

GENERATORI AUTOMATICI DI PARSER. BISON

GENERATORI AUTOMATICI DI PARSER

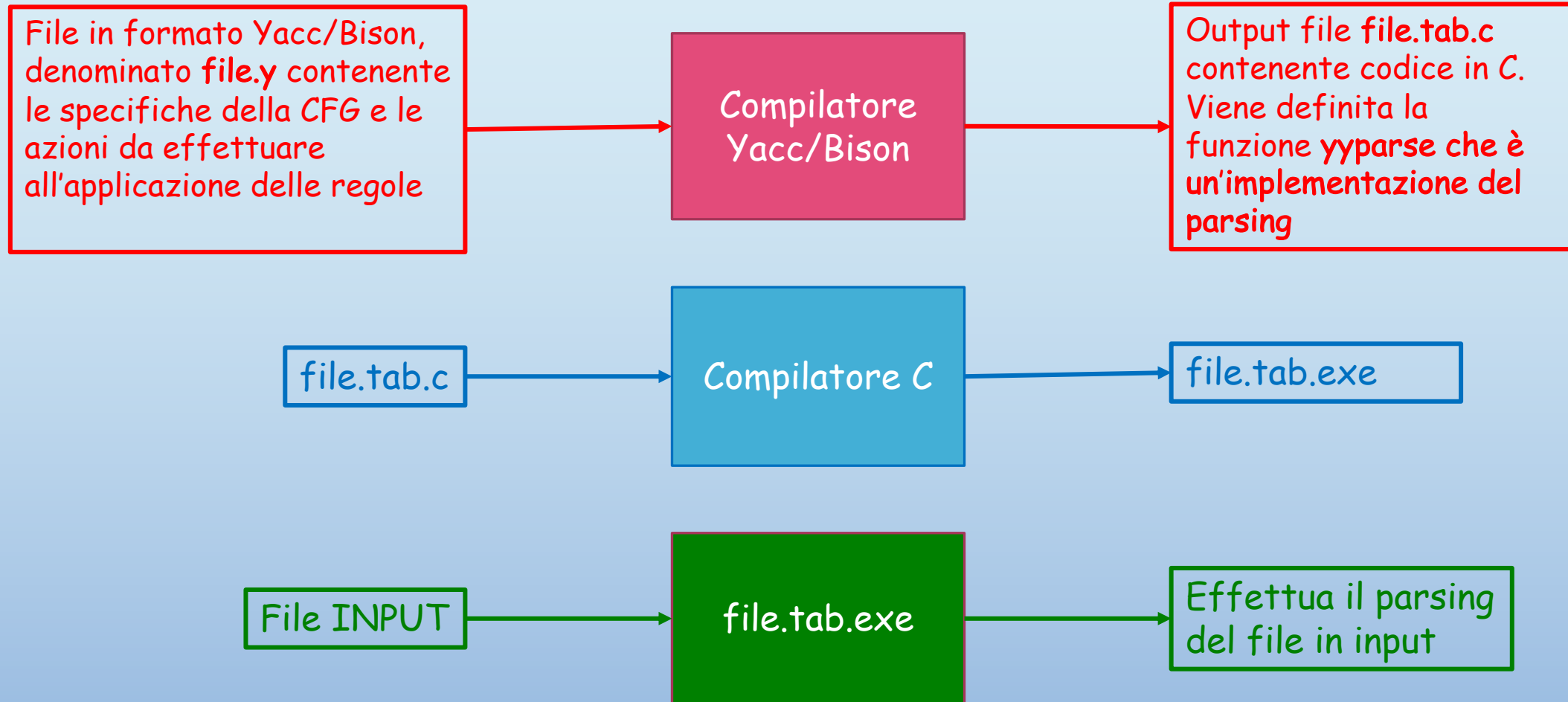
- Un generatore automatico di parser prende in input **un file che specifica la sintassi di un certo linguaggio**, espresso nella forma delle regole di una grammatica CF, includendo altre **funzioni ausiliarie, definizioni di Token, etc**, e produce in **output** un codice (scritto in un certo linguaggio) che **implementa il ruolo del parser**
- Erano chiamati compilatori di compilatori poiché tradizionalmente tutti gli step della compilazione venivano realizzate dal parser. Adesso il parser è considerato solo uno step del processo.
- **Yacc** (**yet another compiler-compiler**) è un generatore largamente usato che incorpora la tecnica del **parsing LALR(1)**.
- Una delle implementazioni più usate di yacc, che è open source, è **BISON**

BISON

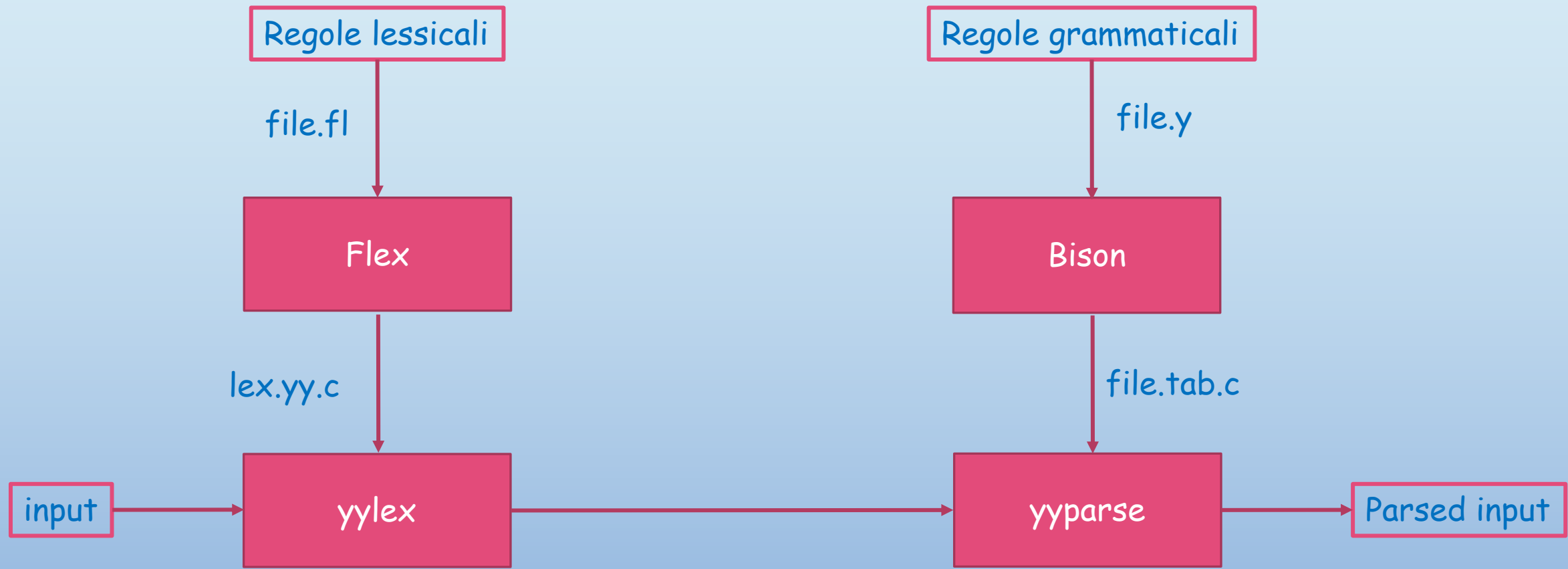
Il linguaggio deve essere descritto mediante una grammatica context-free.

In particolare Bison è ottimizzato per grammatiche **LALR(1)**, ma con dichiarazioni opportune, è anche capace di fare il parsing per una classe più estesa di grammatiche.

USO DI YACC E BISON



USO CONGIUNTO DI FLEX E BISON



FORMA DI UN PROGRAMMA IN BISON

Un file in formato BISON, come per i file flex, consiste di 3 sezioni, separate da %%:

Definizioni

%%

Regole della grammatica

%%

Funzioni ausiliarie

La routine `yyparse` ha bisogno di altre funzioni:

1. L'analizzatore lessicale (`yylex`) che può essere scritto a mano o prodotto da flex
2. La funzione che riporta gli errori (`yyerror`)
3. La funzione main deve contenere la chiamata `yyparse`

SINTASSI E CONVENZIONI

Un **simbolo non terminale** della grammatica è rappresentato nell'input di BISON come un identificatore in C. Per convenzione viene scritto **minuscolo** come **expr**, **stmt** o **declaration**.

La rappresentazione Bison per un **simbolo terminale** è anche chiamata un **token type**. Token types possono essere pure rappresentati come gli identificatori in C. Per convenzione, questi identificatori dovrebbero essere scritti in **maiuscolo** per distinguerli da non terminali: ad esempio, **INTEGER**, **IDENTIFIER**, **IF** o **RETURN**.

Il simbolo terminale **error** è riservato per il recupero degli errori.

Un simbolo **terminale** può essere anche rappresentato mediante un **carattere letterale**. Conviene fare questo ogni volta che un token è un singolo carattere (parentesi, il segno '+', etc.): si usi quello stesso carattere in un letterale come il simbolo terminale per quel token.

1. LA SEZIONE DICHIARAZIONI

La sezione dichiarativa può contenere del codice in C tra %{ e %} e può contenere variabili locali, regole di inclusione di librerie etc.

```
%{  
#include <stdio.h>  
int regs[26];  
int base;  
%}
```

L'espressione **%start** seguita da un non terminale rappresenta
l'assioma

%start espr

LA SEZIONE DICHIARAZIONI-TOKEN

Il modo base per dichiarare il nome di un tipo di token (simbolo del terminale) è il seguente:

%token nome_token

BISON lo convertirà in una direttiva `#define` nel parser, in modo che la funzione `yylex` (se è in questo file) può usare il nome per indicare il codice (intero) di questo tipo di token.

In alternativa a **%token**, che viene usato per token generici, possono usare **%left**, **%right**, **%precedence** o **%nonassoc** se il token considerato è un **operatore** e si desidera specificare anche l'associatività e la precedenza.

%left (**%right**, risp) specifica l'associatività a sinistra (destra, risp)

%nonassoc specifica nessuna associatività (**x op y op z** è considerato un errore)

%precedence, permette di definire solo precedenze ma nessuna associatività.

GESTIONE DELLE PRECEDENZE

Un modo implicito di stabilire le precedenze degli operatori è quella di dichiarare i token degli operatori in un certo ordine. La prima dichiarazione di precedenza/associatività nel file dichiara gli operatori la cui precedenza è più bassa, la successiva dichiarazione di questo tipo dichiara gli operatori di cui la precedenza è un po' più alta, e così via. L'ordine delle precedenze viene stabilita dalla più bassa alla più alta in base all'ordine in cui vengono definiti i token.

ESEMPIO

```
%left '<' '>' '=' '!=' '<=' '>='
```

```
%left '+' '-'
```

```
%left '*' '/'
```

2. REGOLE GRAMMATICALI

Il ruolo fondamentale del parser è convertire le regole grammaticali nell'automa a pila corrispondente. La regola grammaticale è $X \rightarrow X_1 X_2 \dots X_n$, viene espressa mediante

$$X: X_1 X_2 \dots X_n;$$

ESEMPIO. la regola di BISON per lo statement **return** in C è la seguente:

```
stmt: RETURN expr ';' ;
```

RETURN è un simbolo terminale, **expr** è un non terminale. Nota che il punto e virgola tra virgolette **' ; '** è un token (carattere terminale) rappresentato come carattere letterale, che rappresenta parte della sintassi C per lo statement, mentre il punto e virgola nudo alla fine (**;**) e i due punti dopo **stmt** (**:**) sono punteggiatura in BISON utilizzata in ogni regola.

FORMATO DELLE REGOLE GRAMMATICALI

Per esempio,

```
exp: exp '+' exp;
```

dice che due gruppi di tipo **exp**, con un token '+' in mezzo, possono essere combinati in un raggruppamento più ampio di tipo exp.

Si dice che una regola è **vuota** se il suo lato destro (componenti) è vuoto. Significa quel risultato può abbinare la stringa vuota.

ESEMPIO: il punto e virgola opzionale

```
semicolon.opt: | ';' ;
```

AZIONI E VALORI SEMANTICI

Una volta stabilita una **regola sintattica** mediante le componenti, è possibile associare a questa regola un'azione che determina la **semantica della regola**. Un'azione si presenta nella forma:

{Codice in C}

Il proposito delle azioni è spesso quello di **calcolare il valore semantico** di un certo costrutto. Tale valore viene calcolato come nel seguente

ESEMPIO

expr : expr '+' expr { \$\$ = \$1 + \$3; }

Significa che quando viene riconosciuta la regola grammaticale **expr → expr + expr** si associa al lato sinistro della regola il valore ottenuto dalla somma del valore della prima componente della parte destra della regola (\$1) e della terza componente della parte destra della regola (\$3). Il simbolo terminale '+' viene «interpretato» come l'operatore aritmetico +

3. LA SEZIONE FUNZIONI AUSILIARIE

Nella terza sezione è possibile inserire delle funzioni ausiliarie per il parser, che vengono copiate verbatim nel file c generato da bison

YYLEX, YYLVAL, YYSTYPE

Il valore restituito da **yylex** deve essere il **codice numerico positivo** per il tipo di token che ha appena trovato; un valore zero o negativo indica la fine dell'input.

Quando si fa riferimento a un token con il suo nome nelle regole grammaticali, tale nome nel file di implementazione del parser diventa una macro C la cui definizione è il codice numerico assegnato a quel tipo di token. Quindi **yylex** può usare il nome per indicare quel tipo di token.

Ogni token potrebbe portare con se il corrispondente valore (eventualmente memorizzato nella variabile **yyval**, che contiene il valore semantico di **yylex** e il suo tipo è **yytype**)

ESEMPIO

```
%{
#include <stdio.H>
int regs[26];
int base;
%}

%start espr
%token DIGIT LETTER
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS

%%

espr: /*empty */
| espr istr digit
| espr error letter {funzione()};

%%

int main(void) { ... }
```

Tra %{ e %} si trovano inclusione di librerie, definizioni di tipi, variabili usate nelle azioni, direttive, prototipi di funzioni definite in seguito. Nella sezione definizioni sono inclusi inoltre nomi dei simboli terminali (token), regole di precedenza degli operatori, tipi di dato dei valori semantici

%start serve per definire l'assioma

%token descrive i nomi dei token

%left denota una priorità dell'operando a sinistra

Le regole di precedenza degli operatori sono in ordine inverso rispetto all'ordine in cui appaiono nella sezione dichiarazioni.

Regole della grammatica ed eventuali azioni (rappresentate da una funzione)

Routine ausiliarie

CREAZIONE DEL FILE ESEGUIBILE

Per convertire un file della grammatica in un file di implementazione del parser si usa il comando

```
bison file.y
```

BISON produce un file di implementazione del parser dal nome `file.tab.c`. Il file di implementazione del parser contiene il codice sorgente per la procedura `yyparse`.

Le funzioni aggiuntive del file della grammatica (`yylex`, `yyerror` e `main`) sono copiate parola per parola nel file di implementazione del parser.

Infine il `file.tab.c` va compilato con un compilatore C per ottenere l'eseguibile.

```
gcc file.tab.c
```

INTERAZIONE DELL'ANALIZZATORE SINTATTICO CON L'ANALIZZATORE LESSICALE

Com'è noto, la funzione analizzatore lessicale, **yylex**, riconosce i token presenti nel file di input e li restituisce al parser.

BISON non crea automaticamente questa funzione, quindi deve essere scritta separatamente, in modo che **yyparse** possa chiamarla. Nel nostro caso la genereremo mediante la compilazione in **FLEX** di un file.fl.

Se **yylex** è definito in un file sorgente separato, è necessario disporre le definizioni macro di tipo token affinché siano disponibili. Per fare ciò, occorre utilizzare l'opzione **-d** quando si esegue **BISON**, in maniera tale che queste definizioni macro saranno scritte in un **parser header file separato**, **file.tab.h**, che può essere incluso in tutti i file sorgente che ne hanno bisogno (fra cui il file flex). L'istruzione:

bison file.y -d produce in output entrambi i file **file.tab.h** e **file.tab.c**

ISTRUZIONI PER LA COMPILAZIONE

1. Generare un file di testo in formato BISON (sezione dichiarazioni, sezione regole grammaticali, sezione funzioni ausiliari). Salvare possibilmente con l'estensione .y
2. Compilare il file in BISON col comando

```
bison nomefile.y
```

Per creare il file `nomefile.tab.c` oppure

```
bison nomefile.y -d
```

Per creare il file `nomefile.tab.c` e il file `nomefile.tab.h`

il file `nomefile.tab.h` va incluso nel file flex con la direttiva `#include`

3. Compilare con un compilatore C il file nomefile.tab.c insieme con il file lex.yy.c ottenuto dal generatore di analizzatori lessicali (FLEX)

```
gcc lex.yy.c nomefile.tab.c
```

Dando luogo al file eseguibile `a.exe` oppure

```
gcc lex.yy.c nomefile.tab.c -o nome_output
```

Per creare il file eseguibile `nome_output.exe`

ESERCIZIO

Espressioni aritmetiche: scrivere un parser bottom-up (ed un analizzatore lessicale) per la seguente grammatica per il riconoscimento di espressioni aritmetiche:

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow \text{numero}$

$F \rightarrow (E)$

Dove **numero** è un intero senza segno;

ANALIZZATORE LESSICALE (TOKENIZZAZIONE)

$E \rightarrow E + T$
 $E \rightarrow E - T$
 $E \rightarrow T$
 $T \rightarrow T * F$
 $T \rightarrow T / F$
 $T \rightarrow F$
 $F \rightarrow \text{numero}$
 $F \rightarrow (E)$

E' il file creato dal parser, mediante la direttiva -d che contiene l'associazione tra token e codice

Elimina gli spazi

Restituisce all'analizzatore lessicale il token appena lo trova. L'analizzatore sintattico richiamerà l'analizzatore lessicale per la richiesta del prossimo token

```
%{  
#include <stdio.h>  
#include "espr.tab.h"  
%}  
/* regular definitions  
*/  
delim [ \t\n]  
ws {delim}+  
digit [0-9]  
number {digit}+  
%option noyywrap  
%%  
{ws} ;  
{number} {return(NUM);}  
\+ {return(PIU);}  
\- {return(MENO);}  
\* {return(PER);}  
[/] {return(DIVISO);}  
\( {return(PAR_AP);}  
\) {return(PAR_CH);}  
%%
```

Permette al programmatore di inserire un messaggio personalizzato di errore invece del semplice «syntax error»

Token riconosciuti (e restituiti) dall'analizzatore lessicale

```
%{
#include <stdio.h>
}%
%token NUM PIU MENO PER DIVISO
PAR_AP PAR_CH
%start Expr
%error-verbose
%%
Expr: Expr PIU Term
    | Expr MENO Term
    | Term
    ;
Term: Term PER Factor
    | Term DIVISO Factor
    | Factor
    ;
```

ANALIZZATORE SINTATTICO

```
Factor: NUM
| PAR_AP Expr PAR_CH
;
%%
int main(void)
{
    yyparse();
    return 0;
}
yyerror (char *s)
{
    printf ("Errore Sintattico\n");
}
```

ESERCIZI

1. Scrivere un programma che verifichi la correttezza sintattica di un' espressione aritmetica (esercizio precedente)
2. Scrivere un programma che, oltre a verificare la correttezza sintattica, calcoli il valore dell'espressione aritmetica;

3. Scrivere un programma che valuti una lista di espressioni aritmetiche separate da \n

PER CASA

4. Scrivere un programma in bison che valuti un'espressione in forma postfissa.
5. Scrivere un programma in bison che valuti una lista di espressioni in forma postfissa separate da \n.
6. Scrivere un programma in bison che converta in notazione postfissa, le espressioni in forma infissa corretta.
7. Scrivere un programma in bison che converta in notazione infissa, le espressioni in forma postfissa corretta.


SOLUZIONE ESERCIZIO 2- ANALIZZATORE LESSICALE

```
%{
#include <stdio.h>
#include "espr_val.tab.h"
%}
/* Regular definitions */
delim [ \t\n]
ws {delim}+
digit [0-9]
number {digit}+
%option noyywrap
%%

{ws} ;
{number} {yylval=atoi(yytext) ; return(NUM) ;}
\+ {return(PIU) ;}
\- {return(MENO) ;}
\* {return(PER) ;}
[/] {return(DIVISO) ;}
\( {return(PAR_AP) ;}
\) {return(PAR_CH) ;}
%%
```

Scrivere un programma
che, oltre a verificare la
correttezza sintattica,
calcoli il valore
dell'espressione aritmetica;

Attribuisce alla variabile
numerica `yylval` la conversione in
numero intero del lessema (in
formato testo) riconosciuto, che
corrisponde al suo valore
semantico, che ci serve ai fini
della valutazione delle
espressioni



Attribuzione valori semantici

```
%{
#include <stdio.h>
%}
%token NUM PIU MENO PER DIVISO PAR_AP PAR_CH
%start Input
%error-verbose
%%
Input: Expr {printf("Risultato: %d\n", $1);}
;
Expr: Expr PIU Term {$$=$1+$3;}
| Expr MENO Term {$$=$1-$3;}
| Term {$$=$1;}
;
Term: Term PER Factor {$$=$1*$3;}
| Term DIVISO Factor {$$=$1/$3;}
| Factor {$$=$1;}
;
Factor: NUM {$$=$1;}
| PAR_AP Expr PAR_CH {$$=$2;}
;
```

ESERCIZIO 2 ANALIZZATORE SINTATTICO

```
%%
int main()
{if (yyparse()==0)
printf("Espressione corretta");
return (0);
}
yyerror (char *s)
{
printf ("Errore Sintattico\n");
}
```

Queste istruzioni possono anche mancare. In loro assenza il valore di default è il valore semantico attribuito all'unica variabile

ESERCIZIO 3

ANALIZZATORE LESSICALE

```
%{
#include <stdio.h>
#include "espressioni_val.tab.h"
%}
/* regular definitions */
delim  [ \t]
ws      {delim}+
digit  [0-9]
number {digit}+
%option noyywrap
%%
{ws}      ;
{number}   {yylval=atoi(yytext); return(NUM); }
\+        { return(PIU); }
\*        { return(PER); }
\(        { return(PAR_AP); }
\)        { return(PAR_CH); }
\n        { return(NEWL); }
%%
```

Scrivere un programma che valuti una lista di espressioni aritmetiche separate da \n (tutte le operazioni aritmetiche)

```

%{
#include <stdio.h>
%}

%token NUM PAR_AP PAR_CH NEWL
%left PIU
%left PER
%start Input
%error-verbose
%%

Input:          /* epsilon*/
    | Input Line ;

Line: NEWL
    | Expr {printf("Risultato: %d\n", $1);} ;

Expr: Expr PIU Expr      {$$=$1+$3;}
    | Expr PER Expr      {$$=$1*$3;}
    | PAR_AP Expr PAR_CH {$$=$2;}
    | NUM ;

%%

```

un input completo è o una stringa vuota, oppure un input completo seguito da una linea di input (definizione ricorsiva)

ESERCIZIO 3 ANALIZZATORE SINTATTICO

```

main()
{
    yyparse();
}

yyerror(char *s)
{
    printf("Errore Sintattico\n");
}

```

COMPLETO ANALIZZATORE LESSICALE

Scrivere un programma in Bison che valuti un elenco di espressioni aritmetiche separate da newline. (tutte le operazioni +, -, *, /)

```
%{
#include <stdio.h>
#include "espressioni.tab.h"
%}
/* Regular definitions */
delim [ \t]
ws {delim}+
digit [0-9]
number {digit}+
%option noyywrap
%%
{ws} ;
{number} {yylval=atoi(yytext); return(NUM);}
\+ {Return(PIU);}
\- {Return(MENO);}
\* {Return(PER); }
[/] {return(DIVISO); }
\( {return(PAR_AP); }
\) {return(PAR_CH); }
\n {return(NEWL); }
%%
```

Scanner

```
%{
#include <stdio.h>
%}

%token NUM PAR_AP PAR_CH NEWL
%left PIU MENO
%left PER DIVISO
%start Input
%error-verbose
%%

Input:          /* epsilon*/
    | Input Line
    ;

Line: NEWL
    | Expr {printf("Risultato: %d\n",
$1);}
    ;

Expr: Expr PIU Term {$$=$1+$3;}
    | Expr MENO Term {$$=$1-$3;}
    | Term {$$=$1;}
```

```
Term: Term PER Factor {$$=$1*$3;}
    | Term DIVISO Factor {$$=$1/$3;}
    | Factor {$$=$1;}
    ;

Factor: PAR_AP Expr PAR_CH {$$=$2;}
    | NUM {$$=$1}
    ;

%%

int main(void)
{
    yyparse();
}

yyerror(char *s)
{
    printf("Errore Sintattico\n");
}
```

ESERCIZI

1. Scrivere un programma che testi se una parola binaria sull'alfabeto $\{a,b\}$ è palindroma
2. Scrivere un programma che effettui il test di palindromia su una data una lista di stringhe binarie separate da `\n`

PROBLEMA

La grammatica più semplice che riconosce le palindrome su un alfabeto di due lettere è la seguente:

$P \rightarrow \varepsilon \mid a \mid b \mid aPa \mid bPb$

Che è una **grammatica ambigua e non LALR(1)**. E' possibile applicare ugualmente BISON utilizzando le seguenti direttive nella sezione definizioni

```
%glr-parser  
%expect n  
%expect-rr m
```

Dove **glr** sta per **generalized LR parser**. Quando un parser glr incontra un conflitto, concettualmente divide il calcolo secondo le due scelte e continua le due possibili strade in parallelo. Se una delle scelte porta a un parsing la stringa è riconosciuta

%expect n significa che il parser «si aspetta» n conflitti shift-reduce

%expect-rr m significa che «si aspetta» m conflitti reduce-reduce.

Evitano che vengano prodotti dei messaggi di warning riguardo ai conflitti riconosciuti

Scanner

```
%{
#include <stdio.h>
#include "palindrome.tab.h"
}%
/* regular
definitions*/
delim [ \t]
ws {delim}+
%option noyywrap
%%
{ws} ;
a {return(A);}
b {return(B);}
\n {return(NEWL);}
. {printf("Token non riconosciuto\n");}
%%
```

Generalised LR

Aspetta due conflitti
reduce/reduce

Parser

```
%{
#include <stdio.h>
}%
%token A B NEWL
%start input
%error-verbose
%glr-parser
%expect 4
%expect-rr 2
%%
input: /*vuota*/
      | pal ;
pal:   expr NEWL {Printf("palindroma\n");};
expr:  A expr A |
      B expr B |
      A |
      B |
      /*vuota*/ ;
%%
int main(void)
{
  yyparse();
  return 0;
}
yyerror (char *s)
{
  printf ("Non Palindroma\n");
}
```

Aspetta quattro
conflitti shift/reduce

Grammatica:

programma -> program identificatore ";" istruzione

istruzione -> begin istruzione listaistruzioni |
 write espressione |
 read identificatore |
 if test then istruzione else istruzione |
 while test do istruzione |
 identificatore ":=" espressione

listaistruzioni -> end |
 ";" istruzione listaistruzioni

espressione -> costante |
 costante op espressione

costante -> numero |
 identificatore

test -> espressione relop espressione

Token descritti da espressioni regolari:

op : "+" | "-" | "*" | "/"

relop : "<" | "<=" | ">" | ">=" | "=" | "<>"

identificatore : lettera (lettera | cifra)*

numero : cifra+

lettera : "A" | ... | "Z" | "a" | ... | "z"

cifra : "0" | ... | "9"

COMPILATORE PASCAL

Scrivere un analizzatore sintattico per il linguaggio simplepas la cui sintassi è descritta a fianco. Le parti scritte in grassetto rappresentano le parole chiave.

Sono inoltre previsti i caratteri spazio, nuova linea, e tabulazione come separatori, e commenti racchiusi tra parentesi graffe.

```

/*
 * Analizzatore lessicale
 */

%{
#include <stdio.h>
#include "pas.tab.h"
%}

/* regular definitions */

delim      [ \t\n]
ws         {delim}+
letter     [a-zA-Z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+
%option noyywrap

```

```

%%
{ws}                ; //elimina gli spazi
\[^[^\\]*\\]        ; //elimina i commenti
(PROGRAM|program)   { return(PROGRAM); }
(BEGIN|begin)       { return(BEGIN); }
(END|end)           { return(END); }
(IF|if)             { return(IF); }
(THEN|then)         { return(THEN); }
(ELSE|else)         { return(ELSE); }
(READ|read)         { return(READ); }
(WRITE|write)       { return(WRITE); }
(WHILE|while)       { return(WHILE); }
(DO|do)             { return(DO); }
\;                 { return(SEQOP); }
{number}            { return(NUM); }
" :="              { return(ASSOP); }
[ \+ \- \* /]       { return(ARITOP); }
("=" | ">" | "<" | "<=" | ">=" | "<>") { return(RELOP); }
{id}                { return(ID); }
%%

```

Parser

```
%{
#include <stdio.h>
int errore=0;
%}
%token PROGRAM IF THEN ELSE BEGIN END READ WRITE WHILE DO SEQOP ASSOP
ID RELOP ARITOP NUM
%start inizio /* Assioma della grammatica */
%error-verbose
%%
inizio: PROGRAM ID SEQOP istruz
;
istruz:  BEGIN istruz rest_istruz
        |  ID ASSOP expr
        |  IF test THEN istruz ELSE istruz
        |  READ ID
        |  WRITE expr
        |  WHILE test DO istruz
;
rest_istruz: SEQOP istruz rest_istruz
            |  END
;
;
```

```
test: expr RELOP expr
;
expr: constant
    | constant ARITOP expr
;
constant: NUM
    | ID
;
%%
main()
{
    yyparse();
    if (!errore) printf("Programma corretto!!!\n");
}
yyerror (char *s) /* Gestisce la presenza di errori sintattici */
{
    printf ("SYNTAX ERROR %s\n",s);
    errore=1;
}
```

ATTRIBUTI MULTITYPE

In un parser di Bison, ogni simbolo, sia token che non terminale, può avere un valore associato ad esso. Per default, i valori sono tutti **numeri interi**, ma i programmi generalmente richiedono valori di vario tipo. Il costrutto **%union** è usato per creare una dichiarazione di unione del linguaggio C per i valori dei simboli, per specificare l'intera collezione di tipi e permette di scegliere di volta in volta il tipo più utile per i token (con **%token**) e con i simboli non terminali (con **%type**). Per esempio:

```
%union {  
int intero;  
float reale;  
}
```

Funziona come l'union in C, ovvero le due definizioni condividono la stessa area di memoria e sono alternative una all'altra.

Per indicare in flex un attributo di tipo intero si userà **yylval.intero**, altrimenti **yylval.reale**

ESEMPIO

```
%{  
# include <stdio.h>  
# include <stdlib.h>  
# include "fb3-1.h"  
%}  
  
%union {  
  struct ast *a;  
  double d;  
}  
  
/* declare tokens */  
%token <d> NUMBER  
%type <a> exp factor term
```

Attribuzione di un tipo al
token

attribuzione di un tipo ad un non
terminale

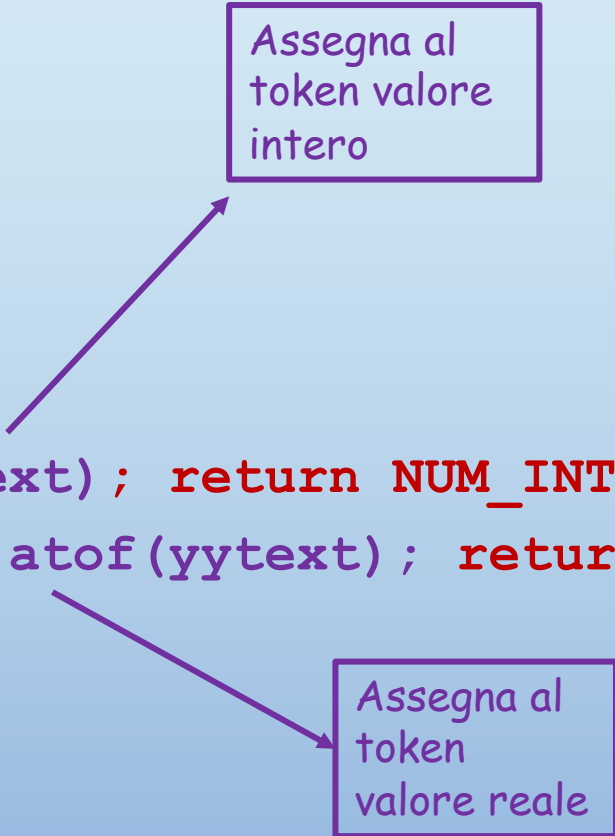
ESEMPIO

Scrivere un programma che valuta le espressioni aritmetiche a valori reali e interi.

Cosa cambia rispetto a quello che calcolava espressioni aritmetiche solo con numeri interi?

COSA CAMBIA NELLO SCANNER (ESEMPIO)

```
%{  
#include "eseunion.tab.h"  
%}  
%option noyywrap  
%%  
[ \t]+ ;  
[0-9]+ {yyval.intero=atoi(yytext); return NUM_INT;}  
[0-9]+ "." [0-9]* {yyval.reale = atof(yytext); return  
NUM_REAL;}  
.|\\n {return yytext[0];}  
%%
```



Assegna al token valore intero

Assegna al token valore reale

COSA CAMBIA NEL PARSER (ESEMPIO)

```
%{  
#include <stdio.h>  
void yyerror(char*);  
%}  
%union {  
int intero;  
float reale;  
}  
%left '+'  
%left '*'  
%token <intero> NUM_INT  
%token <reale> NUM_REAL  
%type <reale> exp  
%start input  
%%  
input: /* empty */  
| exp '\n' {printf("valore=%f\n", $1);}  
;
```

Gli interi sono
convertiti in float

```
exp : exp '+' exp {$$=$1+$3;}  
| exp '*' exp {$$=$1*$3;}  
| NUM_INT {$$=(float)$1;}  
| NUM_REAL {$$=$1;}  
;  
%%  
int main(void) {  
yyparse();  
return 0;  
}  
void yyerror(char* s) {  
fprintf(stderr, "%s\n", s);  
}
```

Nel caso in cui lo scanner riconosce
un intero, il token restituito è
NUM_INT, nel caso in cui riconosce
un reale, il token è NUM_REAL

Le espressioni (simbolo non terminale) hanno
comunque valori reali