Parsing

Dispensa del corso di Linguaggi e Traduttori

A.A. 2005-2006

Giovanni Semeraro

Consiste nel prendere una stringa di simboli di un linguaggio ed una grammatica per quel linguaggio e da esse costruire l'albero sintattico e quindi determinare se la stringa è sintatticamente corretta.

Può essere:

- top-down;

- bottom-up.

TOP-DOWN

E[1] un metodo di parsing che cominciando con il simbolo goal (scopo della grammatica) cerca di produrre una stringa di simboli terminali che è identica al testo sorgente.

In particolare:

- Inizia col simbologie scopo della grammatica come radice dell'albero;
- ad ogni passo rimpiazza il simbolo più a sinistra non terminale della frase corrente xVy con u se esiste una regola del tipo: V -> u così che verrà generata la frase: xuy.

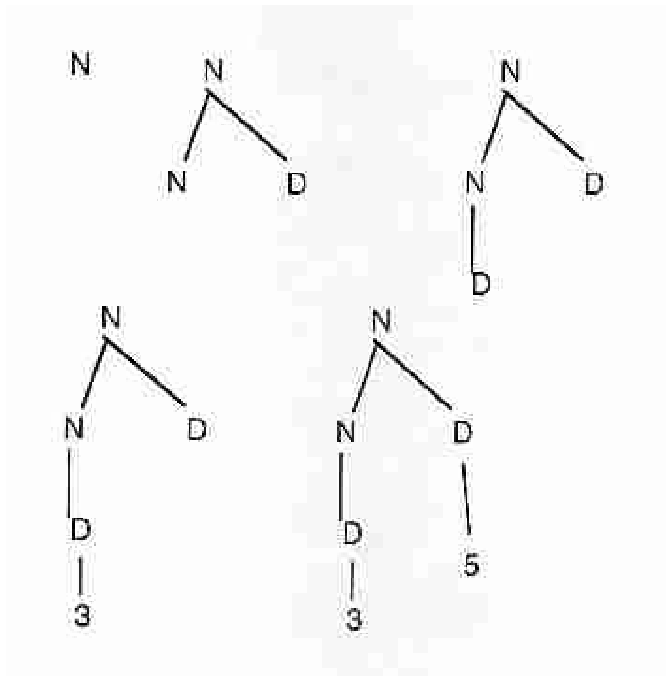
Esempio:

Grammatica:

$N \rightarrow D|ND$ $V_n = \{D, N\}$

$D \rightarrow 0 \mid 1 \mid \ldots \mid 9$ scopo: N

Albero sintattico top-down costruito passo-passo per la stringa 35:



Parser bottom-up:

Costruisce un albero sintattico partendo dalle foglie ed arrivando alla radice.

In pratica crea il contrario di una derivazione destra.

I passi sono:

1) Comincia con la stringa da analizzare (le foglie dell'albero sintattico);

2) Cerca di ridurre la stringa allo scopo della grammatica trovando il corrente *handle:*

Dove l'handle è:

La più grande collezione di terminali e non terminali nella parte più a sinistra dell'input che si possono trovare nel lato destro di una produzione tali che:

- tutti i simboli a destra dell'handle sono terminali;

- rimpiazzando l'handle con la parte sinistra di una produzione e' possibile eventualmente (con altri handle ed eseguendo altri passi) arrivare allo scopo della grammatica.

Creare a mano un parser bottom-up per un vero linguaggio di programmazione è difficoltoso.

Noi vedremo solo i parsers top-down in dettaglio.

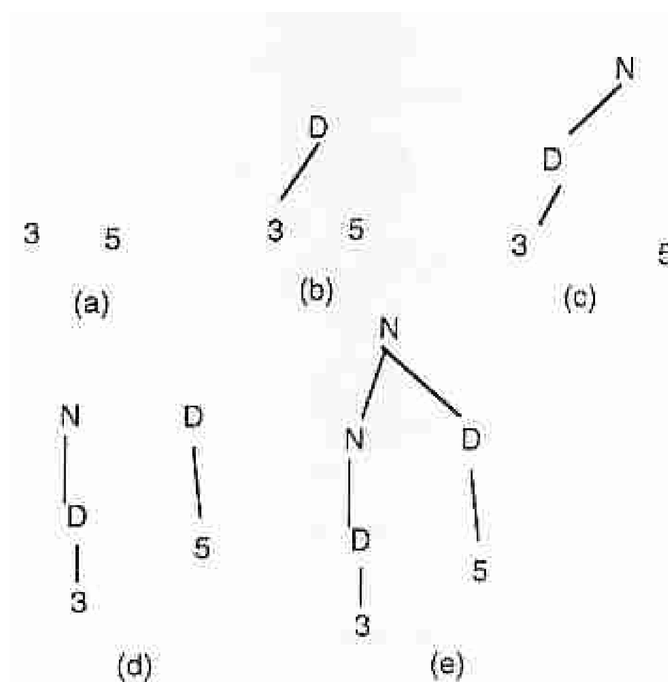
Esempio:

Grammatica:

$N \rightarrow D \mid ND \qquad V_n = \{D, N\}$

$D \rightarrow 0 \mid 1 \mid ... \mid 9 \qquad$ scopo: N

Albero sintattico bottom-up costruito passo-passo per la stringa 35:



Handles:
(a) 3
(b) D
(e) 5
(d) ND

Parsing Top-Down

Backup completo (“brute force” approach).

Cerchiamo di creare un albero sintattico che faccia match con la stringa.

Seguiamo tutte le possibili strade.

Nel caso peggiore, cioè se tentiamo di fare il parsing di una stringa che non è nel linguaggio, si provano tutte le strade possibili per poter generare il fallimento.

Modalità di funzionamento:

1) dato un simbolo non terminale che deve essere espanso si applica la prima produzione per quel non terminale;

2) nella stringa espansa si seleziona il non terminale più[1] a sinistra e si applica la prima produzione;

3) si applica il procedimento al passo 2 per tutti i non terminali fino a che si arriva ad avere prodotto la stringa (successo), oppure si arriva ad avere una parte di stringa che non corrisponde a quella per cui si deve fare il parsing.
   In questo caso si esegue il **backtracking** e se non ci sono più alternative si fallisce.

Esempio

$S \rightarrow aAd \mid aB$ $\quad$ $V_n = \{S,A,B\}$

$A \rightarrow b \mid c$ $\quad$ $V_t = \{a, b, c, d\}$

$B \rightarrow ccd \mid ddc$ $\quad$ scopo: S

stringa in ingresso: accd

(1) genera:
ingresso: accd

(2) genera:a
ingresso: a;ccd

(3) genera: ab
ingresso: ac;cd

(4) genera:ac
ingresso: ac;cd

(5) genera:acd
ingresso: acc;d

(6) genera:a
ingresso: a;ccd

(7) genera:ac
ingresso: ac;cd

(8) genera:acc
ingresso: acc;d

(9) genera:accd
ingresso: accd

Questo metodo non si può applicare ad ogni grammatica libera da contesto.

Definizione

Data una grammatica libera da contesto G=(Vn, Vt, P, S) un terminale X è detto *ricorsivo a sinistra* se $X \Rightarrow^{+} X \alpha$ per un certo $\alpha \in V^*$.

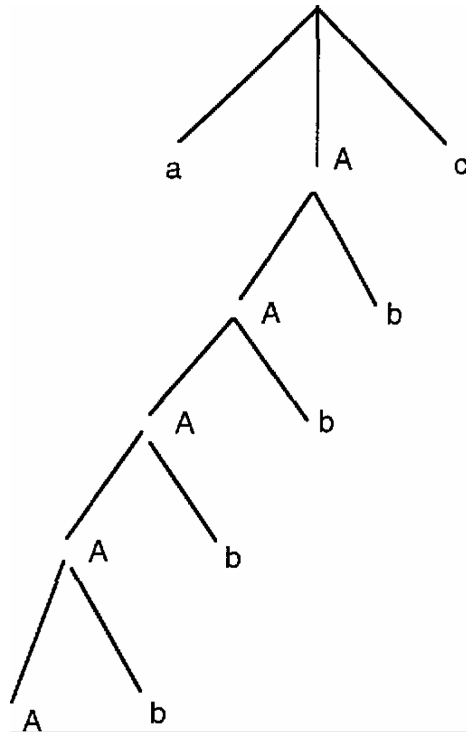
Una grammatica che ha uno o più terminali ricorsivi a sinistra è detta **ricorsiva a sinistra.**

Il nostro metodo in questo caso può andare in un loop infinito.

Esempio:

$S \rightarrow aAc$
$A \rightarrow Ab \mid \varepsilon$
scopo: S
stringa in ingresso: abc

$V_n = \{S, A\}$
$V_t = \{a, b, c\}$



Questo loop si può evitare rendendo opposto l'ordine delle regole per A o riscrivendo la prima produzione per A.

$A \rightarrow \varepsilon \mid Ab$ oppure $A \rightarrow bA \mid \varepsilon$

Non tutte le situazioni possono però essere risolte così facilmente.

Altro esempio:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid a$

E e' ricorsiva a sinistra.

Se si usa un metodo top-down è necessario eliminare la ricorsione a sinistra.

Esiste un metodo generale.

Per ogni regola che contiene una parte ricorsiva a sinistra come:

$A \rightarrow A\alpha \mid ß$

Si introduce un nuovo non terminale e si riscrive la regola come:

$A \rightarrow ßA'$
$A' \rightarrow \varepsilon \mid \alpha A'$

oppure:

$A \rightarrow ß \mid ßA'$
$A' \rightarrow \alpha \mid \alpha A'$

Quindi la produzione:
$E \rightarrow E + T \mid T$

può essere rimpiazzata con

$E \rightarrow TE'$
$E' \rightarrow +T \mid +TE'$

oppure:

$E \rightarrow T \mid TE'$
$E' \rightarrow +T \mid +TE'$

Il metodo può essere esteso a più di due scelte nella parte sinistra:

$A \rightarrow A\alpha 1 \mid A\alpha 2 \mid A\alpha 3 \mid .... \mid A\alpha n \mid \beta 1 \mid \beta 2 \mid ... \mid \beta n$

e' sostituito da:

$A \rightarrow 1A' \mid \beta 2A' \mid .... \mid \beta nA'$

$A' \rightarrow \varepsilon \mid \alpha 1\, A' \mid \alpha 2\, A' \mid \alpha 3\, A' \mid ..... \mid \alpha n\, A'$

Oppure:

$A \rightarrow \beta 1 \mid \beta 2 \mid ..... \mid \beta n \mid \beta 1A' \mid \beta 2A' \mid ..... \mid \beta nA'$

$A' \rightarrow \alpha 1 \mid \alpha 2 \mid ..... \mid \alpha n \mid \alpha 1\, A' \mid \alpha 2\, A' \mid \alpha 3\, A' \mid ..... \mid \alpha n\, A'$

Parsers ricorsivo-discendenti

In questo metodo la strategia discussa precedentemente e' limitata in modo che non sia permesso il backup, che e' molto costoso.

Ovviamente il metodo diventa più efficiente, anche se meno generale poiché non e' applicabile a tutte le

grammatiche libere da contesto.

In questo metodo una sequenza di applicazione di produzioni è realizzata mediante una sequenza di chiamate di funzioni.

In particolare, si scrivono funzioni per ogni non terminale.

Ogni funzione ritorna il valore vero o falso a seconda che riconosca o no una sottostringa che è l'espansione del non terminale.

Se il linguaggio di implementazione del parser supporta la ricorsione, lo stack non è esplicitamente manipolato dall'utente, ma implicitamente dal supporto del linguaggio.

Esempio:

Un algoritmo per l'analisi ricorsivo discendente della seguente grammatica libera da contesto:

<factor>::=(<expr>) | i
<term>::=<factor>*<term>l<factor>
<expr>::=<term>+<expr>l<term>

ha regole ricorsive a destra.

Il parser per questa grammatica contiene una funzione ricorsiva per ogni simbolo non terminale della grammatica (cioè <factor>, <term> e <expr>).

GET_CHAR ritorna nella variabile NEXT (globale) il prossimo carattere nella stringa.

Note:
Il controllo su true o false per le funzioni può essere sostituito da una diretta chiamata alle procedure corrispondenti.

Si poteva trattare una forma iterativa della grammatica del tipo:

<factor>::= (<expr>) | i
<term>::=<factor>{*<factor>}
<expr>::=<term>{+<term>}

In questo caso {*<factor>} e {+<term>} sono realizzati mediante iterazione nelle funzioni <factor> e <term> rispettivamente.

Abbiamo inoltre ignorato la gestione degli errori.

Algoritmo:

Main:

```
1. [Initialize]
   Read (INPUT)
2. [Loop through all input strings]
       Repeat while there stili remains an input string
           Repeat for i = 1,2,....LENGTH(INPUT)
               STRING[i] *<- SUB(INPUT,i,1) CURSOR <- 1
           NEXT <- GET_CHAR
           If EXPR
           Then If NEXT= '#'
                 then Write (INPUT, '□VALID')
                 else Write (INPUT, '□INVALID')
        else Write (INPUT, '□INVALID1)
        Read (INPUT)
   Exit
```

Function EXPR

```
 1. [<expr>::= <term> + <expr> | <term>]
       If not TERM
       then Return(false)
       If NEXT = ' + '
       then NEXT <- GET_CHAR
               If NEXT= '#'
               then Return (false)
               If not EXPR
               then Return (false)
               else Return (true)
       else Return (true)
```

Function TERM

```
1. [<term>::= <factor>*<term> | <factor>]
      If not FACTOR
      then Return(false)
      If NEXT = '*'
      then NEXT <- GET.CHAR
             If NEXT = '#'
             then Return (false)
             If not TERM
             then Return(false)
             else Return(true)
      else Return(true)
```

Function FACTOR

```
1. [<factor)::=(<expr>) | i]
   If NEXT= '#'
   then Return (false)
   If NEXT = '( '
   then NEXT <- GET_CHAR
          If NEXT = '#'
          then Return (false)
          If not EXPR
          then Return (false)
          If NEXT !=  ')'
          then Return (false)
          else NEXT <- GET_CHAR
                 Return(true)
    If NEXT* != ' i '
    then Return (false)
    else NEXT <- GET_CHAR
           Return (true)
```

Function GET_CHAR

```
1. [Returns the next character from the input string]
   CHAR <- STRING [CURSOR]
   CURSOR <- CURSOR + 1
   Return(CHAR)
```

Input: $(i+i) * i\#$

```
h

1   Perform MAIN
1     Call EXPR
1       Call TERM
1         Call FACTOR
1           check for #. No.
1           check for (. Yes. h ← 2.
2           check for #. No.
2           Call EXPR
2             Call TERM
2               Call FACTOR
2                 check for #. No.
2                 check for (. No.
2                 check for i. Yes. h ← 3.
3                 Return true from FACTOR.
3               check for *. No.
3               Return true from TERM.
3             check for +. Yes. h ← 4.
4             check for #. No.
4             Call EXPR
4               Call TERM
4                 Call FACTOR
4                   check for #. No.
4                   check for (. No.
4                   check for i. Yes. h ← 5.
5                   Return true from FACTOR.
5                 check for * . No.
5                 Return true from TERM.
5               Return true from EXPR.
5             Return true from EXPR.
5           check for ). Yes. h ← 6.
6           Return true from FACTOR.
6         check for * . Yes. h ← 7.
7         check for #. No.
7         Call TERM
7           Call FACTOR
7             check for #. No.
7             check for (. No.
7             check for i. Yes. h ← 8.
8             Return true from FACTOR.
8           check for * . No.
8           Return true from TERM.
8         Return true from TERM.
8       check for +. No.
8       Return true from EXPR.
8     check for #. Yes. h ← 9.
9     Return "VALID" from MAIN.
```

Parsing per grammatiche LL(k)

Generalmente la sintassi della maggior parte dei *linguaggi di programmazione* è espressa attraverso la *notazione BNF.* Questo significa che le produzioni della grammatica hanno tutte solo un non terminale nella loro parte sinistra, cioè sono del tipo

A::=п (ovvero A -> п) *.*

Tali grammatiche sono quindi di tipo 2 *(libere da contesto).*

Le tecniche di analisi per i linguaggi liberi da contesto sono state particolarmente studiate. I linguaggi liberi da contesto sono riconosciuti attraverso *automi a pila.*

Tuttavia, per un sottoinsieme di tali linguaggi sono state definite tecniche particolarmente efficienti *(LL parsing)*

La tecnica "LL parsing" (Left-to-right scanning, Left-most derivation) e' un metodo top-clown che si applica a linguaggi deterministici le cui grammatiche hanno particolari proprietà' *(LL(k),* k>=1).

In pratica, quale produzione applicare nella costruzione top-down dell'albero di derivazione e^{x} determinato dal non terminale più a sinistra, dai simboli terminali gia analizzati e dai prossimi *k* simboli della sequenza di input.

Esempio:

S - > C $

C - b A | a B

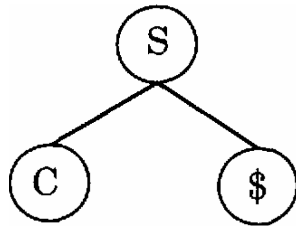
A - > a | a C

B - > b | b C

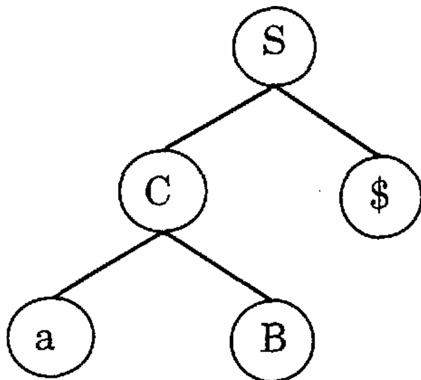
V_n = { S , C , A , B }

Vt = {a, b, $}   scopo: S

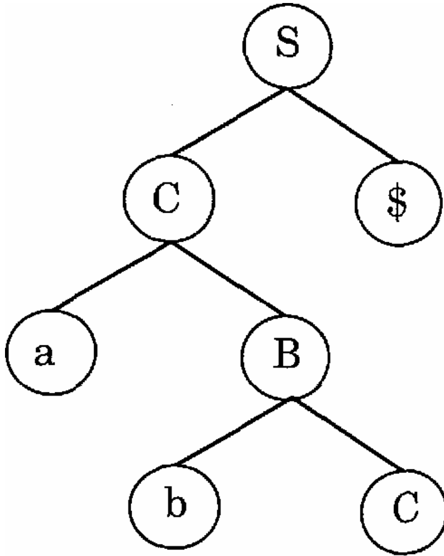
stringa in ingresso: abba$



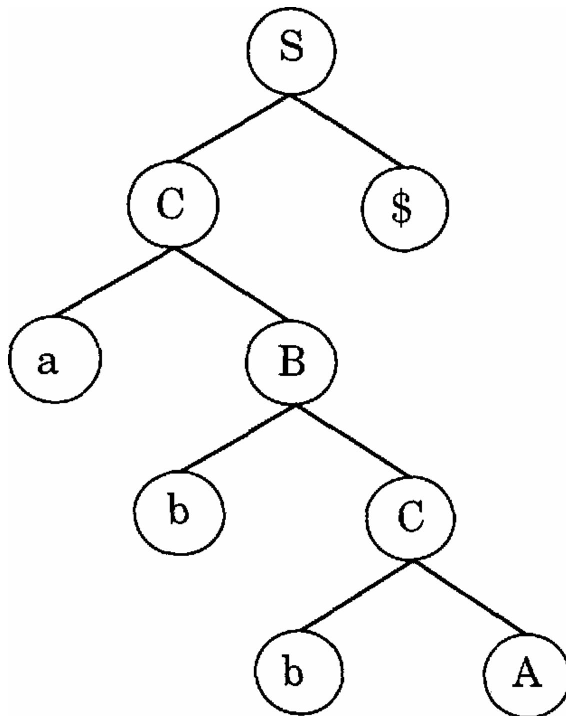
Quale produzione applicare per C e' determinato guardando il prossimo (primo) simbolo in ingresso (a):



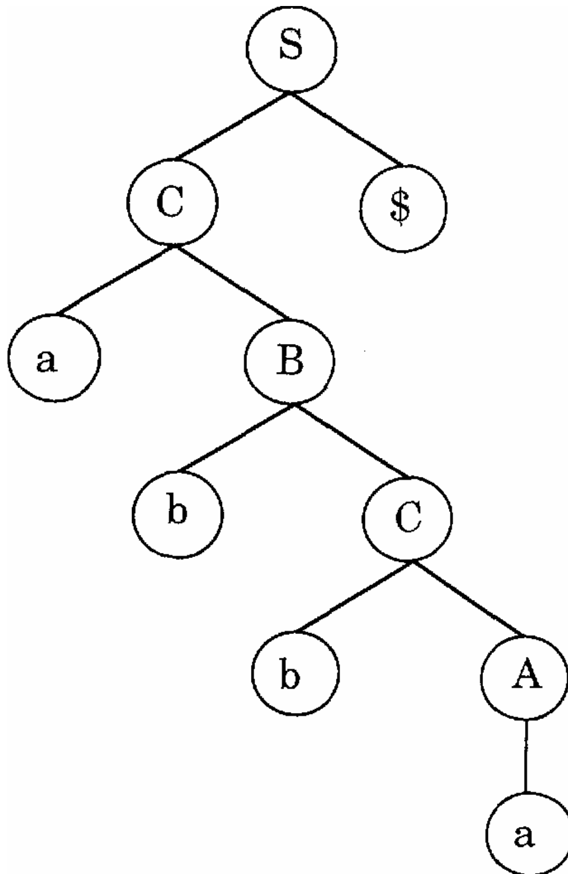
Per determinare come espandere B, si guardano i prossimi 2 simboli in ingresso (bb):



Per determinare come espandere C, si guarda il prossimo simbolo in ingresso (b):



Per determinare come espandere A, si guardano i prossimi 2 simboli in ingresso (a$):



E' una grammatica LL(2).

Per le grammatiche LL(k) e' possibile scrivere analizzatori sintattici che utilizzano il metodo noto come *ricorsivo discendente.*

Questa tecnica non utilizza uno *stack* esplicitamente (come il riconoscitore attraverso PDA), ma lo usa *implicitamente* attraverso l'uso di procedure ricorsive che, ovviamente, sono realizzate attraverso uno stack (modello run-time).

Hanno particolare importanza le grammatiche *LL(1)*, quelle in cui cioè guardando avanti di 1 simbolo di ingresso (1 lookahead) si stabilisce quale produzione applicare (deterministicamente).

In questo caso il riconoscitore *(parser)* ha:

- una procedura (proc_X) per ciascun simbolo $X \in Vn \cup Vt$. proc_X riconosce qualunque stringa derivabile da X.

- Se $X \in Vt (X = a)$, allora proc_a legge il prossimo simbolo da input e verifica che sia uguale al terminale a.

- Se $X \in Vn (X = A)$, e le produzioni per A sono:

  A $\rightarrow$ α1 A $\rightarrow$ α2 ... A $\rightarrow$ αn
  esaminando il prossimo simbolo da input si determina quale produzione applicare.
  Sia A $\rightarrow$ αi tale produzione con αi=X_{i1} X_{i2} ... X_{im}, allora proc_A chiama le procedure:
  proc_X_{i1}; proc_X_{i2};... ; proc_X_{im}.

Esempio:
Riconoscitore per la seguente grammatica di tipo (L L (1)) :

$S \rightarrow aAB \mid bS$
$A \rightarrow aA \mid bB$
$B \rightarrow AB \mid c$

$V_n = \{S, A, B\}$
$Vt = \{a, b, c\}$
scopo: S

Supponiamo di avere una funzione nextsymbol che legge il prossimo simbolo senza far avanzare la testina di lettura su input.

```
program parser ;
var ...
begin
      proc_S;
      if <no more input>
      then halt
      else not_ok
end.
proc_S≡
            case nextsymbol of
            'a': proc_a; proc_A; proc_B;
            'b':proc_b;proc_S;
            'c':not_ok
            end;
proc_A≡
            case nextsymbol of
            'a': proc_a; proc_A;
            'b': proc_b; proc_B;
            'c': not_ok
            end;
```

```
procedure proc_B≡
          case nextsymbol of
          ' a ' , ' b ' : proc_A; proc_B ;
          ' c ':  proc_c
          end;
proc_a≡
          begin
          ch:=read(ch);
          write(ch) ;
          if ch< >'a'  then not_ok
          end;
proc_b≡
          begin
          ch:=read(ch);
          write(ch) ;
          if ch< >'b '  then not_ok
          end;
proc_b≡
          begin
          ch:=read(ch);
          write(ch) ;
          if ch< >'c '  then not_ok
          end;
```

not_ok e', ad esempio, una procedura che abortisce il programma.

Problema: Riconoscitore per la seguente grammatica di tipo2 (LL(1)):

$S \rightarrow 0\ P\ 1\ @$ $\qquad V_n = \{ S, P \}$

$P \rightarrow 2 \mid S$ $\qquad Vt = \{ 0 , 1 , 2 , @ \}$

scopo: S

Genera le stringhe $O^n 2 1^n$, $n \geq 1$

Specifiche: arrestare il riconoscimento al primo simbolo errato. La stringa corretta e' conclusa dal terminatore @.

```
program riconoscitore;
uses crt;
var ch : char;
    ok : boolean;

  procedure leggiero_o_due;
  begin    (* Leggizero o due *)
  ch := readkey;
  write(ch);
  ok := (ch = '0') or (ch = '2')
  end;     (* Leggizero_o_due *)

  procedure leggiuno;
  begin                (* Leggiuno *)
  ch := readkey;
  write(ch) ;
  ok := ch = '1'
  end;                 (* Leggiuno *)
```

```
procedure leggizero;   (* Leggizero*)
 begin
 ch  := readkey;
 write(ch); ok :
 ok := ch =  '0'
 end;                      ( *   Leggizero *)
```

```
procedure proc_S;
  procedure proc_P;
  begin                    (*proc_P*)
  if ok
  then begin
       leggizero_o_due;
       if ok
       then if ch='0' then proc_S
      end
  end;
begin                    (* proc_S *)
if ok then proc_P;
if ok then leggi_uno
end;
procedure leggiterminatore;
begin      (* Leggiterminatore *)
ch := readkey;
write(ch) ;
ok := ch = '@'
end;      (* Leggiterminatore *)
```

```
begin       (* Riconoscitore *)
clrscr;
ok := true;
leggizero;
if ok
then begin
     proc_S;
     if ok then leggi terminatore
     end;
writeln; if
not ok
then write (' non ' ) ;
writeln (' accettata ' ) ;
repeat until keypressed
end.            (* Riconoscitore *)
```

Esempio:

Espressioni aritmetiche sulle variabili a, b, c

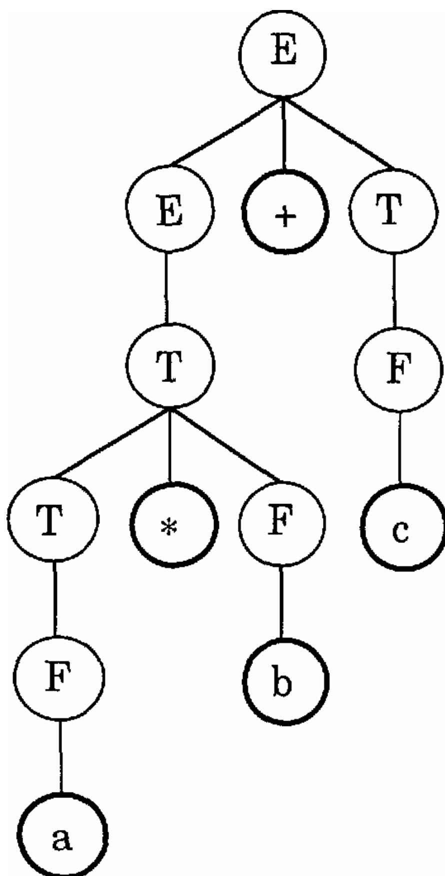
$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow a \mid b \mid c \mid (E)$

E' una grammatica *non ambigua* (esiste un solo albero di derivazione per ogni stringa generata).

Albero di derivazione per a*b+c:



La grammatica e' una semplificazione di quella usata per le espressioni Pascal.

Non è una grammatica *LL(k),* per qualunque k.

Infatti dallo scopo E è possibile generare, ad esempio, (((... (a). ..)))+b e (((... (a). ..)))-b con k+1 parentesi aperte. Guardando k simboli in ingresso non si riesce a stabilire se applicare la produzione $E \to T+E$ oppure $E \to T-E$.

L'analisi discendente ricorsiva *non* e' *applicabile* perche' si ha ricorsione sinistra ($E \to E + T$ | ...)

Per avere un *parsing* efficiente la grammatica viene riscritta come (EBNF):

E→T {+T} | T {-T}

T→F {*F} | F {/F}

F→a|b|c| (E)

Il riconoscitore per questa grammatica usa la tecnica di analisi ricorsiva discendente e può essere schematizzato come segue:

```
begin
   proc_E;
   if nextsymbol= ' $ ' {terminatore}
   then halt
   else not_ok
end

function nextsymbol: char
begin
   nextsymbol := input^
end;

proc_E
      begin
      proc_T;
      while  nextsymbol=' + ' or
            nextsymbol=' - ' do
         begin
            read(ch)
            proc_T
         end
      end;
```

```
proc_T
   begin
   proc_F;
   while nextsymbol='*' or
        nextsymbol='/' do
      begin
         read(ch);
         proc_F
   end
end;

proc_F
   begin
   case nextsymbol of
   ‘a’,’b’,’c’: read(ch);
   ‘(‘:begin
      read(ch)
      proc__E;
      read(ch);
      if ch<>')' then not_ok
      end;
   ')','$': not_ok
   End
   End;
```

Problema:

Data la grammatica con produzioni:

1. E → T+E | T-E | T

2. T → F*T | F/T | F

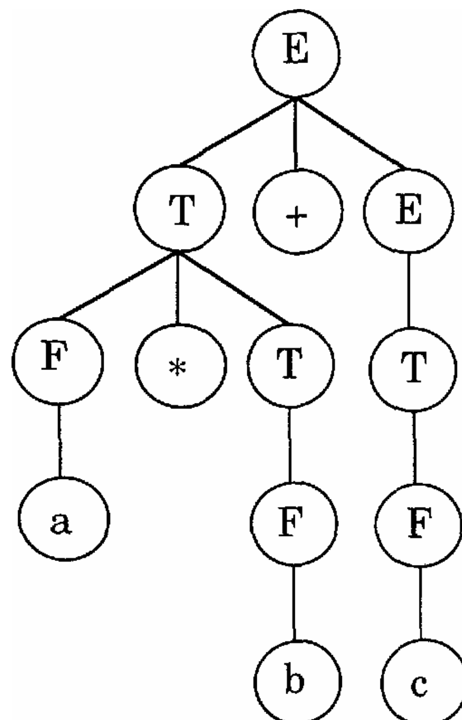
3. F → <lettera> | (E)

4. <lettera> → a | b | ... | z

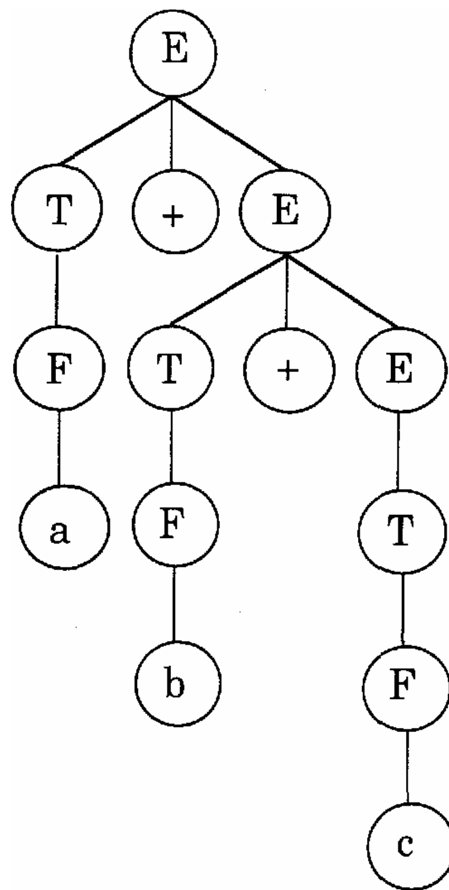
si cerca un algoritmo che riconosca un'espressione.

Si noti che non equivale alla grammatica delle espressioni aritmetiche Pascal (associativita' a destra degli operatori)

Esempio: a*b+c



Esempio: a+b+c



Si noti che non e' associativa a sinistra, ma a destra.

Analizzatore sintattico:

```
program parser_expr;
(* Accetta l'espressione *)
var ch : char;
   ok : boolean;

   procedure analisi;
   var letto : boolean;

procedure
prendi_il_prossimo (var c:char);

   begin        (* prendi_il_prossimo *)
    if not letto then read(c)
    else   letto   := false
    end;        (* prendi_il_prossimo *)
```

```
procedure espressione;
    (* Riconosce un'espressione *)

      procedure termine;
       (* Riconosce un termine *)

  procedure fattore(var p : punt);
        (* Riconosce un fattore *)

 begin   (* analisi *)
 letto := false;
 espressione
 end;     (* analisi *)

 begin  (* parser_expr *)
  write('Inserire una stringa
         terminata con $ ' ) ;
  ok:=true;
  analisi;
  writeln;
  if ok and (ch='$') then
  writeln ('corretta' )
  else writeln ( 'non corretta')
end    (* parser_expr *)
```

```
procedure fattore
begin                        (* fattore *)
prendi il prossimo(ch);
if ch = '('
then (*espressione tra parentesi*)
     begin
     espressione;
     prendi_il_prossimo(ch) ;
     if eh <> ')' then ok:=false
     end
       (*espressione tra parentesi *)
else begin    (* fattore semplice *)
     if not (ch in ['a'..'z'])
     then ok:=false
     end      (* fattore semplice *)
end                  (* fattore *)
```

```
procedure termine;
begin                              (* termine *)
fattore;          (* analisi di un fattore *)
prendi_il_prossimo ( ch ) ;
if ch in   ['*','/']
then begin (*  termine  non concluso *)
       termine;
       end    (* termine non concluso *)
else begin         (* termine concluso *)

       letto  := true
       end;          (* termine concluso *)
end;                               (* termine *)
```

```
procedure espressione;
begin           (* espressione *)
termine; (*analisi di un termine*)
prendi_il_prossimo(ch) ;
if ch in ['+' , '-']
then begin (*espr. non terminata*)

      espressione
    (* esamina il resto
        dell'espressione *)
      end
    (* espr. non terminata *)
else begin
    (* espressione terminata *)
      letto := true
      end;
    (* espressione terminata *)
end;              (* espressione *)
```

ESERCIZIO

Un file sequenziale di tipo text (ESPR.TXT) contiene stringhe (una per linea) costituite dai caratteri {a,b,c,*,+}. Si realizzi un programma Pascal che riconosca le stringhe del file appartenenti alla grammatica G=(Vt, Vn, P, S):

V_t={a,b,c,*,+}
V_n={E,T}
P={E::=a*T + b
T ::= E | c}

```
program parser;
var
   f   : text;
   S     : string;
   ok    : boolean;

procedure parser (S: string; var ok
boolean) ;
var i:integer;
   procedure E;
      procedure T;
        begin
        case S[i] of
         'a': E;
          'c': i:=i+1
        else ok:=false
                    end
        end;        {T}
```

```
begin {E}
     if (S[i]='a' and (S[i+1]='*')
     then begin
            i:=i+2;
                 T;
                 if (S[i] = '+' ) and
 (S[i+1]='b') and ok
               then i:=i+2
          else ok:=false
         end
     else ok:=false
   end;   {E}

begin  {parser}
if length(S)>0
then begin
       ok:=true;
       i:=1;
       E
     end
else ok:=false  {stringa vuota}
end;   {parser}
```

```
begin  {main}
  assign(f, 'E:ESPR.TXT');
  reset(f);
  ok:=true
  while not eof(f) and ok do
    begin
    readln(f,S);
    parser(S/ok);
    end;
  if ok then write('parsing
terminato correttamente')
  else write('parsing terminato con
errore')
end.  {main}
```

Grammatiche ridotte

Regole della forma:

$A \rightarrow A$ possono essere eliminate della grammatica.

Rendono la grammatica ambigua perchè possono essere applicate più volte generando alberi sintattici differenti.

Si consideri la seguente grammatica:

$S \rightarrow aAa$

$A \rightarrow Sb$

$A \rightarrow bBB$

$B \rightarrow abb$

$B \rightarrow aC$

$C \rightarrow aCA$

Si nota che non si può produrre alcuna sentenza terminale dalla produzione C, per cui si possono eliminare tutte le regole che contengono C.

$S \rightarrow aAa$

$A \rightarrow Sb$

$A \rightarrow bBB$

$B \rightarrow abb$

Se un simbolo non terminale genera almeno una stringa terminale e ' detto attivo non terminale.

Esercizio: scrivere un algoritmo che determina tutti i simboli non terminali attivi.

Un altro insieme di simboli non utili sono quelli che non appaiono in almeno una sentenza derivabile dallo scopo (simboli non-raggiungibili).

Esempio:

$S \rightarrow aSb$
$S \rightarrow bAB$
$S \rightarrow a$
$B \rightarrow d$
$A \rightarrow aAc$
$C \rightarrow aSbS$
$C \rightarrow aba$

C non è raggiungibile e quindi possono essere cancellate le regole con C come parte sinistra:

$S \rightarrow aSb$
$S \rightarrow bAB$
$S \rightarrow a$
$B \rightarrow d$
$A \rightarrow aAc$

Esercizio: scrivere un algoritmo che determina tutti simboli raggiungibili.

Una grammatica è ridotta se si mantengono solo i simboli attivi e raggiungibili e si eliminano le regole del tipo: $A \rightarrow A$.

Esempio:

$S \rightarrow ccc$
$S \rightarrow Abccc$
$A \rightarrow Ab$
$A \rightarrow aBa$
$B \rightarrow aBa$
$B \rightarrow AC$
$C \rightarrow Cb$
$C \rightarrow b$

A e B sono inattivi:

$S \rightarrow ccc$
$C \rightarrow Cb$
$C \rightarrow b$

C non è raggiungibile

$S \rightarrow ccc$

è la grammatica ridotta.

(i passi vanno applicati strettamente in questo ordine).

D'ora in poi assumeremo che le grammatiche siano sempre ridotte.

ESERCIZI PROPOSTI (da sviluppare in Pascal e Prolog)

1)
Data la grammatica (scopo <intero>):
<cifra>::= 0|1|2|3I4I5I6I7I8I9IAIBICIDIEIF
<intero senza segno>::=<cifra>l
<intero senza segno> <cifra>
<segno>::= +l-
<interox:=<intero senza segno>l
<segno><intero senza segno>

indicare il tipo di linguaggio generato e realizzare un riconoscitore nell'ipotesi di leggere da tastiera (un carattere alla volta) una sequenza di caratteri che termini con il simbolo speciale @. Il riconoscitore, nel caso di stringa non accettata, deve arrestarsi sul primo simbolo non corretto. Modificare poi la grammatica per rifiutare interi con zeri a sinistra non significativi.

2)
Data la grammatica :
$S \rightarrow aX \mid aY$
$X \rightarrow 1S$
$Y \rightarrow 1$
con Vn ={S,X,Y} Vt ={a,1} e scopo =S, realizzare un riconoscitore nell'ipotesi di leggere da tastiera (un carattere alla volta) una sequenza di caratteri che termini con il simbolo speciale @. Il riconoscitore, nel caso di stringa non accettata, deve arrestarsi sul primo simbolo non corretto.

3) Data la grammatica :

$Z \rightarrow Ua \mid Vb \mid Wc$
$U \rightarrow Zb \mid b$
$V \rightarrow Za \mid a$
$W \rightarrow Zc \mid c$

con $V_n = \{U,V,W,Z\}$
$V_t = \{a,b,c\}$
e scopo = Z

Dire che linguaggio genera e progettare un riconoscitore.