Formato intermedio del programma sorgente

Dispensa del corso di Linguaggi e Traduttori

A.A. 2005-2006

Giovanni Semeraro

Il compilatore, durante la traslazione, puo' creare una forma sorgente intermedia.

Nella forma intermedia si ignorano alcuni dettagli tipici della macchina target.

Inoltre, sono piu' semplici alcune ottimizzazioni del codice.

Dopo aver generato la forma intermedia, questa puo' essere, nella versione prototipale, semplicemente interpretata.

Inoltre il compilatore e' piu' portabile.

Il maggiore svantaggio e' che il codice oggetto puo' diventare meno ottimizzato di quello prodotto direttamente.

Esistono vari tipi di codice intermedio, ne discuteremo alcuni quali:

- Notazione polacca;
- Notazione a n-tuple;
- Alberi sintattici astratti;
- Codice di una macchina astratta.

Notazione polacca

Sono fra i primi sviluppati ed i piu' popolari. E' stato utilizzato per rappresentare la forma intermedia di espressioni.

Puo' essere una forma Polacca **prefissa** o **postfissa.**

La notazione polacca serve per specificare le espressioni aritmetiche in modo non ambiguo senza l'utilizzo di parentesi e quindi offre dei vantaggi rispetto alia tradizionale notazione **infissa.**

Per notazione infissa intendiamo che ogni operatore e' posizionato fra i suoi due operandi (es. a+ b).

Consideriamo una definizione ricorsiva di una espressione aritmetica con l'uso completo di parentesi.

Per semplicita' consideriamo variabili con una singola lettera ed interi non negativi, nonche' gli operatori +,
-, *, e **/.**

L'insieme di tutte le espressioni valide in forma infissa e' contenuto nella seguente definizione ricorsiva:

1. Sono valide espressini infisse contenenti una lettera od un intero non negativo;
2. Se A e B sono espressioni infisse valide allora lo sono anche (A+B), (A-B), (A*B), (A/B);
3. Solo le espressioni agli step 1 e 2 sono espressioni infisse valide.

Si noti che si devono sempre usare delle parentesi.
Per ridurre il numero di parentesi si può assegnare un ordine di precedenza fra gli operatori.

Nel caso che la precedenza degli operatori sia la stessa si puo' stabilire che l'ordine di valutazione sia da sinistra a destra **(associativo a sinistra).**

E' una convenzione usata comunemente.
2-6/3 + 8 sta per (2-(6/3) + 8).

Si consideri la seguente espressione:
w - x/y + z*5
Per valutare l'espressione che corrisponde a:
(w - (x/y)) + (z * 5).
si deve scandire piu' volte da sinistra a destra.

Le parentesi alterano la precedenza degli operatori, ma non evitano la scansione della stringa più volte.

La scansione ripetuta si puo' evitare se si converte prima ad una espressione equivalente prefissa o suffissa in cui le espressioni hanno la forma:

- suffissa o postfissa
<operando> <operando> <operatore>.

- prefissa
<operatore><operando> <operando>.

Invece di quella infissa:
<operando> <operatore> <operando>.

Queste forme sono chiamate anche **forme polacche.**
Nelle forme polacche non esistono le parentesi.

Esempi:

Infissa	Postfissa	Prefissa
x	x	x
x-y	xy-	-xy
x-y+z	xy-z+	+-xyz
(w+x)*(y+z)	wx + yz +*	*+wx+yz
(x+y)/z	xy+z/	/+xyz
x/y/z	xy/z/	//xyz

Un' espressione infissa con parentesi viene traslata in notazione postfissa convertendo inizialmente la sottoespressione più innestata e poi procedendo verso l'esterno.

Esempio:
(x*((y+z) - 5))
prima si converte (y+z) in yz+ e poi si procede fino ad arrivare al risultato: xyz+5-*

In pratica, la forma postfissa o suffissa e' associata ad un parser bottom-up, quella prefissa ad un parser top-down.

Pensiamo adesso a come si valuta un' espressione suffissa come wx + yz + *.

Si scandisce la stringa da sinistra a destra fino ad incontrare il +, gli operandi che lo precedono immediatamente sono applicati al + ed il risultato (r1) sostituito.
Troviamo allora r1 y z + *.

Analogamente applicheremo poi + a y e z ed il risultato r2 verra' sostituito: r1 r2 *. Poi si terminera' con la moltiplicazione di r1 e r2. Si noti che si e' eseguita una singola scansione della stringa.

In pratica si eseguono le seguenti operazioni fino ad aver processato tutta la stringa:
1. Trova l'operatore piu' a sinistra;
2. Trova i due operandi che precedono immediatamente l'operatore;
3. Esegui l'operazione;
4. Rimpiazza l'operatore e i suoi operandi con il risultato.

La Conversione da forma infissa a forma polacca

Consideriamo il caso senza parentesi. Si puo' ottenere facimente utilizzando uno stack.

Consideriamo i seguenti operatori con l'associata precedenza:

Simbolo	Precedenza f
+,-	1
*,/	2
variabili ed interi non negativi	3
#	0

Inizialmente lo stack e' settato al simbolo speciale vuoto. L'algoritmo ripetutamente controlla il valore corrente di input ed il top dello stack.

Se il valore di precedenza del corrente simbolo di input Ic e' maggiore di quello al top dello stack, Ic e' inserito al top dello stack e si determina un nuovo Ic.

Se invece il valore di precedenza del r simbolo di input Ic e' minore od uguale di quello al top dello stack St, St è scritto nella stringa di output dopo di che il procedmento continua confrontando Ic con il nuovo simbolo al top dello stack.

Il procedimento continua finche' sono stati controllati tutti i valori dell'input.

Esempio:

Traslazione di a - b * c + d/5 in notazione polacca suffissa.

Ic	St	Espressione Polacca
	#	
a	#a	
-	# -	a
b	# - b	a
*	# -*	ab
c	#-*c	ab
+	#+	abc*-
d	#+d	abc*-
/	#+/	abc*-d
5	#+/5	abc*-d
#	#	abc*-d5/+

Si noti che, a causa della precedenza alta, una variabile od una costante e' sempre inserita immediatamente nello stack. Inoltre l'espressione a -b - c è convertita in ab-c- corrispondente a (a-b) - c.

L'algoritmo non tratta le parentesi e gli operatori unari del tipo -a, ma puo' essere esteso a questi casi.

Dalla notazione polacca e' molto semplice generare codice per la valutazione delle espressioni nell'ordine esatto. La notazione polacca puo' facilmente essere estesa per trattare anche statement del linguaggio non aritmetici (quali statements condizionali, assegnamento ecc).

Notazione ad N-tuple

E' un codice intermedio in cui ogni istruzione consiste di N campi.

Il primo campo specifica un operatore, mentre gli altri N-1 costituiscono gli operandi.

In questa forma e' molto facile la generazione di codice per una macchina a registri.

Noi vedremo **triple** (N =3) e **quadruple** (N=4).

Triple:
Consideriamo l'epressione:
X + Y
Puo' essere rappresentata da:
+ X Y,
cioe':

<operatore> <operando1> <operando2>

L'espressione W * X + (Y + Z) è rappresentata nella notazione a triple come segue:

1. *, W, X
2. +, Y, Z
3. +, (1), (2)

La tripla 3. specifica che i risultati delle triple 1 e 2 vanno utilizzati come operandi nella terza operazione.

Esempio:

Lo statement condizionale:

if X > Y
then Z := X
else Z := Y + 1

puo' essere rappresentato in una notazione a triple come segue:

1. -, X, Y
2. BMZ, (1), 5
3. :=, Z,X
4. BR, , (7)
5. +, Y, 1
6. :=,Z,(5)
7.

Gli operatori BMZ e BR specificano, rispettivamente, salto condizionato se <= 0 o salto incondizionato.

Le triple non sono molto adatte per l'ottimizzazione del codice. Se si fanno degli spostamenti nell'ordine si ripercuotono sugli operandi di alcune istruzioni.

Quadruple:

In questo caso abbiamo quattro campi che rappresentano rispettivamente:

<operazione> <operatorel> <operatore2> <risultato>

Il risultato e' normalmente una variabile temporanea che puo' essere assegnata ad un registro od ad una cella di memoria centrale in seguito dal compilatore.

Esempio:

(A + B) * (C + D) – E

puo' essere rappresentato da:

+, A, B, T1
+, C, D, T2
*, T1, T2, T3
-, T3, E, T4

dove T1, T2, T3 e T4 sono variabili temporanee.

Le quadruple sono piu' adatte per ottimizzare il codice.

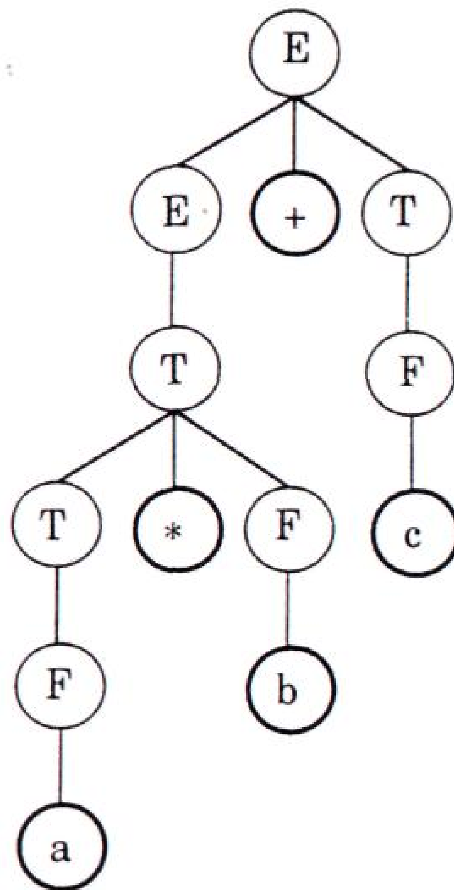
Alberi Sintattici Astratti

In questo caso si utilizza l'albero sintattico prodotto dal compilatore, privandolo pero' di alcune informazioni oramai inutili.

Nell'albero sintattico astratto i nodi che non sono foglie rappresentano operatori, mentre le foglie variabili.

Esempio:
E --> T | E (+ | -) T
T --> F| T (* | /)F
F --> a | b | c | (E)

L'albero sintattico astratto (per a*b+c) viene costruito dall'albero sintattico:



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

Nota:

Le produzioni fanno sì che l'albero sintattico astratto generato porti all'interpretazione usuale dell'espressione. La ***priorità degli operatori*** viene incorporata nell'albero sintattico astratto.

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

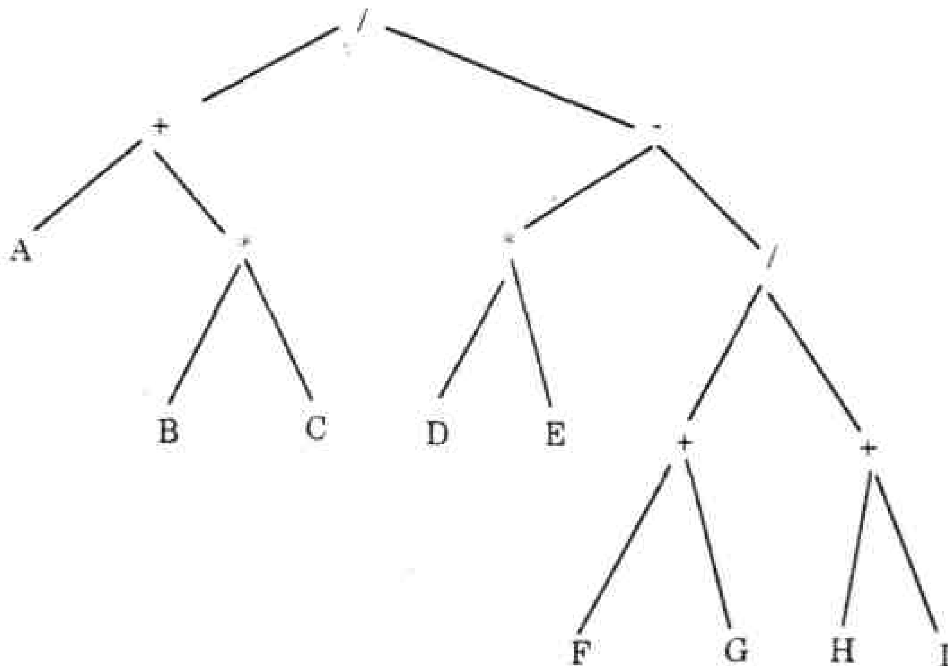
L'analisi discendente ricorsiva ***non è applicabile*** perchè si ha ricorsione sinistra (E --> E + T | ...)

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)

Esempio:

L'espressione (A + B * C) / (D* E - (F + G) / (H+I)), può essere rappresentata dall'albero sintattico astratto seguente:



Percorrere l'albero in pre-ordine o post-ordine porta alla rappresentazione polacca prefissa o postfissa rispettivamente:

/+ A * BC - * DE / + FG + HI
ABC * + DE * FG + HI + / - /

Anche le triple possono essere viste come una rappresentazione dell'albero sintattico astratto.

In particolare l'albero precedente corrisponde al seguente codice a triple:

1. *,B,C
2. +,A,(1)
3. *, D, E
4. +, F, G
5. +, H, I
6. /, (4), (5)
7. -, (3), (6)
8. /, (2), (7)

L'ultima tripla è associata alla radice dell'albero. Ogni tripla corrisponde ad un sottoalbero, con radice l'operando.

Codice di macchine astratte

E' un approccio che garantisce piu' portabilita' e adattabilita' del compilatore.

Supponiamo di voler portare un compilatore da una macchina X ad una macchina Y. Dovremo dunque riscrivere la parte di generazione di codice.

Cio' e' piu' semplice se il compilatore e' stato progettato con due parti distinte e comunicanti con una precisa interfaccia.

La prima parte (front end) tratta con il programma sorgente, la seconda (back end) con la macchina target.

Solo la parte dipendente dalla macchina target deve essere cambiata.

L'interfaccia puo' essere realizzata mediante una macchina di riferimento "astratta".

I costrutti del linguaggio sorgente possono essere tradotti in pseudo-istruzioni per la macchina astratta, che e' progettata per un particolare linguaggio (ad esempio Pascal).

Poi ci si dovra' occupare della efficiente realizzazione della macchina astratta sulla macchina fisica.

Questa separazione vuole dire, almeno in teoria, che passando da una macchina target ad un'altra si deve riprogettare solo il back end.

Supponiamo di dover progettare i compilatori per m differenti linguaggi su n differenti macchine.

Allora avremo la progettazione di n*m compilatori completi se non usiamo l'approccio, macchine astratte, mentre solo m front end ed n back end usando l'approccio machine astratte.

Il problema e' allora trovare una macchina astratta che possa essere efficente per linguaggi cosi' diversi quali Pascal, C, Fortran, Lisp, Prolog.

P-machine: una macchina astratta per il Pascal

Compilatore Pascal-P: e' un compilatore portabile per il Pascal che produce codice (P-code) per una macchina astratta (P-machine).

La P-machine e' una macchina a stack che puo' poi essere implementata in modo efficiente in divrsi computer.

La P-machine ha 5 registri ed una memoria.

La memoria e' divisa in due parti: la prima parte contiene il codice (CODE), la seconda (STORE) i dati.

La memoria STORE cntiene una parte a stack ed una ad heap, gestite secondo le regole gia' viste nella parte di gestione della memoria.

Esistono poi varie istruzioni, molto vicine a quelle del linguaggio Pascal.

Vedremo in un seminario la macchina astratta per il linguaggio Prolog (Warren's Abstract Machine -WAM).

Espressioni e priorità degli operatori

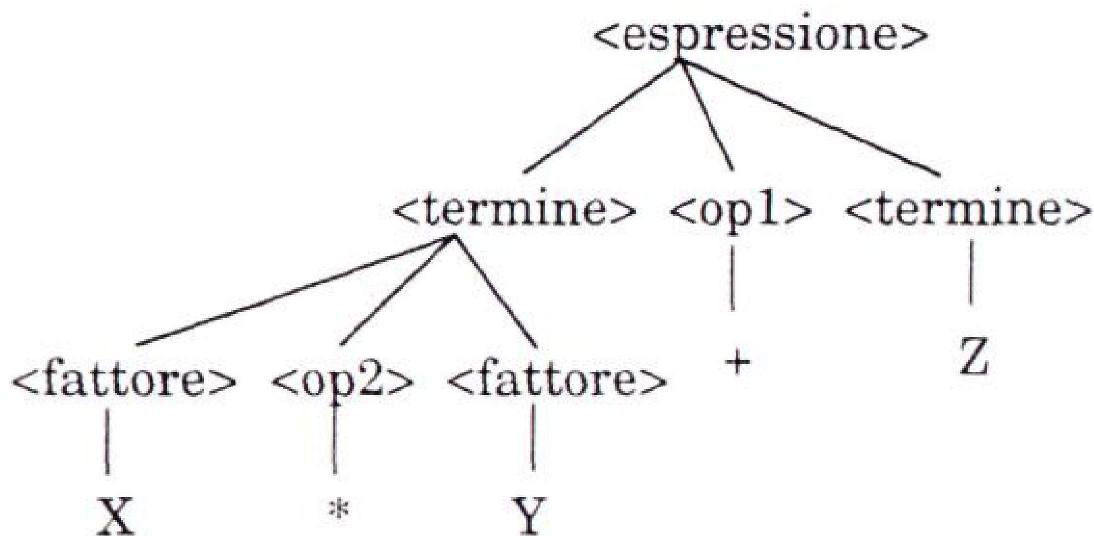
Le regole relative alla priorita' degli operatori sono stabilite dalla sintassi delle espressioni.

Caso semplificato: operatori +, -, *, / su numeri reali (variabili con identificatore X, Y, Z)

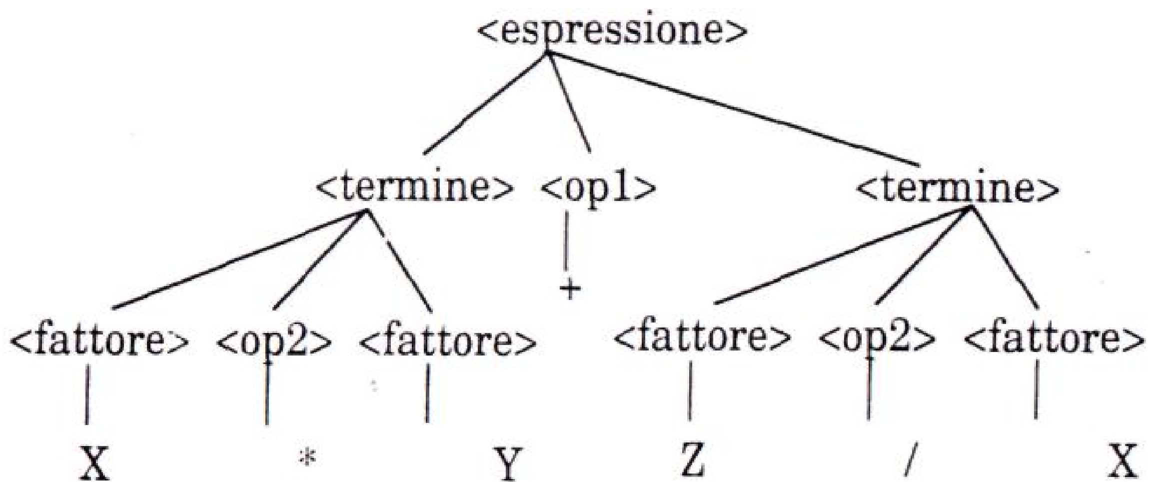
<espressione> ::= <termine> { <opl> <termine> }
<termine> ::= <fattore> { <op2> <fattore> }

<op1> ::= + | -
<op2> ::= * | /
<fattore> ::= X | Y | Z

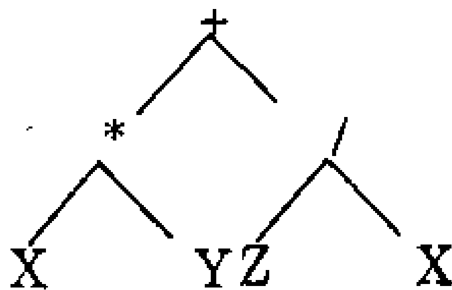
X*Y + Z



X*Y+Z/X

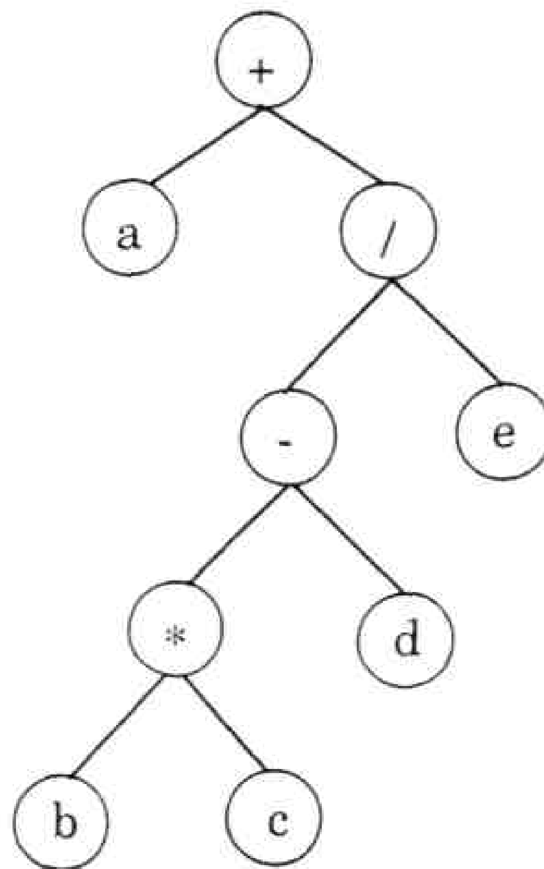


X*Y + Z/X
XY*ZX/+ (polacca postfissa)



Corrisponde ad una visita in ordine posticipato dell'albero sintattico astratto.

Esempio: a+(b*c-d)/e



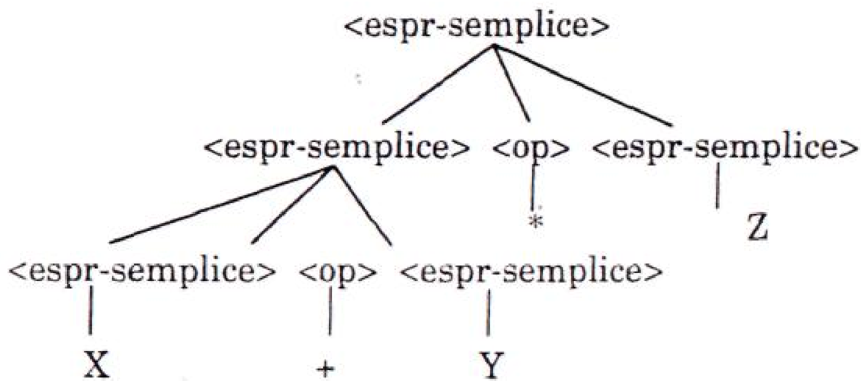
Ordine anticipato: +a/-*bce
(polacca prefissa, operatore, 1° operando, 2° operando)

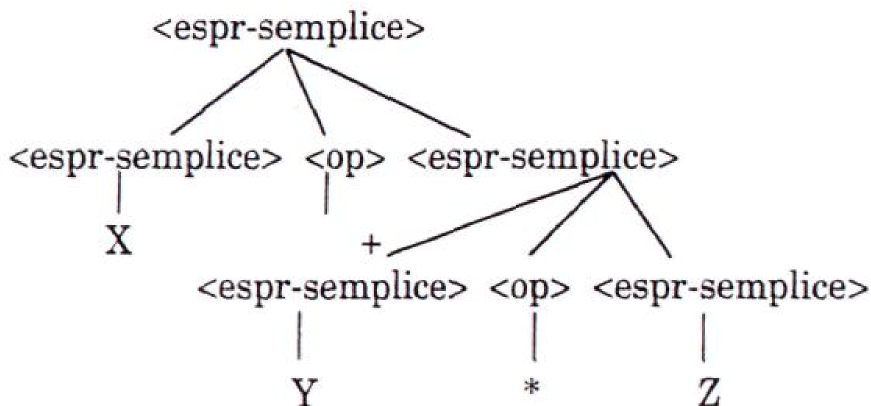
Ordine ritardato: abc*d-e/+
(polacca postfissa, 1° operando, 2° operando, operatore)

Visita simmetrica: a+b*c-d/e
(priorita' degli operatori)

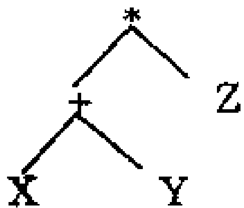
La grammatica:
<espr-semplice> ::= X | Y | Z |
<espr-semplice> <op> <espr-semplice>
<op> ::= + | - | * | /
è ambigua.

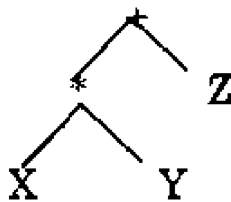
X + Y * Z





Abbiamo due possibili alberi sintattici astratti:





Esercizio:
Scrivere un programma che letta in ingresso una sequenza di caratteri terminata da '.', rappresentante una espressione aritmetica valida con operatori binari +, -, *, / infissi, stampi la stringa in forma polacca postfissa.

<espressione>::= <termine> {(+ | -) <termine>}
<termine>::= <fattore> {(*|/) <fattore>}
<fattore> ::= <lettera> | (<espressione>)

a	a+b	a*b+c	a*b+c/d+e	
a	ab+	ab*c+	ab*cd/+e+	polacca

a*(b-c)*d	a*((b-c)/d)+e	
abc-*d*	abc-d/*e+	polacca

Soluzione della traduzione in polacca postfissa:

I livello di specifica:

```
program polacca (input, output);
var C: char;
begin
    saltabianchi;
    repeat
        write(' ');
        espressione;
        writeln
    until C='.'
end.
```

Codifica procedure:

```
procedure saltabianchi;
begin
    repeat
        read(C)
    until ( C < > ' ' ) or eoln(input)
end;
```

```
procedure espressione;
var op: char;
begin
    termine;
    while (C='+') or (C='-') do
        begin op:=C;
            saltabianchi;
            termine;
            write(op)
        end
end;

procedure termine;
var op: char;
begin
    fattore;
    while (C='*') or (C='/') do
        begin    op:=C;
                 saltabianchi;
                 fattore;
                 write(op)
        end
end;

procedure fattore;

begin
     if C='(' then    begin
                         saltabianchi;
                         espressione
                      end
     else write(C);
     saltabianchi end;
```

Struttura del programma:

```
program polacca (input, output);
var C: char;
    procedure saltabianchi;
    ...
    end;
    procedure espressione;
    var op: char;
        procedure termine;
        var op: char;

            procedure fattore;
            ...
            end;        {fattore}
        ...
        end;        {termine}

    ...
    end;     {espressione}

begin           {corpo main program}
    saltabianchi;
    espressione;
    writeln
end.
```

Mutua ricorsione:

espressione --> termine --> fattore --> espressione.

Esercizio:

Il riconoscitore per la grammatica:
<espressione> ::= <termine> { <op1> <termine> }
<termine> ::= <fattore> { <op2> <fattore> }
<op1> ::= + | -
<op2> ::= * | /
<fattore> ::= a | b | c
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;
```

Il passo successivo è l'inserzione di **azioni** in queste procedure.

Deve essere definita una funzione f_X, per ciascuna procedura proc_X, che restituisce un albero sintattico astratto rappresentante l'espressione aritmetica riconosciuta da proc_X.

Usiamo una variabile **t** di tipo tree (albero binario di caratteri) ed il costruttore **cons_tree.**

L'inserzione di queste azioni risulta specificata come segue:

```
var t: tree;
begin
    t:=f_E;
    if nextsymbol='$'       {terminatore}
    then                    {restituisci t}
    else not_ok
end.

function f_E: tree≡
            var t: tree;
            begin
            t:=f_T;
            while nextsymbol='+' or nextsymbol='-' do
                begin
                      read(ch);
                     t:= cons_tree(ch, t, f_T)
                 end;
            f_E:=t
            end;
```

```
function f_T: tree≡
            var t: tree;
            begin
            t:=f_F;
            while nextsymbol='*' or nextsymbol='/' do
                 begin
                        read(ch);
                        t:= cons_tree(ch, t, f_F)
                  end;
            f_T:=t
            end;

function f_F: tree≡

            begin
            case nextsymbol of
            'a','b','c': begin
                           read(ch);
                           f_F:=cons_tree(ch,nil,nil)
                         end;
            '(': begin
                  read(ch);
                  f_F:= f_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:

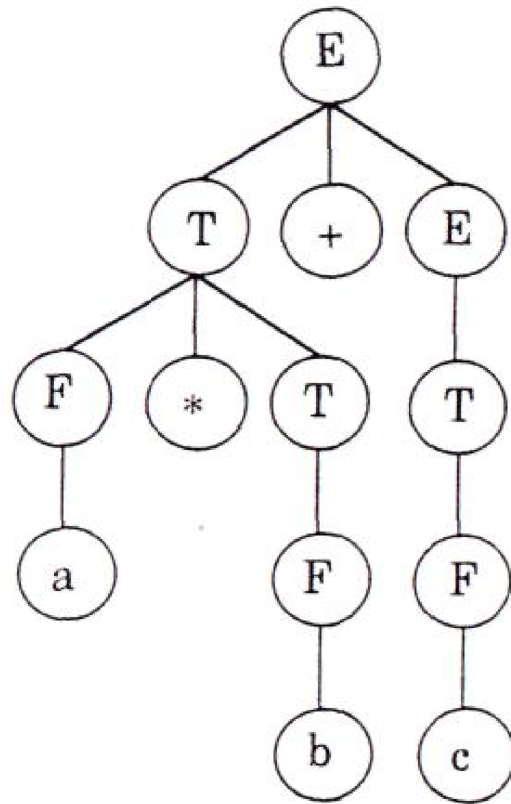
- riconosce una espressione;

- genera un albero che la rappresenta (albero sintatico astratto);

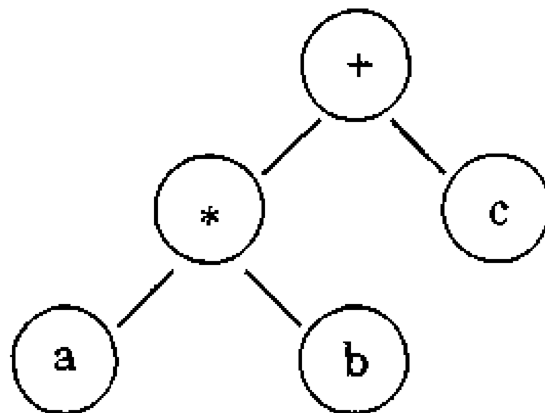
- è ricorsivo.

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

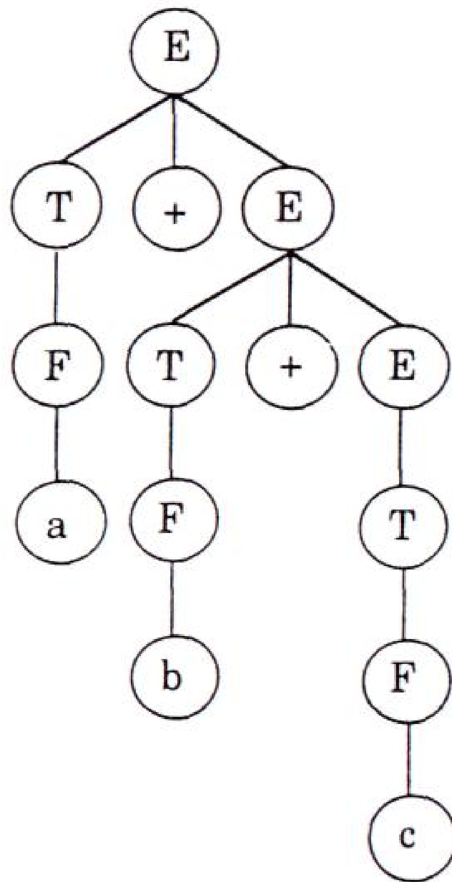
Esempio: a*b+c



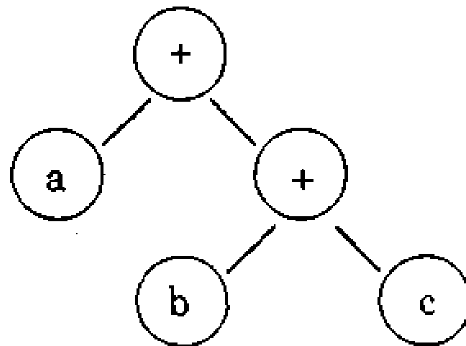
Albero sintattico astratto:



Esempio: a+b+c



Albero sintattico astratto:



Si noti che non è 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 ch <> ')' 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 *)
```

Costruzione dell'albero sintattico astratto:

```
program expr;
(* Accetta l'espressione, costruisce l'albero e
calcola il valore. Non effettua il controllo ! *)
type punt = ^nodo;
      nodo = record
            info : char;
            sx,
            dx   : punt
      end;
var operandi: array ['a' .. 'z'] of real;
  c      : char;
  pt      : punt;

procedure analisi(var p : punt);
var ch    : char;
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(var p : punt);
  (* Legge un'espressione e costruisce
    il relativo albero binario
    con puntatore alla radice p     *)
    var pt : punt;
```

```
procedure termine(var p : punt);
   var pf: punt;
   (* Legge un termine e costruisce il
      relativo albero binario con
      radice p         *)

     procedure fattore(var p : punt);
     (* Legge un fattore di una
        espressione e costruisce il
        relativo albero binario di
        radice p    .     *)

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

function valore(p : punt) : real;
begin
if p^.info in ['a'..'z'] (* nodo foglia *)
then valore := operandi[p^.info]
else                          (* nodo intermedio *)
    case p^.info of
       '+' : valore :=
              valore(p^.sx) + valore(p^.dx);
      '-' : valore :=
              valore(p^.sx) - valore(p^.dx);
       '*' : valore :=
              valore(p^.sx) * valore(p^.dx);
       '/' : valore :=
             valore(p^.sx) / valore(p^.dx);
     end    (* case *)
  end;     (* valore *)
```

```
begin     (* expr *)
inizializzare gli operandi
write('Inserire una espressione ');
analisi(pt);
writeln;
writeln('Il valore é ',valore(pt))
end.      (* expr *)

procedure fattore(var p : punt);
(* Legge un fattore di una
  espressione e costruisce il
  relativo albero binario di
  radice p             *)

begin          (* fattore *)
prendi_il_prossimo(ch);
if ch = '('
then begin (* espressione tra parentesi *)
          espressione(p);
          prendi_il_prossimo(ch);
      end   (* espressione tra parentesi *)
else begin   (* fattore semplice *)
   new(p);
   p^.info := ch;
   p^.sx   := nil;
   p^.dx   := nil;
   end;   (* fattore semplice *)
end;           (* fattore *)
```

```
procedure termine(var p : punt);
var pf: punt;
(* Legge un termine di una
  espressione e costruisce il
  relativo albero binario con
  radice p              *)

begin    (* termine *)
fattore(pf); (* an. di un fattore *)
prendi_il_prossimo(ch);
if ch in ['*','/']
then begin (* termine non concluso *)
    new(p);
    p^.info := ch;
    p^.sx   := pf;
    termine(p^.dx);
    end   (* termine non concluso *)
else begin (* termine concluso *)
    p := pf;
    letto := true;
    end; (* termine concluso *)
end;     (* termine *)
```

```
procedure espressione(var p : punt);
(* Legge un 'espressione e costruisce
 il relativo albero binario   con puntatore alla radice p     *)
 var pt: punt;

begin       (* espressione *)
termine(pt); (* an. di un termine *)
prendi_il_prossimo(ch);
if ch  in['+','-']
then begin (* espr. non terminata *)
           new(p);    (* crea nodo radice del sotto-albero *)
           p^.info := ch;
           p^.sx   := pt; (* connette l'albero del termine pt come
           sottoalbero sinistro dell'op letto *)
           espressione(p^.dx); (* esamina il resto dell'espres-
           sione,che verrà connessa come sottoalbero dx *)
      end    (* espr. non terminata *)
else begin (* espressione terminata *)
  p := pt;
  letto := true;
  end;   (* espressione terminata *)
end;     (* espressione *)
```