione esempio, ma non in **INITIAL** (che indica lo stato all'inizio della scansione), mentre nel primo esempio le regole per “**bye**” sono attive sia all'inizio, sia nella condizione **condiz-1**.
Tale differenza è data dal fatto che, nel caso di **%s**, le regole prive di start conditions sono rese **attive**.
Manca un tassello per comprendere appieno l'utilizzo delle start condition, ovvero come utilizzare **BEGIN** per passare alle varie condizioni. Nel seguente esempio lo scanner enterà nella condizione **condiz1** a patto che la variabile **valore** sia diversa da zero:

```
        int valore;

%x condiz1
%%
        if ( valore )
            BEGIN(condiz1);

<condiz1>eureka    printf(“Ho trovato!”);

...altre regole...
```

Detto ciò, mostriamo un esempio più complesso, in cui lo scanner può interpretare in **due maniere differenti** una stringa come “**123.456**”, a seconda che sia preceduta o meno dalla stringa “**expect-floats**”. Invito gli studenti ad esercitarsi con tale esempio e se possibile modificarlo a piacere, inserendo ulteriori funzionalità.

```
%{ #include <math.h> %}
%s expect
%%
expect-floats                 BEGIN(expect);

<expect>[0-9]+"."[0-9]+       {
                              printf( "trovato reale = %f\n",
                              atof( yytext ) ); }
<expect>\n                    {
                     /* essendo fine riga, abbiamo bisogno di un
                        altro “expect-number” prima di riconoscere
                        ulteriori numeri */
                              BEGIN(INITIAL);
                              }

[0-9]+                        {
                              printf( "trovato intero = %d\n",
                              atoi( yytext ) );
                              }

"."                           printf( "trovato un punto\n" );
```

07 – Parsing

07 – A Analisi sintattica / parsing

Dopo aver affrontato gli argomenti riguardanti l'analisi lessicale, e aver sperimentato l'uso del software **flex**, passiamo alla fase successiva: l'**analisi sintattica**, comunemente detta **parsing**, che consiste, detto in parole povere, nel **determinare** se una stringa di **token** può essere generata da una **grammatica**.

Quando iniziamo l'analisi sintattica del nostro codice abbiamo già a disposizione i risultati dell'analisi lessicale (scanning), ovvero una suddivisione dei vari simboli incontrati, organizzata in una **symbol table**, come può essere questa:

```
pos := val + rate * 60;
```

pos	:=	val	+	rate	*	60	;
ID.0	ASS_OP	ID.1	AR_OP	ID.2	AR_OP	NUM	TERM

ID.n: identificativo di una variabile
ASS_OP: operazione di assegnamento
AR_OP: operazione aritmetica
NUM: numero
TERM: simbolo terminale della istruzione

L'analisi sintattica consiste nel "**riconoscere**" delle sentenze (ad es. associare un valore ad una variabile, un ciclo if, una funzione), e associare tali sentenze ad un **parse tree**.

In parole più povere, l'analisi lessicale associa ai **token** il tipo ed il valore, mentre quella sintattica cerca di riconoscere la struttura di un determinato linguaggio, ad es. un linguaggio di programmazione.

Alcune considerazioni e definizioni

I linguaggi di programmazione hanno regole a cui sottende la struttura sintattica di programmi ben formati. Nel linguaggio **C**, ad esempio, un programma è fatto da blocchi, un blocco è fatto da statement, uno statement è fatto da espressioni, le espressioni sono composte da token, e così via.

La sintassi dei costrutti dei linguaggi di programmazione può essere descritta da **context-free grammars**, o notazione **BNF** (Backus-Naur Form).

Le grammatiche offrono vantaggi significanti:
- una grammatica fornisce una precisa **specifica sintattica**;

- da alcune classi di grammatiche possiamo costruire un parser che determina se un programma sorgente è **ben formato**. Come beneficio addizionale, il processo di parsing rivela **ambiguità sintattiche** che potrebbero altrimenti non essere riconosciute;
- Se un linguaggio **evolve**, i suoi nuovi costrutti possono essere aggiunti con facilità.

I parser basati sugli algoritmi di **Cocke-Younger-Kasami**, o di **Earley**, possono "parsare" qualsiasi tipo di grammatica, ma sono talmente **inefficienti** in termini prestazionali da non essere interessanti in applicazioni pratiche; verranno perciò ignorati nel nostro corso.

Escludendo tale categoria possiamo perciò **suddividere** i parser in due grandi classi: **top-down** e **bottom-up**; tali definizioni si riferiscono all'ordine in cui i nodi del **parse tree** vengono costruiti.
Nel **top-down** la costruzione parte dalla radice dell'albero e procede verso le foglie; nel **bottom-up** la costruzione parte dalle foglie per arrivare alla radice.

I parser top-down o bottom-up più efficienti lavorano soltanto con alcune **classi di grammatiche**, come le **LL** e le **LR**, ma questa limitazione non impedisce di gestire la stragrande maggioranza dei costrutti, e in generale è comunque possibile applicarli ai moderni linguaggi di programmazione.

Un **LL-parser** effettua il parsing dell'input da sinistra (**L**eft) a destra, e costruisce una derivazione **L**eftmost (si espande sempre il simbolo nonterminale più a sinistra) della sentenza.

Un **LR-parser** effettua il parsing dell'input da sinistra (**L**eft) a destra, e costruisce una derivazione **R**ightmost (si espande sempre il simbolo nonterminale più a destra) della sentenza.

Vengono spesso usati i termini **LR(1)-parser**, oppure **LL(1)-parser**, dove il numero indica il numero di simboli di input "**look ahead**" (ovvero, esaminati in avanti) non consumati che vengono usati per prendere decisioni nella costruzione dell'albero di parsing.
Quando tale numero è **1** viene spesso omesso.

Questo schema sintetizza le interazioni tra **analisi lessicale** e **parsing**:

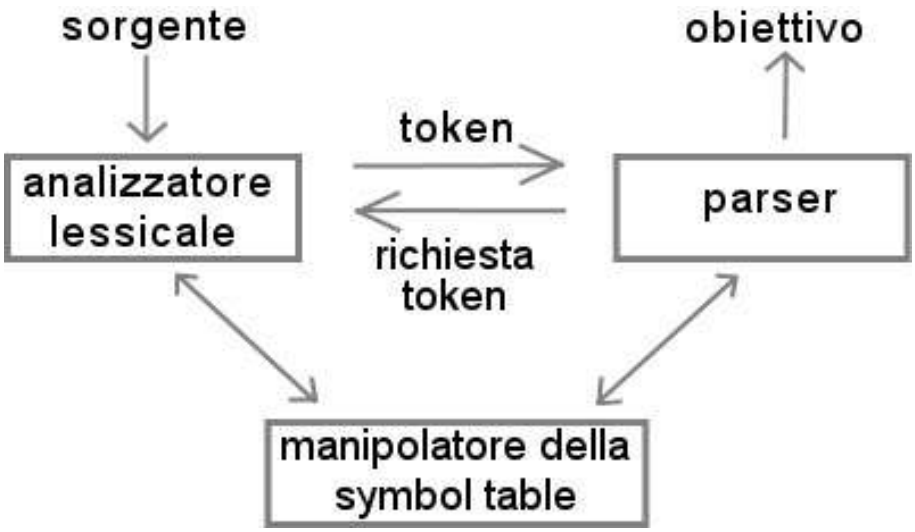

figura 07.01

Vediamo ora un esempio di grammatica e relativo parsing, sia LL che LR.

Grammatica:
S --> if E then S else S fi | while E do S od | skip
E --> num | id

esempio 1:

```
1   2    3
if  num  then
4       5   6   7     8
while   id  do  skip  od
9     10
else  skip
11
fi
```

Vediamo il parsing con algoritmo **top-down**, che equivale in questo caso ad un parsing **LL(1)**:

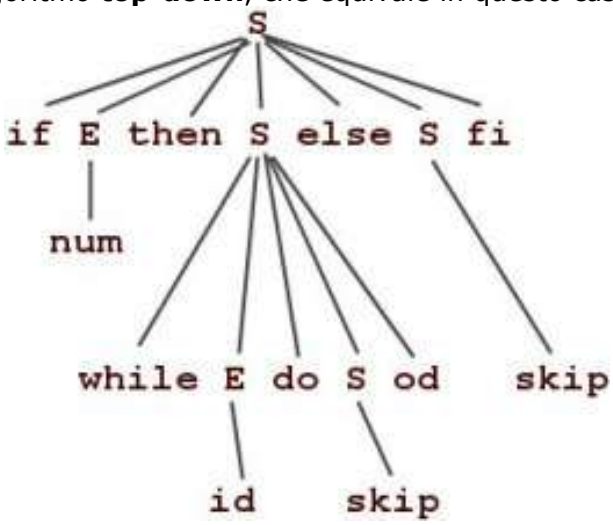

figura 07.02

Qui di seguito i vari passi, evidenziando in grassetto sottolineato l'applicazione delle regole grammaticali:

LL(1)-parsing, top-down:

```
S --> if E then S else S fi
--> if num then S else S fi
--> if num then while E do S od else S fi
--> if num then while id do S od else S fi
--> if num then while id do skip od else S fi
--> if num then while id do skip od else skip fi
```

Vediamo invece il parsing con algoritmo **bottom-up**, che in questo caso equivale ad un **LR**:

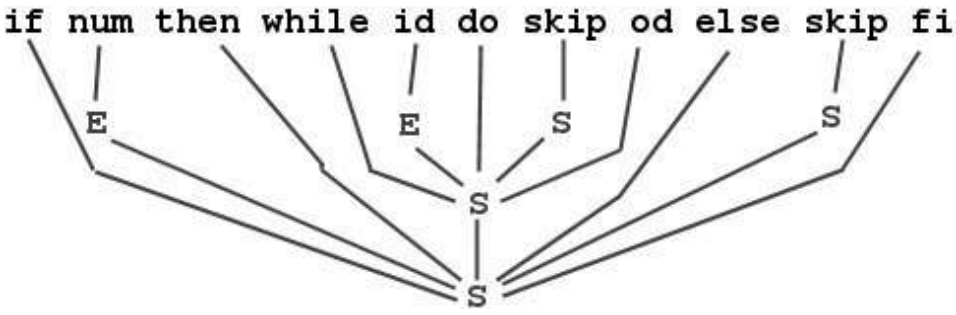

figura 07.03

Qui di seguito i vari passi, evidenziando in grassetto sottolineato ciò che cambia:

LR-parsing, con bottom-up:

```
if num then while id do skip od else skip fi <--
if E then while id do skip else skip fi <--
if E then while E do skip else skip fi <--
if E then while E do S else skip fi <--
if E then S else skip fi <--
if E then S else S fi <--
S
```

Le **grammatiche context-free** sono riconoscibili da **automi a pila**, nel caso specifico di algoritmo **top-down**.

L'algoritmo **top-down** utilizza una tabella che possiamo definire come **tabella di parsing**; eccone un esempio per la nostra grammatica di prima, in cui il simbolo **$** indica il fondo della pila:

Token	S	E
id	-	E --> id
num	-	E --> num
if	S --> if E then S else S	-
then	-	-
else	-	-
fi	-	-
while	S --> while E do S od	-
do	-	-
od	-	-
skip	S --> skip	-
$	-	-

figura 07.04

Mostriamo i movimenti della pila durante la fase di parsing in un **top-down**, evidenziando in giallo i **token** che vengono tirati fuori dalla pila (azione **pop**) e inseriti nel parse tree, e in grigio i simboli **nonterminali** che vengono espansi:

1.

$	**S**								

2.

$	**fi**	**S**	**else**	**S**	**then**	**E**	**if**		

3.

$	**fi**	**S**	**else**	**S**	**then**	**E**			

4.

$	**fi**	**S**	**else**	**S**	**then**	**num**			

5.

$	**fi**	**S**	**else**	**S**	**then**				

6.

$	**fi**	**S**	**else**	**S**					

7.

$	**fi**	**S**	**else**	**od**	**S**	**do**	**E**	**while**	

8.

$	**fi**	**S**	**else**	**od**	**S**	**do**	**E**		

9.

$	**fi**	**S**	**else**	**od**	**S**	**do**	**id**		

10.

$	**fi**	**S**	**else**	**od**	**S**	**do**			

11.

$	**fi**	**S**	**else**	**od**	**S**				

12.

$	**fi**	**S**	**else**	**od**	**skip**				

13.

$	**fi**	**S**	**else**	**od**					

14.

$	**fi**	**S**	**else**						

15.

$	**fi**	**S**							

16.

$	**fi**	**skip**							

17.

$	**fi**								

18.

$									

figura 07.05

Proviamo ora a vedere il comportamento per un algoritmo **bottom-up**, facendo attenzione alle azioni di **shift** (“mettere il token nello stack”) e alle azioni di **reduce** (sostituire un token con un simbolo, basandosi sulla grammatica).

(1) token	**azione**	**Pila -->**									
if	shift	if									

(2) token	**azione**	**Pila -->**									
num	shift	if	num								

(3) token	**azione**	**Pila -->**									
then	reduce	if	E								

(4) token	**azione**	**Pila -->**									
-	shift	if	E	then							

(5) token	**azione**	**Pila -->**									
while	shift	if	E	then	while						

(6) token	azione	Pila -->									
id	shift	if	E	then	while	id					

(7) token	azione	Pila -->									
do	reduce	if	E	then	while	E					

(8) token	azione	Pila -->									
-	shift	if	E	then	while	E	do				

(9) token	azione	Pila -->									
skip	shift	if	E	then	while	E	do	skip			

(10) token	azione	Pila -->									
od	reduce	if	E	then	while	E	do	S			

(11) token	azione	Pila -->									
-	shift	if	E	then	while	E	do	S	od		

(12) token	azione	Pila -->									
else	reduce	if	E	then	S						

(13) token	azione	Pila -->									
-	shift	if	E	then	S	else					

(14) token	azione	Pila -->									
skip	shift	if	E	then	S	else	skip				

(15) token	azione	Pila -->									
fi	reduce	if	E	then	S	else	S				

(16) token	azione	Pila -->									
-	shift	if	E	then	S	else	S	fi			

(17) token	azione	Pila -->									
-	reduce	S									

Figura 07.06

07 – B bison

Bison è un software reso disponibile dal **GNU** project della **F**ree **S**oftware **F**oundation (**FSF**); consiste in un generatore di parser che di fatto converte una descrizione per una grammatica in un programma C in grado di effettuare il parsing di tale grammatica.

Nel caso particolare di Bison, si tratta di una grammatica **LALR**, ovvero Look-Ahead LR.

Bison è compatibile con **Yacc** (**Y**et **A**nother **C**ompiler-**C**ompiler), e generalmente è in grado di accettare senza ulteriori modifiche le grammatiche scritte per Yacc.

Il **bison source** (sorgente bison) è un file che definisce delle regole grammaticali; una volta "compilato" dal software **bison**, produce un file costituito da istruzioni C, che possiamo chiamare **bison parser**.

Nel **bison parser** è definita una funzione chiamata **yyparse** che implementa la grammatica definita nel bison source.

Tale funzione non è sufficiente, da sola, ad eseguire il task assegnato: dobbiamo preoccuparci noi di fornire un **analizzatore lessicale** (ad esempio con flex), una funzione che ci comunica eventuali **errori** in fase di parsing, e ovviamente la funzione **main**, la quale deve a sua volta invocare **yyparse**.

E' convenzione nominare i bison source con l'estensione **.y** (ad es. **prova.y**).

Vediamo l'aspetto di un tipico **bison source**:

```
%{
C code
%}

Bison declarations

%%
%{
C code
%}

Grammar rules

%%

Additional C code
```

Vediamo ora un esempio concreto di un piccolo **bison source**, una calcolatrice con notazione polacca inversa:

```
* Calcolatrice con notazione polacca inversa */

%{
#define YYSTYPE double
#include <math.h>
#include <ctype.h>
#include <stdio.h>
%}
%token NUM
%%
input:      /* vuoto */
            | input line;
line: '\n'
            | exp '\n'  {printf("\t%.10g\n", $1);};
exp:        NUM               {$$ = $1;   }
            | exp exp '+'           {$$ = $1 + $2;    }
            | exp exp '-'           {$$ = $1 - $2;    }
            | exp exp '*'           {$$ = $1 * $2;    }
            | exp exp '/'           {$$ = $1 / $2;    };
%%
/* analizzatore lessicale */

int yylex (void)
      {     int c;

      /* elimina gli spazi vuoti*/
      while (  (c = getchar()) == ' ' || c == '\t' ) ;

      /* processa i numeri */
      if (c == '.' || isdigit (c) )
            {
            ungetc (c,stdin);
            scanf ("%lf",&yylval);
            return NUM;
            }

      /* restituisce la fine dell'input */
      if (c == EOF) return 0;

      /* restituisce un singolo caratere */
      return c;
      }

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

int yyerror (const char *s)
      {
      printf ("%s\n", s);
      }
```

Vediamo come **compilare** un bison source per poi utilizzarlo. Una volta lanciato, bison crea un file con estensione **.tab.c** , il quale va poi compilato per ottenere un eseguibile.

```
sim@debian:~/$ ls
rpcalc.y

sim@debian:~/$ bison rpcalc.y
sim@debian:~/$ ls
rpcalc.y   rpcalc.tab.c

sim@debian:~/$ gcc rpcalc.tab.c -o rpcalc.exe
sim@debian:~/$ ./rpcalc.exe
5 22 +
        27
13 4 -
        9
4 6 *
        24
15 3 /
        5
```

Invito gli studenti ad esercitarsi con questo esempio, e a modificarlo rendendo la calcolatrice più completa.

Precedenza degli operatori

Quando viene dichiarato un token si può utilizzare **%token**, oppure una parola chiave diversa per specificare l'associatività degli operatori, in particolare **%left**, **%right**, **%nonassoc**.

Tali parole chiave vengono chiamate "**precedence declarations**", dichiarazioni di precedenza.
La sintassi è identica a quella relativa a %token, ovvero:

%left symbols...
%left <type> symbols...

L'associatività di un operatore **op1** determina come vengono considerati usi ripetuti di tale operatore: l'espressione
x op1 y op1 z
viene analizzata (parsed) raggruppando dapprima **x con y** se l'operatore è definito con **%left**, oppure **y con z** se l'operatore è definito con **%right**.

%nonassoc specifica invece una non associatività, e nel nostro caso porterebbe ad un errore.
La associatività sinistra o destra di un operatore determina come esso si annidi con altri operatori.

Tutti i token dichiarati in una **singola dichiarazione di precedenza** hanno identica precedenza, e si annidano insieme in base alla loro associatività.

Quando invece due token, dichiarati in **diverse dichiarazioni di precedenza**, vengono associati, quello dichiarato per **ultimo** ha precedenza maggiore e quindi viene raggruppato per primo.

Invito gli studenti ad esercitarsi con esempi propri sulle "precedence declarations", una parte molto importante e spesso causa di errori incomprensibili.

07 – C flex e bison insieme

Vediamo ora un esempio di utilizzo di bison in combinazione con flex.

Esempio 1: semplice calcolatrice

```
/*  file: simple_calc.y  */
%{
#define YYSTYPE int
%}

%token NUM PIU MENO PER DIVISO ACCAPO

%%
input:    /* empty */
        | input line  ;

line:   ACCAPO
        | exp ACCAPO ;

exp:      NUM
        | exp exp PIU
        | exp exp MENO
        | exp exp DIVISO
        | exp exp PER  ;
%%

int yyerror(char *s)
{
    extern char yytext[];
    printf("Al token \"%s\": %s\n", yytext, s);
}

int main() { yydebug = 1; yyparse(); }
```

```
/*  file: simple_calc.flex  */
%{
#include"simple_calc.tab.h"

%}
%option noyywrap
%%
[ \t\r\f]              ;
[\n]                   { return(ACCAPO); }
"+"                    { return(PIU); }
"-"                    { return(MENO); }
"*"                    { return(PER); }
"/"                    { return(DIVISO); }
[1-9][0-9]*            {
                          int i;
                          yylval = sscanf(yytext,"%d",&i);
                          return(NUM);
                        }
.                         { printf("Non riconosciuto.\n"); exit(1);}
%%
```

Da questi due file è possibile generare un eseguibile che risponde alle regole da noi specificate, che vediamo alla prossima pagina.

L'opzione **-l** usata con **flex** serve per attivare la massima compatibilità possibile con l'originale lex dello Unix di AT&T.

Vediamo le opzioni usate con **bison**:
-t : il parser viene eseguito in trace mode (modalità traccia)
-d : produce un file di header
-v : produce un file .output che descrive l'automa.

```
sim@debian:~/$ ls
simple_calc.y          simple_calc.flex

sim@debian:~/$ bison -t -d -v simple_calc.y
sim@debian:~/$ ls
simple_calc.y          simple_calc.flex
simple_calc.tab.h      simple_calc.output
simple_calc.tab.c

sim@debian:~/$ flex -osimple_calc.yy.c -l simple_calc.flex
sim@debian:~/$ ls
simple_calc.y          simple_calc.flex
simple_calc.tab.h      simple_calc.output
simple_calc.tab.c      simple_calc.yy.c

sim@debian:~/$ gcc simple_calc.tab.c simple_calc.yy.c -o simple.exe
sim@debian:~/$ cat simple_calc.esempio
4 9 +

sim@debian:~/$ ./simple.exe < simple_calc.esempio

Starting parse
Entering state 0
Reducing via rule 1 (line 8),  -> input
state stack now 0
Entering state 1
Reading a token: Next token is 257 (NUM)
Shifting token 257 (NUM), Entering state 2
Reducing via rule 5 (line 14), NUM  -> exp
state stack now 0 1
Entering state 5
Reading a token: Next token is 257 (NUM)
Shifting token 257 (NUM), Entering state 2
Reducing via rule 5 (line 14), NUM  -> exp
state stack now 0 1 5
Entering state 7
Reading a token: Next token is 258 (PIU)
Shifting token 258 (PIU), Entering state 8
Reducing via rule 6 (line 15), exp exp PIU  -> exp
state stack now 0 1
Entering state 5
Reading a token: Next token is 262 (ACCAPO)
Shifting token 262 (ACCAPO), Entering state 6
Reducing via rule 4 (line 12), exp ACCAPO  -> line
state stack now 0 1
Entering state 4
Reducing via rule 2 (line 9), input line  -> input
state stack now 0
Entering state 1
Reading a token: Now at end of input.
Shifting token 0 ($), Entering state 12
Now at end of input.
sim@debian:~/$
```

Gli **stati** a cui si riferisce il nostro output (qui in debug mode, come indicato nel file .y) derivano dalla notazione usata nel file **simple_calc.output**, che qui vediamo.

Per generare tale file è sufficiente lanciare **bison** aggiungendo l'opzione **-v**.

```
/*  file: simple_calc.output  */
Grammar
  Number, Line, Rule
    1   8 input -> /* empty */
    2   9 input -> input line
    3  11 line -> ACCAPO
    4  12 line -> exp ACCAPO
    5  14 exp -> NUM
    6  15 exp -> exp exp PIU
    7  16 exp -> exp exp MENO
    8  17 exp -> exp exp DIVISO
    9  18 exp -> exp exp PER

Terminals, with rules where they appear

$ (-1)
error (256)
NUM (257) 5
PIU (258) 6
MENO (259) 7
PER (260) 9
DIVISO (261) 8
ACCAPO (262) 3 4

Nonterminals, with rules where they appear
input (9)
    on left: 1 2, on right: 2
line (10)
    on left: 3 4, on right: 2
exp (11)
    on left: 5 6 7 8 9, on right: 4 6 7 8 9

state 0
    $default    reduce using rule 1 (input)
    input       go to state 1

state 1
    input  ->  input . line   (rule 2)
    $           go to state 12
    NUM         shift, and go to state 2
    ACCAPO      shift, and go to state 3
    line        go to state 4
    exp         go to state 5

state 2
    exp  ->  NUM .   (rule 5)
    $default    reduce using rule 5 (exp)

state 3
    line  ->  ACCAPO .   (rule 3)
    $default    reduce using rule 3 (line)

state 4
    input  ->  input line .   (rule 2)
    $default    reduce using rule 2 (input)

state 5
    line  ->  exp . ACCAPO   (rule 4)
    exp  ->  exp . exp PIU   (rule 6)
    exp  ->  exp . exp MENO   (rule 7)
    exp  ->  exp . exp DIVISO   (rule 8)
    exp  ->  exp . exp PER   (rule 9)
    NUM         shift, and go to state 2
    ACCAPO      shift, and go to state 6
    exp         go to state 7
```

```
/*  file: simple_calc.output  CONTINUA  */

state 6
    line  ->  exp ACCAPO .   (rule 4)
    $default    reduce using rule 4 (line)

state 7
    exp  ->  exp . exp PIU   (rule 6)
    exp  ->  exp exp . PIU   (rule 6)
    exp  ->  exp . exp MENO   (rule 7)
    exp  ->  exp exp . MENO   (rule 7)
    exp  ->  exp . exp DIVISO   (rule 8)
    exp  ->  exp exp . DIVISO   (rule 8)
    exp  ->  exp . exp PER   (rule 9)
    exp  ->  exp exp . PER   (rule 9)
    NUM         shift, and go to state 2
    PIU         shift, and go to state 8
    MENO        shift, and go to state 9
    PER         shift, and go to state 10
    DIVISO      shift, and go to state 11
    exp         go to state 7

state 8
    exp  ->  exp exp PIU .   (rule 6)
    $default    reduce using rule 6 (exp)

state 9
    exp  ->  exp exp MENO .   (rule 7)
    $default    reduce using rule 7 (exp)

state 10
    exp  ->  exp exp PER .   (rule 9)
    $default    reduce using rule 9 (exp)

state 11
    exp  ->  exp exp DIVISO .   (rule 8)
    $default    reduce using rule 8 (exp)

state 12
    $           go to state 13

state 13
    $default    accept
```

07 – D VCG (Visualization of Compiler Graphs)

Eseguendo bison con l'opzione **-g** si richiede la generazione di un file con estensione vcg: si tratta di un formato particolare per descrivere dei grafici.

E' possibile visualizzare tali grafici con tool appositi, come ad esempio **aiSee** per Windows, reperibile a questo URI:

http://www.absint.com/aisee/download/

Suggerisco anche questo link, completo di molte informazioni anche per Linux:
http://rw4.cs.uni-sb.de/~sander/html/gsvcg1.html#examples

Una volta installato, bisogna aprire il file **vcg** da noi creato per poter visualizzare un grafico come questo:

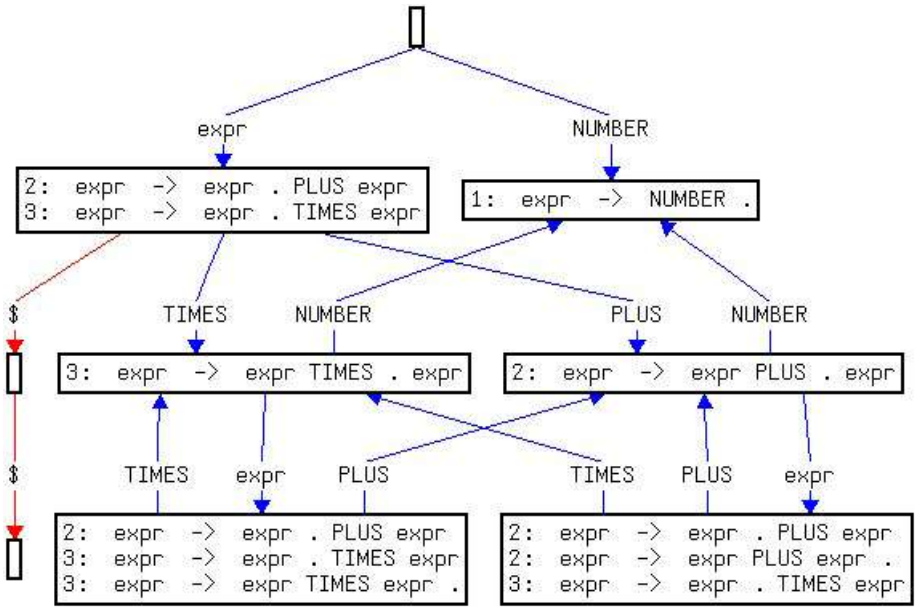

figura 07.07

Si tratta della rappresentazione di un automa, e questo schema dovrebbe permettere di capire il funzionamento della nostra grammatica.

Invito gli studenti a provare questa cosa con propri esempi, e cercare di capire bene il funzionamento della grammatica scelta.

07 – E Conflitti

Nel corso del parsing l'automa che si occupa di prelevare i token e trasformarli in base alle regole grammaticali può incontrare dei **conflitti**, quando ad esempio non sa se effettuare una operazione di **shift** oppure di **reduce**.

Si definiscono conflitti **shift-reduce** quando le due opzioni possibili sono shift e reduce; allo stesso modo si definiscono conflitti **reduce-reduce** quando le due opzioni possibili sono entrambe reduce, ma con espressioni differenti.

Esempio di shift-reduce conflict

```
...
%{
void yyerror(char *s);
%}
%union {
node_ptr node;
}
%token PLUS TIMES
%token <node> NUMBER
%type <node> expr
%start expr
%%
expr :      NUMBER |
            expr PLUS expr |
            expr TIMES expr ;
%%
...
```

Tali ambiguità possono essere risolte in vari modi; generalmente gli **shift-reduce** si risolvono aggiungendo simboli nonterminali, mentre i **reduce-reduce** si possono risolvere utilizzando un'analisi lessicale più intelligente.

Esempio di reduce-reduce conflict

```
...
%left PLUS TIMES LP RP
%token <node> NUMBER ID
%type <node> expr zeroaryfuncall unaryfuncall array
%start expr
%%
expr :            zeroaryfuncall |
                  unaryfuncall |
                  array |
                  expr PLUS expr ;

zeroaryfuncall :  ID LP RP ;
unaryfuncall :    ID LP RP ;
array :           ID LP RP ;
%%
...
```

Lo studente è invitato a provare i due esempi di cui sopra e risolvere i conflitti.

07 – F esempi

esempio 1: infix_calc
Questo è un esempio di una calcolatrice con notazione infissa, con un conflitto di **shift/reduce**. Lanciando il programma da un input "4 * 2 + 5 * 3", ci accorgiamo che prima tutti i numeri vengono spostati nello stack e convertiti in espressioni, poi la riduzione degli operatori binari viene effettuata dalla destra dello stack, e quindi l'interpretazione di quella espressione diviene "4 * (2 + (5 * 3))", diversa da ciò che vorremmo. Qui i due sorgenti:

infix_calc.flex

```
%{
#include"infix_calc.tab.h"
%}
%option noyywrap
%%
[ \t\r\f]             ;
"+"                     return(PLUS);
"*"                     return(TIMES);
[1-9][0-9]*             return(NUMBER);
.                      { printf("Not recognized.\n"); exit(1);}
%%
```

infix_calc.y

```
%{
#define YYSTYPE int
void yyerror(char *s);
%}
%token PLUS TIMES NUMBER
%%
expr          : NUMBER           |
                expr PLUS expr   |
                expr TIMES expr ;
%%
void yyerror(char *s)
{
    extern char yytext[];
    printf("At token \"%s\": %s\n", yytext, s);
}
int main() { yydebug = 1; yyparse(); }
```

Ecco i comandi da lanciare (da notare che il bison comunica la presenza di conflitti shift/reduce), o con **file di input** o con **standard input**:

```
sim@debian:~/$ bison -d -v -t -g infix_calc.y
infix_calc.y contains 4 shift/reduce conflicts.
sim@debian:~/$ flex -oinfix_calc.yy.c -l infix_calc.flex
sim@debian:~/$ gcc infix_calc.tab.c infix_calc.yy.c -o infix_calc.exe
sim@debian:~/$ ./infix_calc.exe < FILE_DI_INPUT
...

sim@debian:~/$ ./infix_calc.exe
starting parse
Entering state 0
Reading a token:
```

Vediamo il file **vcg** generato dal **bison**:

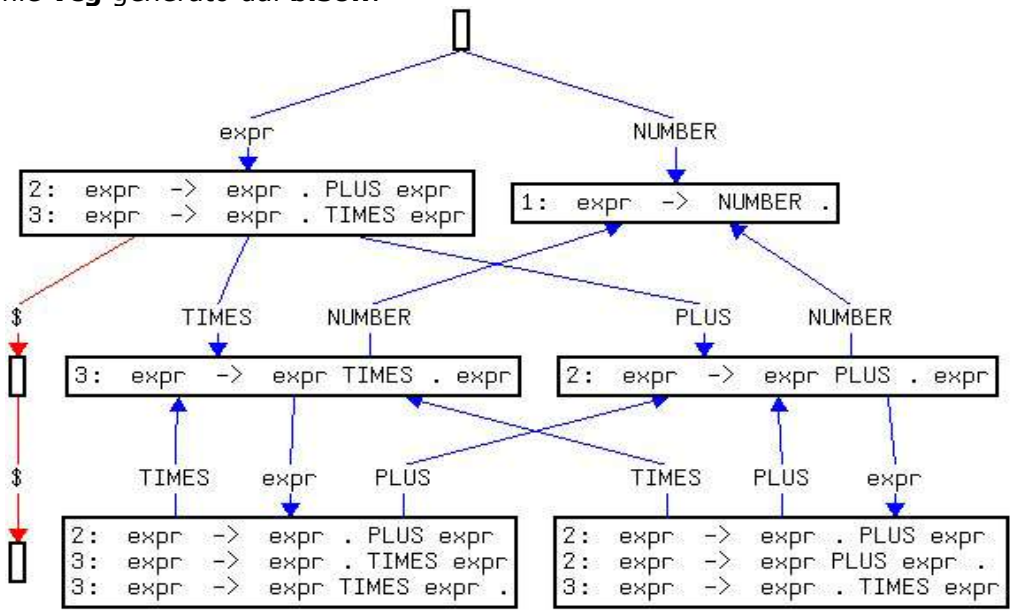

figura 07.08

Invito a verificarne il funzionamento con alcuni input, come ad esempio **4 * 2 + 5 * 3** .

esempio 2: infix_calc_2

In questo altro esempio di una calcolatrice con notazione infissa, il conflitto **shift/reduce** viene evitato, introducendo un nonterminale.
Eseguendo il programma da un input uguale a prima, "4 * 2 + 5 * 3", ci accorgiamo che il comportamento è il seguente:
- "4 * 2" viene riconosciuto come un factor F
- F viene riconosciuto come espressione E
- "5 * 3" viene riconosciuto come un factor F'
- E + F' viene riconosciuta come espressione

L'input è perciò interpretato come (4 * 2) + (5 * 3).
Il file **flex** corrispondente è identico al primo esempio (eccetto che nell'**include** il nome del file è ovviamente diverso), perciò non verrà qui ripetuto.

Vediamo il sorgente bison:

infix_calc_2.y

```
%{
#define YYSTYPE int
void yyerror(char *s);
%}
%token PLUS TIMES NUMBER
%%
expr          : expr PLUS factor  |
                factor  ;
factor        : factor TIMES NUMBER    |
                NUMBER ;
%%
void yyerror(char *s)
{   extern char yytext[];
    printf("At token \"%s\": %s\n", yytext, s);
}
int main() { yydebug = 1; yyparse(); }
```

Ecco i comandi:

```
sim@debian:~/$ bison -d -v -t -g infix_calc_2.y
sim@debian:~/$ flex -oinfix_calc_2.yy.c -l infix_calc_2.flex
sim@debian:~/$ gcc infix_calc_2.tab.c infix_calc_2.yy.c -o infix_calc_2.exe
sim@debian:~/$ ./infix_calc_2.exe < FILE_DI_INPUT
...

sim@debian:~/$ ./infix_calc_2.exe
starting parse
Entering state 0
Reading a token:
```

Vediamo, alla pagina successiva, il grafico corrispondente:

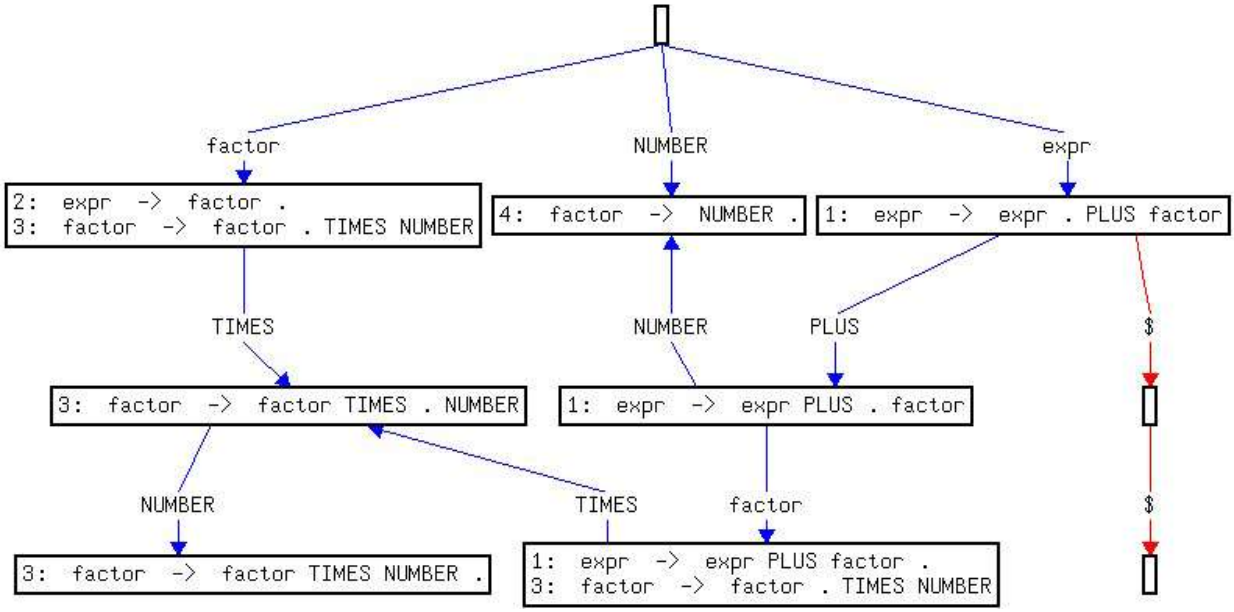

figura 07.09

esempio 3: infix_calc_3

Un altro semplice esempio di calcolatrice a notazione infissa, in cui il conflitto **shift/reduce** viene evitato usando le precedenze nel sorgente Bison.
I comandi da lanciare al prompt sono i soliti.

Come nel secondo esempio, una espressione come "4 * 2 + 5 * 3" viene correttamente interpretata come (4 * 2) + (5 * 3).

"6 + 4 + 7" viene interpretata come (6 + 4) + 7.

Ecco il sorgente bison:

infix_calc_3.y

```
%{
#define YYSTYPE int
void yyerror(char *s);
%}
%left  PLUS
%left  TIMES
%token NUMBER
%%
expr         : NUMBER           |
               expr PLUS expr   |
               expr TIMES expr ;
%%

void yyerror(char *s)
{
    extern char yytext[];
    printf("At token \"%s\": %s\n", yytext, s);
}
int main() { yydebug = 1; yyparse(); }
```

Ecco il grafico **vcg**:

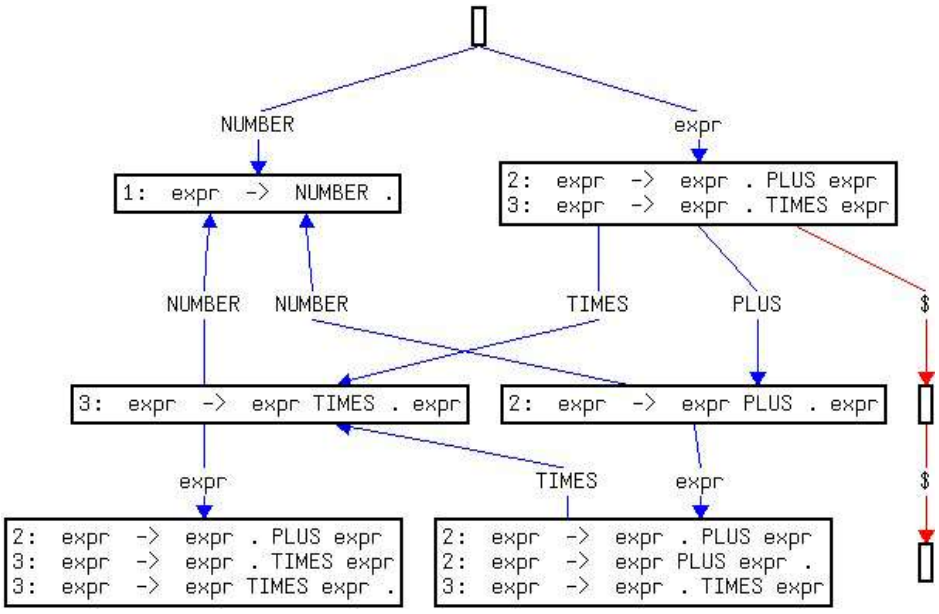

figura 07.10

esempio 4: IF-THEN-ELSE

In questo esempio vogliamo definire una grammatica che accetti stringhe contenenti il tipico costrutto **IF-THEN-ELSE** dei comuni linguaggi di programmazione.
Vediamo lo scanner:

```
%{
#include"ifthenelse.tab.h"
%}
%option noyywrap
%%
[ \n\t\r\f]              ;
"IF"                    return(IF);
"THEN"                  return(THEN);
"ELSE"                  return(ELSE);
"ISTR"                  return(ISTR);
[a-zA-Z]*               return (ID);
.                       { printf("Not recognized.\n"); exit(1);}
%%
```

E il sorgente bison:

```
%{
void yyerror(char *s);
#define YYSTYPE int
%}
%token IF THEN ELSE
%token ID ISTR
%%
stat :      IF ID THEN stat ELSE stat |
            IF ID THEN stat |
            INSTR;
%%
void yyerror(char *s)
{
    extern char yytext[];
    printf("At token \"%s\": %s\n", yytext, s);
}
int main() { yydebug = 1; yyparse(); }
```

I relativi comandi per lanciarlo:

```
sim@debian:~/$ bison -d -v -t -g ifthenelse.y
ifthenelse.y contains 1 shift/reduce conflict.
sim@debian:~/$ flex -oifthenelse.yy.c -l ifthenelse.flex
sim@debian:~/$ gcc ifthenelse.tab.c ifthenelse.yy.c -o ifthenelse.exe
sim@debian:~/$ ./ifthenelse.exe < FILE_DI_INPUT
...

sim@debian:~/$ ./ifthenelse.exe
starting parse
Entering state 0
Reading a token:
```

Ecco il grafico **vcg**:

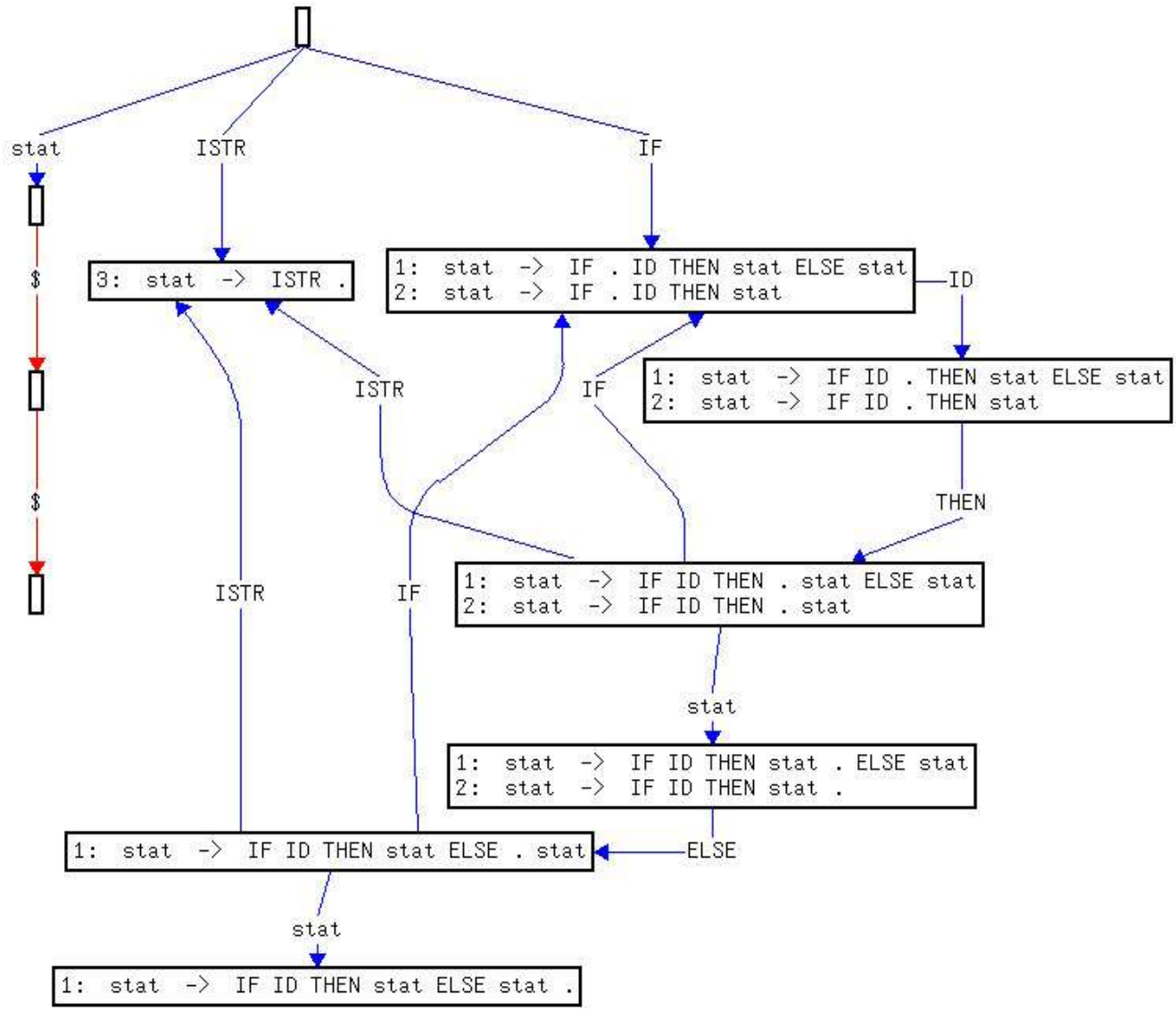

figura 07.11

esempio 5: IF-THEN-ELSE con conflitto risolto

```
%{
void yyerror(char *s);
#define YYSTYPE int
%}
%token IF THEN ELSE
%token ID INSTR
%%
stat     : balanced | unbalanced;
balanced : IF ID THEN balanced ELSE balanced |
           INSTR;
unbalanced : IF ID THEN stat |
             IF ID THEN balanced ELSE unbalanced;
%%
void yyerror(char *s)
{
    extern char yytext[];
    printf("At token \"%s\": %s\n", yytext, s);
}
int main() { yyparse(); }
```

Ecco i comandi:

```
sim@debian:~/$ bison -d -v -t -g ifthenelse_2.y
sim@debian:~/$ flex -oifthenelse_2.yy.c -l ifthenelse_2.flex
sim@debian:~/$ gcc ifthenelse_2.tab.c ifthenelse_2.yy.c -o ifthenelse_2.exe
sim@debian:~/$ ./ifthenelse_2.exe < FILE_DI_INPUT
...

sim@debian:~/$ ./ifthenelse_2.exe
starting parse
Entering state 0
Reading a token:
```

Ecco il grafico **vcg**:

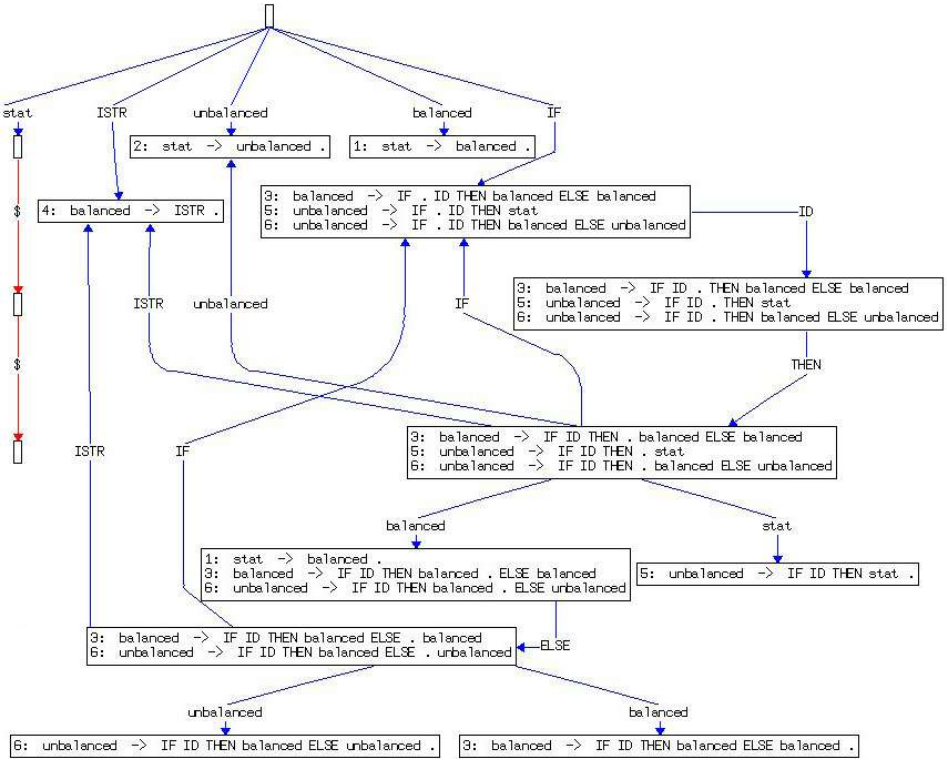

figura 07.12

Vediamo l'output generato dall'esecuzione del parser con la seguente stringa di input:

IF ciao THEN IF bye THEN ISTR ELSE ISTR ELSE ISTR

```
Starting parse
Entering state 0
Reading a token: IF
Next token is 257 (IF)
Shifting token 257 (IF), Entering state 1
Reading a token:
ciao
Next token is 260 (ID)
Shifting token 260 (ID), Entering state 5
Reading a token:
THEN
Next token is 258 (THEN)
Shifting token 258 (THEN), Entering state 6
Reading a token:
IF
Next token is 257 (IF)
Shifting token 257 (IF), Entering state 1
Reading a token:
bye
Next token is 260 (ID)
Shifting token 260 (ID), Entering state 5
Reading a token:
THEN
Next token is 258 (THEN)
Shifting token 258 (THEN), Entering state 6
Reading a token:
ISTR
Next token is 261 (ISTR)
Shifting token 261 (ISTR), Entering state 2
Reducing via rule 4 (line 10), ISTR  -> balanced
state stack now 0 1 5 6 1 5 6
Entering state 8
Reading a token:
ELSE
Next token is 259 (ELSE)
Shifting token 259 (ELSE), Entering state 9
Reading a token:
ISTR
Next token is 261 (ISTR)
Shifting token 261 (ISTR), Entering state 2
Reducing via rule 4 (line 10), ISTR  -> balanced
state stack now 0 1 5 6 1 5 6 8 9
Entering state 10
Reducing via rule 3 (line 10), IF ID THEN balanced ELSE balanced  -> balanced
state stack now 0 1 5 6
Entering state 8
Reading a token:
ELSE
Next token is 259 (ELSE)
Shifting token 259 (ELSE), Entering state 9
Reading a token:
ISTR
Next token is 261 (ISTR)
Shifting token 261 (ISTR), Entering state 2
Reducing via rule 4 (line 10), ISTR  -> balanced
state stack now 0 1 5 6 8 9
Entering state 10
Reducing via rule 3 (line 10), IF ID THEN balanced ELSE balanced  -> balanced
state stack now 0
Entering state 3
Reducing via rule 1 (line 8), balanced  -> stat
state stack now 0
Entering state 12
Reading a token:
.
non riconosciuto.
```

figura 07.13

08 – Modalità di esame

08 – A Modalità di esame

L'esame di Laboratorio di Linguaggi di Programmazione e Compilatori viene svolto secondo modalità particolari, di cui vengono qui di seguito dati i dettagli.

- L'esame è **scritto**; dopo circa tre giorni dall'esame scritto, vengono pubblicati nel sito web i risultati relativi.

- Ad alcuni studenti potrebbe venire richiesto **OBBLIGATORIAMENTE** un breve colloquio orale; in tal caso, i cognomi di questi studenti verranno evidenziati con un asterisco o simili. Sarà facoltà del docente richiedere o meno tale colloquio.

- Questi studenti, ed eventualmente quelli intenzionati a sostenere la prova orale, sono pregati di darne comunicazione al docente entro il giorno precedente alla data fissata per l'orale. Gli altri, invece, verranno nel giorno suddetto solo per registrare il voto sul libretto universitario.

- Per poter registrare il voto è necessario un voto scritto pari o superiore a 18, oppure un voto scritto pari o superiore a 15 unito ad un colloquio orale (che ovviamente dovrà permettere di raggiungere almeno la sufficienza complessiva). Tale eventualità è da considerarsi una eccezione.

- Un voto rimane valido solo per la sessione in cui è stato ottenuto, ovvero sessione estiva (giugno-luglio), sessione autunnale (settembre), sessione invernale (gennaio-febbraio).

- Per poter registrare il voto è necessario aver superato l'esame di Programmazione I.

- E' praticamente impossibile copiare.
Ogni tentativo in tal senso verrà severamente punito.

- E' NECESSARIO venire all'esame con un documento di riconoscimento VALIDO e con il libretto universitario.

- Il docente sceglierà la disposizione degli studenti nell'aula di esame, e potrà in ogni momento annullare la prova di uno studente per validi motivi.

- Alcune parti delle dispense andranno studiate in maniera generale, o addirittura non studiate affatto.
DA STUDIARE SENZA PARTICOLARE APPROFONDIMENTO (numeri di pagina):
- 19,20,21; 03-C, pag. 31,32,33;

DA NON STUDIARE:
- 03-D, 33-34; precedenza degli operatori, 75; 78-88;

Come regola generale, tutto ciò che è un riferimento storico o descrittivo contiene elementi che non verranno richiesti (ad es, la data in cui è stato inventato GREP non viene richiesta).

Per ulteriori domande sulle modalità di esame, è consigliato consultare il docente.