Linguaggi formali e compilazione

Corso di Laurea in Informatica

A.A. 2014/2015

Linguaggi formali e compilazione

Strumenti di sviluppo per scanner e parser

- Il lexical analyzer Lex
- Uso di Yacc/Bison

Che cosa è Lex

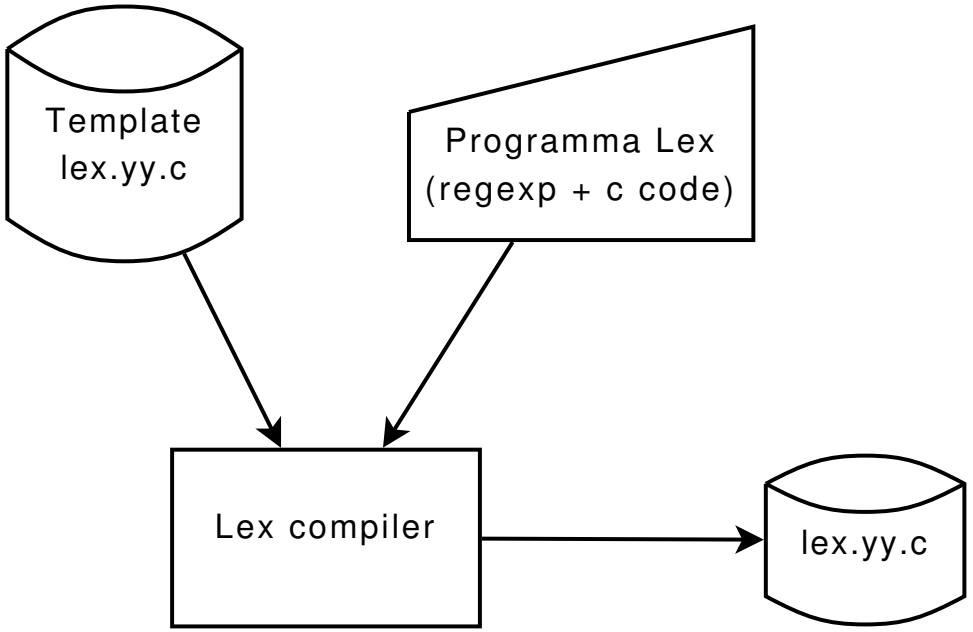
- *Lex* è un *generatore di scanner*.
- Si tratta cioè di un software in grado di generare automaticamente un altro programma che riconosce stringhe di un linguaggio regolare.
- Non solo, il software generato da Lex ha “capacità di scanning”, cioè di acquisire le stringhe da analizzare in sequenza (da file o standard input) e di passare l’output ad un altro programma, tipicamente un parser.
- Lex può quindi essere uno strumento prezioso nella realizzazione di un compilatore.

Come funziona Lex

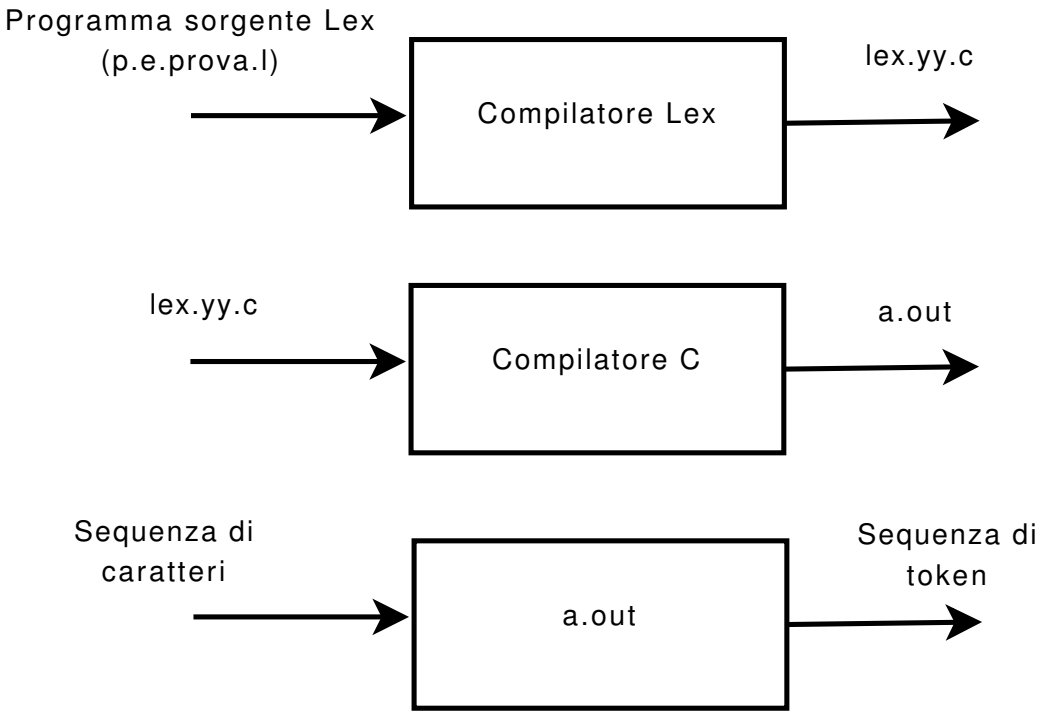
- L'input per un programma Lex è costituito "essenzialmente" da un insieme di espressioni regolari/pattern da riconoscere.
- Inoltre, ad ogni espressione regolare $\mathcal{E}$, l'utente associa un'*azione*, espressa sotto forma di codice C.
- Lex "trasforma" $\mathcal{E}$ nella descrizione di un "automa" che riconosce il linguaggio descritto da $\mathcal{E}$.
- Lex inoltre associa, ad ogni stato terminale dell'automa che riconosce $\mathcal{E}$, il corrispondente software fornito dall'utente.
- Nel programma generato da Lex, tale software andrà in esecuzione quando viene riconosciuto un lessema di $\mathcal{E}$.

Come funziona Lex (2)

- Lex inserisce la descrizione dell'automa e il codice fornito dall'utente in uno scheletro di programma (*template*) per ottenere così il programma finale "completo", per default chiamato `yy.lex.c`.
- La parte più "complessa" dal punto di vista teorico consiste proprio nella trasformazione delle espressioni regolari in automi.
- Ma noi sappiamo già "che cosa ci sta sotto"!



Schema d'uso di Lex



Un primo programma Lex

- Programma Lex per riconoscere (un sottoinsieme de) i token del linguaggio Pascal

```
delim [ \t\n]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id {letter}({letter}|{digit})*
number [+-]?{digit}+(\.{digit}+)?(E[+-]?{digit}+)?
%%
{ws} { }
if {printf("IF ");}
then {printf("THEN ");}
else {printf("ELSE ");}
{id} {printf("ID ");}
{number} {printf("NUMBER ");}
"<" {printf("RELOP ");}
"<=" {printf("RELOP ");}
"=" {printf("RELOP ");}
"<>" {printf("RELOP ");}
">" {printf("RELOP ");}
">=" {printf("RELOP ");}
":=" {printf("ASSIGNMENT ");}
%%
main()
{ yylex();
  printf("\n"); }
```

Esecuzione del programma

- Se mandiamo in esecuzione il programma C ottenuto dopo le due compilazioni (si rammenti lo schema d'uso) con il seguente input:

```
if var1<0.0 then
    sign := -1
    else sign := 1
```

otteniamo come output la sequenza di token

IF ID RELOP NUMBER THEN ID ASSIGNMENT NUMBER ELSE ID ASSIGNMENT NUMBER

in realtà poiché i token name vengono internamente rappresentati mediante interi, la sequenza le output potrebbe essere:

12 2 6 1 13 2 18 1 14 2 18 1

Struttura generale di un programma Lex

- Un generico programma Lex contiene tre sezioni, separate dal dalla sequenza %%

```
Dichiarazioni
%%
Regole di traduzione
%%
Funzioni ausiliarie
```

- La sezione “Dichiarazioni” può contenere definizione di costanti e/o variabili, oltre alle cosiddette *definzioni regolari*, cioè espressioni che consentono di “dare un nome” ad espressioni regolari.
- La sezione più importante è quella relativa alle regole di traduzione che contiene le descrizioni dei pattern da riconoscere e, corrispondentemente, le azioni che devono essere eseguite dallo scanner.

Struttura generale di un programma Lex (continua)

- L'ultima sezione può contenere funzioni aggiuntive (che vengono tipicamente invocate nella parte relativa alle regole di traduzione).
- Se lo scanner non è utilizzato come routine del parser o di altro programma, quest'ultima sezione contiene anche il main program.
- Se presente, il main conterrà ovviamente la chiamata allo scanner (funzione `yylex`).

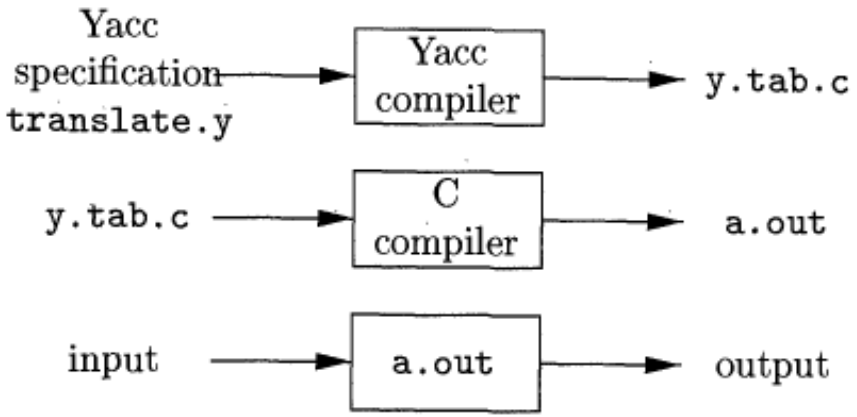
Non solo riconoscimento di token

- Il seguente programma Lex conta caratteri, parole e linee presenti in un file (assimiglia dunque al programma `wc` di Unix/Linux).

```
%{
unsigned charCount = 0, wordCount = 0, lineCount = 0;
%}

word [^ \t\n]+
eol \n

%%

{word}      {wordCount++; charCount += yyleng; }
{eol}       {charCount++; lineCount++}
.           charCount++;

%%
main()
{ yylex();
  printf("%d %d %d\n", lineCount, wordCount, charCount);
```

Generare parser con Yacc

- *Yacc* (*Yet another compiler-compiler*) è uno strumento per generare parser (del tipo *LR*, che studieremo in seguito) per (un sottoinsieme di) grammatiche context-free o, più generalmente, per implementare il front-end di un compilatore.
- Il seguente schema (tratto dal libro di Aho et al. citato nella pagina web del corso) mostra il classico uso di Yacc nella realizzazione di un parser/traduttore.



Struttura generale di un programma Yacc

- È simile alla struttura dei programmi Lex e contiene, in generale, tre sezioni, separate dalla sequenza `%%`:

```
Sezione delle definizioni
%%
Sezione delle regole grammaticali
%%
Sezione dei sottoprogrammi utente
```

  - Nella prima sezione si trovano le dichiarazioni dei token presenti nelle produzioni e le dichiarazioni dei simboli utilizzati nelle porzioni di codice C specificate nel programma (racchiuse fra `%{` e `%}`).
  - Nella seconda sezione vengono elencate le produzioni della grammatica e le cosiddette *regole di traduzione*, che sono di fatto frammenti di codice C;
  - Nell'ultima sezione si trova ancora codice C che verrà copiato "verbatim" nel programma C prodotto da Yacc (`y.tab.c`).

Struttura generale di un programma Yacc (2)

- Le regole di traduzione (che, come detto, sono porzioni di codice C) in generale consentono di realizzare applicazioni arbitrarie, non solo un “compilatore”.
- Ad ogni produzione è associata (eventualmente per “default”) un’opportuna regola che viene eseguita nel momento in cui il parser “usa” quella produzione.
- Il codice C della terza sezione deve (in alternativa):
  - implicitamente o esplicitamente utilizzare la funzione `yylex()` di Lex per leggere (e “tokenizzare”) l’input;
  - ridefinire una propria funzione `yylex()` che implementa un analizzatore lessicale.

Un primo esempio (quasi) completo

```
%{    /* From Aho, Sethi, Ullman, fig. 4.58, p. 265, extended by Hans Aberg
         and adapted to the LFC course by Mauro Leoncini. */
#include "calc.h"
#include <stdio.h>
%}

%token NUMBER
%token SQRT PI
%token QUIT

%%
session:  session expr '\n' { printf("%.12g\n", $2) }
     |    session '\n'
     |    session QUIT '\n' { printf("Bye\n"); return SUCCESS; }
     |    /* empty */
     |    error '\n'    { printf("Please re-enter last line: "); yyerrok; }
     ;
expr:   expr '+' term        { $$ = $1 + $3 }
     |  expr '-' term        { $$ = $1 - $3 }
     |  term                 { $$ = $1 }   /* Default action; written for clarity */
     ;
.....
%%
int main() {
       return yyparse();
}
int yyerror(char* errstr) {
       printf("Error: %s\n", errstr);
       return FAILURE;
}
```

Interazione con lo scanner

- Yacc interagisce in maniera “nativa” con Lex, allo scopo di “tokenizzare” l’input.
- Si può tuttavia utilizzare un qualunque altro scanner, a patto naturalmente di osservare le convenzioni che governano l’interazione con Yacc.
- L’aspetti più importante riguarda le regole di passaggio dei token.
- Dividendo le “competenze”:
  - Yacc stabilisce quali sono i token;
  - Lex (lo scanner) definisce come sono fatte le istanze (lessemi) dei token.

Interazione con lo scanner (2)

- Supponiamo, come esempio, che nel programma Yacc `esempio.y` si trovino le seguenti definizione esplicite di token:

```
%token NUMBER
%token ID
%token ASSIGNMENT
```

- Il compilatore Yacc “traduce” queste definizioni in direttive per il preprocessor del C.
- Nel programma compilato (`y.tab.c`) potremmo quindi trovare le seguenti linee:

```
#define NUMBER 258
#define ID 259
#define ASSIGNMENT 260
```

- Yacc si aspetta dunque che lo scanner restituisca il numero 258 quando trova un’istanza di `NUMBER` nel file di input.

Interazione con lo scanner (3)

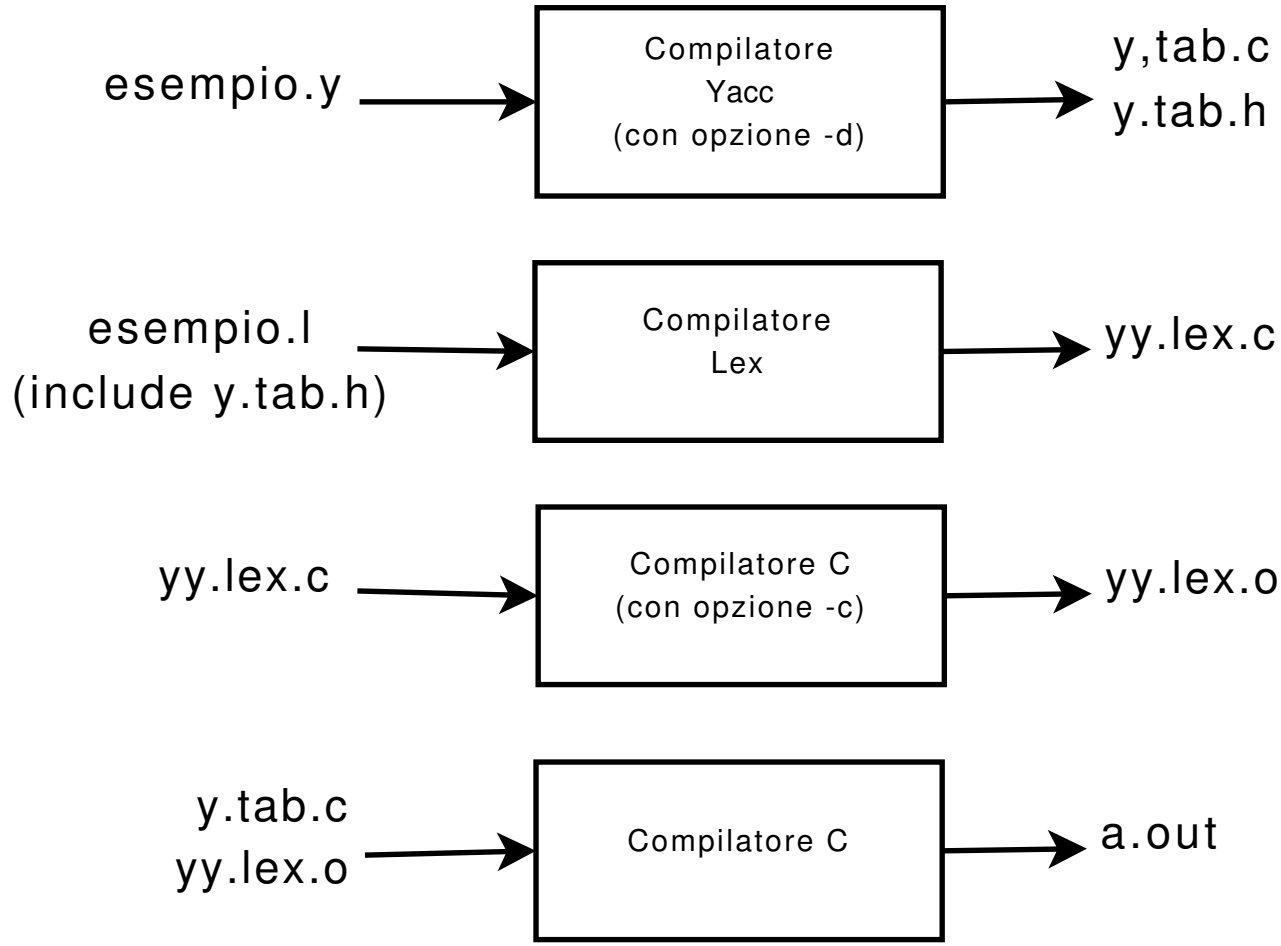
- Da parte sua, Lex deve sapere come riconoscere istanze di `NUMBER` o `ID` nel file di input.
- Già sappiamo che tale “conoscenza” è fornita allo scanner tipicamente mediante espressioni regolari.
- Ma come fa Lex a sapere che, trovata un’istanza di `NUMBER` nell’input, deve restituire a Yacc il numero 258?
- Al riguardo, se il programma Yacc viene compilato con la speciale direttiva `-d`, allora oltre al file oggetto (`y.tab.c`), viene creato un file di definizioni (`y.tab.h`) che può essere usato da Lex.

Interazione con lo scanner (4)

- I token formati da un solo carattere fanno eccezione, o meglio sono trattati implicitamente.
- Dal lato di Yacc essi non vengono definiti con un comando `token` bensì semplicemente utilizzati nelle produzioni racchiusi fra singoli apici.
- Il codice numerico di un token formato da un singolo carattere è semplicemente il suo codice ASCII,
- Per questo motivo, per i token dichiarati esplicitamente (come `NUMBER`, `ID` e `ASSIGNMENT` dell'esempio precedente) Yacc utilizza codici numerici maggiori di 255.

Schema di interazione Lex/Yacc

- Il seguente schema mostra il classico uso congiunto di Lex e Yacc nella realizzazione di un parser/traduttore.



Interazione con lo scanner (5)

- Finora, però, ci siamo solo occupati del token name. Che cosa succede se un token ha anche un valore?
- Ad esempio, al nome `NUMBER` è associato un valore numerico, mentre ad `ID` è associato un identificatore.
- Il valore di un token può anche richiedere una rappresentazione complessa (ad esempio una `struct`)
- La situazione più semplice si verifica quanto tutti i token hanno valori di uno stesso tipo, ad esempio `int`.
- In questi casi Yacc si può utilizzare la variabile globale `yylval` (scritta da Lex e letta da Yacc).
- La definizione di `yylval` viene inserita nel programma oggetto da Yacc; essa viene inoltre definita come external nel file di definizioni `y.tab.h`.

Dichiarazioni e regole di traduzione

- Come già anticipato, nella sezione centrale di un programma Yacc trovano posto le produzioni della grammatica.
- Consideriamo, come semplice esempio, la produzione $< expr > \rightarrow < expr > + < term >$ (che spesso abbiamo scritto come $E \rightarrow E + T$).
- In Yacc essa viene scritta nel modo seguente:

```
expr:  expr '+' term
```

  in cui `expr` e `term` sono simboli non-terminali, il metasimbolo ':' sostituisce la freccia verso destra ($\rightarrow$), non presente nelle tastiere standard, mentre il simbolo '+' è un terminale.
- In generale, un simbolo terminale della grammatica è o un simbolo/identificatore racchiuso tra singoli apici, oppure un identificatore esplicitamente elencato nella prima sezione del programma.

Dichiarazioni e regole di traduzione (2)

- Ad ogni produzione può essere associato (dal programmatore) opportuno codice C.
- Ad esempio, la scrittura della diapositiva precedente potrebbe essere completata nel seguente modo:

```
expr:  expr '+' term  { $$ = $1 + $3; }
```

  in cui la "stringa" racchiusa tra parentesi graffe (che non è ancora codice C) verrà trasformata in codice C da Yacc.
- Se il programmatore non scrive alcun codice per una data produzione, Yacc inserice implicitamente la regola:

```
{ $$ = $1; }
```

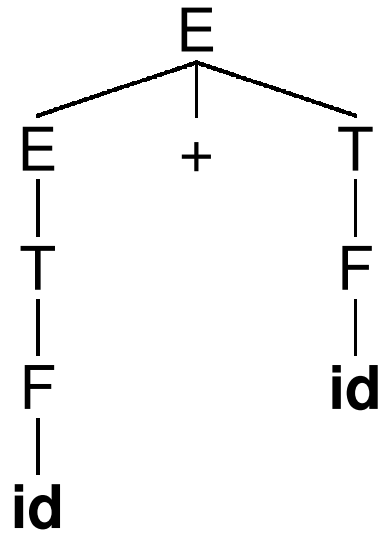
- Per comprendere questo codice (e, soprattutto, poterne scrivere altro!) è necessario riflettere ancora un poco proprio su produzioni e derivazioni.

Simboli e istanze di simboli

- Consideriamo la derivazione destra di **id** + **id** nella grammatica per le espressioni più volte usata:

$$\begin{aligned} E &\Rightarrow E + T \Rightarrow E + F \Rightarrow E + \mathbf{id} \\ &\Rightarrow T + \mathbf{id} \Rightarrow F + \mathbf{id} \Rightarrow \mathbf{id} + \mathbf{id} \end{aligned}$$

- I simboli (terminali e non terminali) usati sono 5 e tuttavia, se contiamo le molteplicità (cioè le “istanze diverse” dei simboli), arriviamo a 9, come si vede bene dal parse tree:



Simboli e istanze di simboli (2)

- Se distinguiamo istanze diverse di uno stesso simbolo usando indici (con un'eccezione per il simbolo '+'), possiamo riscrivere i passaggi di una derivazione canonica destra per $\mathbf{id} + \mathbf{id}$ nel modo seguente, dove mostriamo le produzioni applicate e le forme di frase intermedie:

$$
\begin{array}{lll}
1. & E_2 \to E_1 + T_2, & E_2 \Rightarrow E_1 + T_2 \\
2. & T_2 \to F_2, & \Rightarrow E_1 + F_2 \\
3. & F_2 \to \mathbf{id}_2, & \Rightarrow E_1 + \mathbf{id}_2 \\
4. & E_1 \to T_1, & \Rightarrow T_1 + \mathbf{id}_2 \\
5. & T_1 \to F_1, & \Rightarrow F_1 + \mathbf{id}_2 \\
6. & F_1 \to \mathbf{id}_1, & \Rightarrow \mathbf{id}_1 + \mathbf{id}_2
\end{array}
$$

- Bisogna però spiegare lo "strano" uso degli indici (che non partono da, bensì arrivano a 1).

Un'anticipazione sull'algoritmo di parsing

- L'algoritmo di parsing utilizzato da Yacc è di tipo bottom-up.
- Per mostrare la correttezza di una frase α, il parser parte da α e "risale" all'assioma applicando, in ordine "rovesciato", una sequenza di riscritture corrispondenti ad una derivazione canonica destra.
- Anche le produzioni sono applicate "alla rovescia" (prendendo così il nome di *riduzioni*): si sostituisce cioè la parte destra con la parte sinistra.
- Se $\alpha = \mathbf{id} + \mathbf{id}$, la sequenza di riduzioni e la corrispondente trasformazione sull'input sono:

1.	$F_1 \to \mathbf{id}_1$,	$\mathbf{id}_1 + \mathbf{id}_2 \Rightarrow F_1 + \mathbf{id}_2$
2.	$T_1 \to F_1$,	$\Rightarrow T_1 + \mathbf{id}_2$
3.	$E_1 \to T_1$,	$\Rightarrow E_1 + \mathbf{id}_2$
4.	$F_2 \to \mathbf{id}_2$,	$\Rightarrow E_1 + F_2$
5.	$T_2 \to F_2$,	$\Rightarrow E_1 + T_2$
6.	$E_2 \to E_1 + T_2$,	$\Rightarrow E_2$

Simboli e istanze di simboli (3)

- Ad ogni istanza di un simbolo non terminale, e a molti simboli terminali, Yacc associa (almeno logicamente), una variabile interna, manipolabile dal programmatore usando identificatori appropriati.
- L'identificatore `$$` denota sempre la variabile associata alla testa della produzione.
- Gli identificatori `$1`, `$2`, ... si riferiscono invece alle variabili associate alle istanze dei simboli (terminali e non terminali) nella parte destra della produzione.
- Ad esempio, nella regola

  ```
  expr:  expr '+' term  { $$ = $1 + $3; }
  ```

  i simboli `$$` e `$1` si riferiscono (alle variabili associate) ad istanze del non terminale *E* (`expr` nel programma Yacc) mentre `$3` si riferisce ad un'istanza di *T* (`term`).

Simboli e istanze di simboli (4)

- Quali siano effettivamente le istanze (e dunque le variabili interne) alle quali si fa riferimento in una regola dipende dal preciso momento in cui viene effettuata la riduzione che usa quella regola.
- Ad esempio, con riferimento al riconoscimento di **id** $+$ **id**, la riduzione $E \rightarrow E + T$ viene utilizzata una sola volta.
- Sappiamo anche che tale riduzione andrebbe in realtà letta come $E_2 \rightarrow E_1 + T_2$.
- Possiamo allora concludere che, nel codice generato da Yacc, `$$` diventerà un riferimento (l-value) alla variabile associata ad E_2, mentre `$1` e `$3` diventeranno riferimenti (r-value) alle variabili associate rispettivamente a E_1 e T_2.

Un primo esempio completo: una semplicissima calcolatrice

- Riconsideriamo ancora una volta, con “piccole modifiche”, la grammatica per le espressioni aritmetiche più volte utilizzata:

$$
\begin{aligned}
L &\rightarrow E\ \textbf{eol} \\
E &\rightarrow E + T \mid T \\
T &\rightarrow T \times F \mid F \\
F &\rightarrow (E) \mid \textbf{digit}
\end{aligned}
$$

- Come prima cosa si può osservare la presenza di una nuova produzione (e di un nuovo assioma L) il cui significato “informale” è il seguente: una linea di comando (non terminale L) è un’espressione (E) seguita da un simbolo di fine riga (**eol**).

Una semplicissima calcolatrice (2)

- Si noti quindi come la produzione aggiuntiva ci proietti in un “contesto computazionale”, dove si suppone che l’espressione venga digitata sulla linea di comandi di un terminale a caratteri.
- La seconda modifica è in realtà una semplificazione, inaccettabile nella pratica ma utile a scopo didattico.
- Per rendere più semplice la creazione di un analizzatore lessicale la grammatica prevede che le espressioni possano essere formate solo a partire da operandi numerici composti da una sola cifra.
- Il codice Yacc corrispondente è mostrato nella diapostiva successiva.

Codice Yacc per la semplice calcolatrice


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

%token DIGIT

%%
line   : expr '\n'          { printf("%d\n", $1); }
       ;
expr   : expr '+' term      { $$ = $1 + $3; }
       | term
       ;
term   : term '*' factor    { $$ = $1 * $3; }
       | factor
       ;
factor : '(' expr ')'       { $$ = $2; }
       | DIGIT
       ;
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

Una semplicissima calcolatrice (3)

- Come si può notare, nella terza sezione del programma è inclusa una procedura dal nome `yylex`.
- Tale procedura viene invocata dal parser tutte le volte che necessita del “prossimo” simbolo terminale (token).
- Dal nome sappiamo che “deve” essere la procedura che implementa lo scanner (dunque questa applicazione non usa Lex).
- Lo scanner in questo caso è realmente molto semplice: ogni carattere (ad eccezione di quelli che corrispondono alle cifre decimali 0-9) viene letto e passato al parser.
- Naturalmente ciò che viene passato è un numero (il codice ASCII del carattere).

Una semplicissima calcolatrice (4)

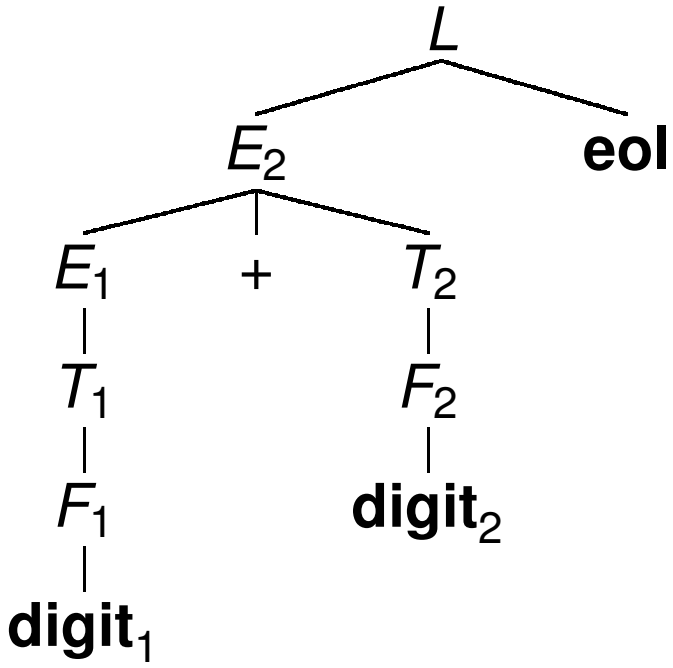
- Solo nel caso di cifre al parser vengono passati sia token name che token value.
- Esplicitamente, come valore di ritorno, viene passato il token name, in questo caso `DIGIT`.
- Come a suo tempo osservato, e come si può vedere ora dall'esempio, il token name serve al parser per stabilire la correttezza sintattica delle espressioni.
- Il valore del particolare token `DIGIT` viene passato tramite la variabile globale `yylval`
- Il valore passato è naturalmente il valore numerico della cifra letta.

Un esempio di computazione

- Supponiamo che i nomi attribuiti alle variabili interne da Yacc seguano le seguenti convenzioni:
  - `E1, E2, ...` per le variabili associate ad istanze del non terminale *E* (`expr` in Yacc);
  - `T1, T2, ...` per le variabili associate ad istanze del non terminale *T* (`term`);
  - `F1, F2, ...` per le variabili associate ad istanze del non terminale *F* (`factor`);
  - `L` per la variabile associata all'unica istanza dell'assioma *L* (`line`).

Un esempio di computazione (2)

- Consideriamo ora derivazione canonica destra e parse tree per la stringa **digit** $+$ **digit eol**.



- La principale differenza con la derivazione per **id** $+$ **id** è la prima riscrittura $L \rightarrow E$ **eol**.
- La successiva diapositiva mostra la sequenza di riduzioni applicate dal parser e il codice C eseguito, nell'ipotesi che l'input digitato sia `5+8`.

Un esempio di computazione (3)

Riduzione	Codice eseguito
$F_1 \rightarrow \textbf{digit}_1$	`F1 = yylval`
$T_1 \rightarrow F_1$	`T1 = F1`
$E_1 \rightarrow T_1$	`E1 = T1`
$F_2 \rightarrow \textbf{digit}_2$	`F2 = yylval`
$T_2 \rightarrow F_2$	`T2 = F2`
$E_2 \rightarrow E_1 + T_2$	`E2 = E1 + T2`
$L \rightarrow E_2$ **eol**	`printf("%d\n", E2)`

- L'esecuzione, nella sequenza indicata, dei singoli frammenti di codice porta al calcolo corretto della somma e alla relativa stampa del valore calcolato.
- Si noti che il valore associato dal parser ai token con nome **digit** è recuperato dalla variabile globale `yylval`.

Un esempio più complesso

- Vogliamo realizzare una calcolatrice decisamente più avanzata di quella vista finora.
- In particolare, la nuova versione deve:
  - utilizzare operandi numerici (razionali) espressi in virgola fissa;
  - disporre almeno delle quattro operazioni aritmetiche, dell'elevamento a potenza e dell'estrazione di radice quadrata;
  - consentire il calcolo (interattivo) di più espressioni, fino all'immissione di un esplicito comando di uscita.
- Curiosamente, è forse quest'ultimo il requisito meno agevole da soddisfare (o meglio, sul quale bisogna ragionare un poco di più).

La grammatica per la nuova calcolatrice

- La parte “facile” riguarda l’estensione del set di operazioni disponibili, con le regole di precedenza e di associatività abituali:

$$
\begin{aligned}
E &\rightarrow E + T \mid E - T \mid T \\
T &\rightarrow T \times F \mid T/F \mid F \\
F &\rightarrow P\hat{}F \mid -F \mid P \\
P &\rightarrow (E) \mid \textbf{sqrt}(E) \mid \textbf{pi} \mid \textbf{number}
\end{aligned}
$$

- Con questa definizione abbiamo anche aggiunto l’operatore $-$ unario e la costante pi greco.
- Si noti che la grammatica utilizza 10 token, di cui almeno tre (**sqrt**, **pi** e **number**) andranno definiti esplicitamente nel programma Yacc.
- Quel che vogliamo fare ora è estendere la grammatica in modo da garantire la possibilità di eseguire più volte il calcolo di espressioni.

Una grammatica interattiva

- Ci aspettiamo, ovviamente, che un'espressione immessa venga calcolata non appena si incontra, nello stream di input, il carattere di `'\n'` (newline), dopodiché vogliamo che il sistema si predisponga ad accettare una nuova espressione.
- L'esecuzione termina quando, anziché un'espressione, l'utente digita (ad esempio) il comando `quit`.
- Possiamo allora definire una sessione di lavoro come una sequenza di espressioni alternate a caratteri `'\n'` e terminate dal comando `quit`.
- Usando la notazione ovvia, possiamo descrivere le sessioni di lavoro come sequenze:

$$E\ \textbf{eol}\ E\ \textbf{eol} \ldots E\ \textbf{eol quit eol}$$

che la nostra grammatica dovrà essere in grado di generare.

La grammatica per la nuova calcolatrice

- La grammatica estesa per soddisfare anche il requisito di interattività è la seguente:

$$
\begin{aligned}
S &\rightarrow S\ \textbf{quit}\ \textbf{eol} \mid S\ E\ \textbf{eol} \mid S\ \textbf{eol} \mid \epsilon \\
E &\rightarrow E + T \mid E - T \mid T \\
T &\rightarrow T \times F \mid T/F \mid F \\
F &\rightarrow P\hat{}F \mid -F \mid P \\
P &\rightarrow (E) \mid \textbf{sqrt}(E) \mid \textbf{pi} \mid \textbf{number}
\end{aligned}
$$

dove la produzione $S \rightarrow S\ \textbf{eol}$ è stata aggiunta per non incorrere in errore nel caso in cui l'utente digiti due newline consecutivi.

Una rudimentale gestione degli errori

- Nel codice per il parser, esiste una produzione aggiuntiva, che fa uso del non terminale riservato `error`.

```
session:  error '\n'
```

- Yacc esegue (idealmente) una riduzione a questo non terminale tutte le volte che si verifica una condizione di errore.
- Tale riduzione (o meglio, la parte destra della produzione, che contiene l'errore) non include il newline.
- Ad esempio, se l'utente digita: `4+-3 '\n'`, il parser "riduce" la stringa `4+-3` (ma non il newline) ad `error`.
- In questo modo il parsing può procedere e la successiva riduzione effettuata sarà proprio `session:  error '\n'`.

Il parser per la calcolatrice avanzata


```
%{
        /* From Aho, Sethi, Ullman, fig. 4.58, p. 265, extended by Hans Aberg
           and adapted to the LFC course by Mauro Leoncini. */

#include "calc.h"
#include <stdio.h>
%}

%token NUMBER
%token SQRT PI
%token QUIT

%%
session: session expr '\n' { printf("%.12g\n", $2) }
      |  session '\n'
      |  session QUIT '\n' { printf("Bye\n"); return SUCCESS; }
      |  /* empty */
      |  error '\n'         { printf("Please re-enter last line: "); yyerrok; }
      ;
expr:    expr '+' term      { $$ = $1 + $3 }
      |  expr '-' term      { $$ = $1 - $3 }
      |  term               { $$ = $1 }   /* Default action; written for clarity */
      ;
term:    term '*' factor    { $$ = $1 * $3 }
      |  term '/' factor    { if ($3==0.0) printf("Warning: divide by zero\n");
                              $$ = $1 / $3 }
      |  factor             { $$ = $1 }   /* Default action; written for clarity */
      ;
factor:  power '^' factor   { $$ = pow($1, $3) }
      |  '-' factor         { $$ = -$2 }
      |  power              { $$ = $1 }   /* Default action; written for clarity */
      ;
power:   '(' expr ')'       { $$ = $2 }
      |  PI                 { $$ = 4*atan(1) }
      |  SQRT '(' expr ')'  { if ($3<0.0) printf("Warning: negative sqrt argument\n")
                              $$ = sqrt($3) }
      |  NUMBER             { $$ = $1 }   /* Default action; written for clarity */
      ;
%%
```

Una rudimentale gestione degli errori (2)

- Si noti come il parser effettua il “recupero” dall’errore.
- Come già osservato, dopo la riduzione “implicita” viene eseguita la riduzione

```
session:  error '\n'
```

- A quest’ultima è associato codice C che:
  - richiede all’utente di digitare nuovamente l’input;
  - informa il parser (mediante la macro `yyerrok`) che è stato effettuato il ripristino dalla condizione di errore.
- Si noti che, in realtà, non viene eseguita nessuna riduzione implicita.
- Quel che accade è semplicemente che Yacc, dopo avere rilevato l’errore, ignora l’input fino al successivo newline.

Lo scanner per la calcolatrice avanzata


```
%{

        /* From Aho, Lam, Sethi, Ullman, fig. 4.60, p. 295, extended
           by Hans Aberg and adapted to the LFC course by Mauro Leoncini. */


#include "calc.h"
#include "calc.tab.h"

%}

%option noyywrap

digit           [0-9]
number          {digit}+\.?|{digit}*\.{digit}+

%%

[ \t]           { /* Skip blanks and tabs. */ }
{number}        { sscanf(yytext, "%lf", &yylval); return NUMBER; }
"sqrt"          { return SQRT; }
"pi"            { return PI; }
"quit"|"exit"   { return QUIT; }
\n|.            { return yytext[0]; }

%%
```

Calcolatrice: versione 3

- L'ultima versione della calcolatrice prevede la possibilità di utilizzare variabili e assegnamenti.
- Una variabile alla quale sia stato assegnato un valore può essere utilizzata nelle espressioni.
- Gli dentificatori di variabile verranno memorizzati, insieme al valore correntemente associato, in una struttura dati (symbol table) che viene alimentata e consultata solo dallo scanner.
- Potremo quindi avere sessioni di lavoro del tipo

```
a = 3
b = 1
c = -2
x1 = (-b+sqrt(b^2-4*a*c))/2
x1 = (-b-sqrt(b^2-4*a*c))/2
```

Tipo dei token value

- Il primo e più delicato problema che dobbiamo affrontare riguarda il valore dei token passati dallo scanner al parser.
- Avremo infatti token con valore numerico (i numeri, appunto) e token (gli identificatori) il cui valore è un puntatore alla symbol table.
- Il problema si ripercuote naturalmente anche sui simboli non terminali, ai quali (come abbiamo visto) sono associate variabili interne.
- Sappiamo cioè che la presenza nel programma Yacc (ad esempio) del seguente codice

  ```
  power:  NUMBER { $$ = $1; }
  ```

  provoca un'azione che consiste nell'assegnamento del valore del token alla variabile associata a (un'istanza di) `power`.
- È chiaro quindi che il tipo di token value si riflette sul tipo del non terminale.

La soluzione in Yacc/Lex

- Quando sussiste questo problema (token con valori di tipo diverso), si può definire il tipo della variabile `yylval` che, come sappiamo, rappresenta il "canale" di comunicazione fra Lex e Yacc, come *unione* di tipi.
- Se nel programma Yacc è presente la definizione:

```
%union {
    double dval;
    struct symtbl *symptr;
}
```

allora Yacc genererà le seguenti definizioni C (includendole anche nel file `y.tab.h`)

```
typedef union YYSTYPE {
    double dval;
    struct symtbl *symptr;
}
extern YYSTYPE yylval;
```

La soluzione in Yacc/Lex (2)

- Il valore opportuno (nel caso dell'esempio, numero in precisione doppia oppure puntatore alla symbol table) può ora essere assegnato da Lex usando la usuale notazione:
  - `yylval.dval` se il token ha valore numerico in precisione doppia;
  - `yylval.symptr` se il valore del token è un puntatore alla symbol table.
- Nel programma Yacc è necessario anche inserire le seguenti definizioni:

```
%token <dval> NUMBER
%token <symptr> ID;
```

che consentono a Yacc di attribuire il tipo "giusto" alle variabili interne associate rispettivamente ai token `NUMBER` e `ID`.

La soluzione in Yacc/Lex (3)

- Si noti, peraltro, che tali definizioni consentono a Yacc di qualificare automaticamente le variabili presenti nelle regole.
- Ad esempio, sarà sufficiente scrivere

```
power:  NUMBER { $$ = $1; }
```

anziché

```
power:  NUMBER { $$ = $1.dval; }
```

- Infine, affinché Yacc possa assegnare il tipo corretto alle variabili interne associate ai non terminali (in quasto caso `double`), è necessario inserire le seguenti definizioni:

```
%type <dval> expr
%type <dval> term
%type <dval> factor
%type <dval> power
```

(non è ovviamente necessario associare un tipo a `statement` o `session`).

La symbol table

- La symbol table in questo esempio viene gestita interamente dallo scanner.
- È realizzata in maniera molto semplice, come array (con dimensionamento statico) di puntatori a coppie <identificatore, valore>.
- Quando lo scanner incontra un identificatore nell'input, ricerca (sequenzialmente) il simbolo nella tabella.
  - Se il simbolo è già presente, associa tale valore al token;
  - se il simbolo non è presente ed esiste ancora spazio nella tabella, inserisce il simbolo;
  - altrimenti genera una condizione di errore (non recuperabile).

La symbol table (2)

- La definizione della tabella è:

```
#define NSYM 30
struct symtbl {
    char *name;
    double value;
} symtbl[NSYM];
```

- Il codice della funzione di lookup è riportato di seguito.

```
/* look up a symbol table entry, add if not present */
struct symtbl *symlookup(s) char *s; {
   char *p;
   struct symtbl *sp;
   for (sp = symtbl; sp < &symtbl[NSYM]; sp++) {
          /* is it already here? */
          if (sp->name && !strcmp(sp->name, s))
                return sp;
          /* is it free? */
          if(!sp->name) {
                sp->name = strdup(s);
                return sp;
          }
          /* otherwise continue to next */
   }
   yyerror("Too many symbols") ;
   exit(FAILURE); /* cannot continue */
} /* Symlookup */
```

Scanner per la calcolatrice avanzata


```
%{

        /* From Levine, Mason, Brown, Lex & Yacc, O'Reilly.
           Examples 3.4, 3.5, 3.6, and 3.7, adapted to the LFC
           course by M. Leoncini. */

#include "calc.tab.h"
#include "calc.h"
%}

%option noyywrap

%%

[ \t]             { /* Skip blanks and tabs. */ }
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
                  yylval.dval = atof(yytext);
                  return NUMBER;
                  }
"sqrt"            { return SQRT; }
"pi"              { return PI; }
"quit"|"exit"     { return QUIT; }
[A-Za-z][A-Za-z0-9]* { /* return symbol pointer */
                    yylval.symptr = symlookup(yytext);
                    return ID;
                  }
\n|.              { return yytext[0]; }

%%
```

Parser per la calcolatrice avanzata


```
%union {
  double dval;
  struct symtbl *symptr;
}
%type <dval> expr
%type <dval> term
%type <dval> factor
%type <dval> power
%token <dval> NUMBER;
%token <symptr> ID;
%token SQRT PI
%token QUIT
%%
session:    session statement '\n'
      |     session '\n'
      |     session QUIT '\n' { printf("Bye\n"); return SUCCESS; }
      |     /* empty */
      |     error '\n'        { printf("Please re-enter last line: "); yyerrok; }
      ;
statement: ID '=' expr        { $1->value = $3; printf("= %g\n", $3); }
      |     expr              { printf("= %g\n", $1); }
expr:       expr '+' term     { $$ = $1 + $3; }
      |     expr '-' term     { $$ = $1 - $3; }
      |     term              { $$ = $1; }    /* Default action; written for clarity */
      ;
term:       term '*' factor   { $$ = $1 * $3; }
      |     term '/' factor   { if ($3==0.0) printf("Warning: divide by zero\n"); $$ = $1/$3;}
      |     factor            { $$ = $1; }    /* Default action; written for clarity */
      ;
factor:  power '^' factor     { $$ = pow($1, $3); }
      |    '-' factor         { $$ = -$2; }
      |    power              { $$ = $1; }    /* Default action; written for clarity */
      ;
power:     '(' expr ')'       { $$ = $2; }
      |    PI                 { $$ = 4*atan(1); }
      |    SQRT '(' expr ')'  { if ($3<0.0) printf("Warning: negative sqrt argument\n");
                                $$ = sqrt($3); }
      |    NUMBER             { $$ = $1; }    /* Default action; written for clarity */
      |    ID                 { $$ = $1->value; }
      ;
%%
```