Espressioni logiche

Antonio Natali

21 giugno 2004

Indice

Capitolo 1

Riconoscimento e valutazione di espressioni logiche

1.1 Impostazione del riconoscitore

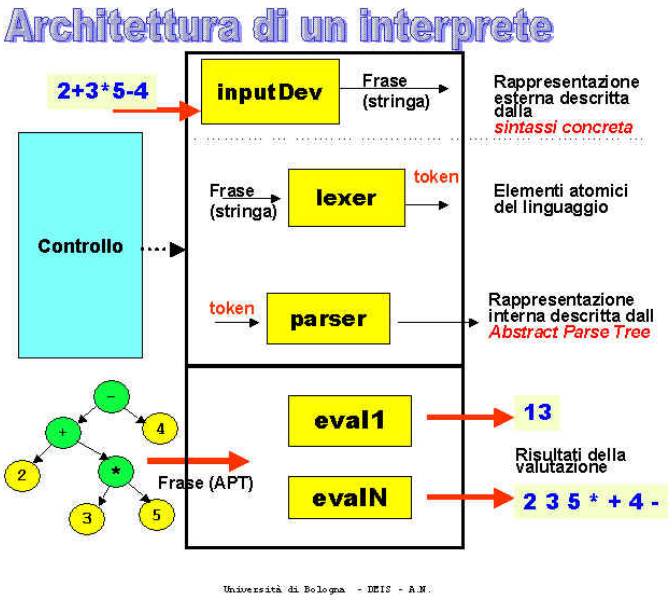
1.1.1 Architettura logica

L'uso di una grammatica *context free* permette di descrivere in modo formale la sintassi concreta di un linguaggio. Normalmente la grammatica è studiata in modo che il linguaggio risulti non solo ben definito, ma anche chiaro all'intuizione di chi legge. La sintassi concreta del linguaggio cioè include molti elementi, quali simboli di punteggiatura, parole chiave, etc. spesso introdotti solo per rendere leggibile il linguaggio agli esseri umani.

L'informazione sulla struttura delle frasi è ovviamente essenziale in ogni processo di elaborazione che coinvolga le frasi del linguaggio. Il primo e fondamentale processo di elaborazione che riguarda le frasi di un linguaggio di programmazione è la sua *interpretazione*, cioè l'attività svolta da un automa (detto interprete) appositamente costruito per riconoscere ed eseguire le frasi del linguaggio.

L'interprete di un linguaggio, come ogni automa, può essere realizzato con tecnologie molto diverse. Il modo più semplice ed economico è certo quello di realizzarlo come un sistema software, essendo l'elaboratore elettronico un automa universale capace di simulare il comportamento di ogni altro automa attraverso una opportuna programmazione.

I principali componenti dell'architettura di un interprete sono rappresentati nella figura che segue:



Nella figura:

- `inputDev` denota il dispositivo dal quale il sistema può prelevare la frase di ingresso;
- `lexer` è il componente che indivuda gli elementi atomici (*token*) che costituiscono la frase di ingresso;
- `parser` è l'analizzatore sintattico, che controlla la corrispondenza della frase alle regole della grammatica che definisce la sintassi concreta del linguaggio. Il parser può produrre direttamente il risultato della valutazione della frase, ma è molto più frequente che produca una rappresentazione interna della frase di ingresso detta *abstract parse tree* (**apt**);
- `eval` denota il componente che, ricevuto in ingresso l'apt, provvede a effettuare azioni di intepretazioe della frase o produzione di nuova informazione, quale una riscrittira della frase in forma postfissa, produzione di codice, etc.

La parte di `controllo` stabilisce le modalità con cui attivare le diverse parti che formano un sistema di interpretazione di frasi. E' molto frequente che il controllo sia definito come un programma principale che invoca il parser che a sua volta invoca il lexer, il quale accede al dispositivo di ingresso. Vi sono però altre modalità di controllo in relazione a diversi modi di concepire l'*architettura* del sistema. E' il caso ad esempio di sistemi di interpretazione distribuiti in cui il dispositivo di ingresso e l'analizzatore sintattico possono

essere allocati su un nodo di elaborazione (*client*) e la valutazione su un nodo diverso (*server*).

Nel seguito svilupperemo un interprete per espressioni cercando di approfondire il rapporto tra tecniche di riconoscimento, impostazione architetturale e uso di linguaggi di programmazione. In particolare useremo `Prolog` come linguaggio per la specifica dei comportamenti (algoritmi) e `Java` come linguaggio per la organizzazione strutturale del sistema di riconoscimento. Per la interoperabilità tra i due linguaggi utilizzeremo `tuProlog`, un inteprete Prolog scritto in Java.

Seguendo un approccio di sviluppo a spirale, costruiremo diversi prototipi di sistema di interpretazione per un semplicissimo linguaggio nel campo delle espressioni logiche. Ciò ci permetterà di definire, confrontare e discutere senza troppo sforzo diverse impostazioni architetturali. Una volta decisa l'organizzazione ritenuta più conveniente in relazione ai requisiti del problema, verificheremo le caratteristiche di modularità ed estendibilità di quanto realizzato impostando un inteprete per espressioni logiche `and-or-not` e aritmetiche. Successivamente questo inteprete potrà essere ulteriormente esteso con riferimento all'uso di simboli (costanti e variabili), alla definizione di funzioni, oggetti, etc. in modo da discutere e ricostruire enti computazionali e comportamenti titpici dei più diffusi linguaggi di programmazione.

1.1.2 Produzioni

Per comprendere le tecniche di base per la progettazione e la costruzione di interpreti, definiamo una prima grammatica molto semplice `G=(VN, VT, S, P)` con:

- `VN = { E,F }` (vocabolario dei non terminali)
- `VT = { true, false, or }` (vocabolario dei terminali)
- `S = E` (*scopo*)
- *produzioni*

```
P = {
    E ::= E or F | F
    F ::= true | false
  }
```

La variabile sintattica `E` (scopo della grammatica) intende denotare una *espressione*. La variabile sintattica `F` intende denotare un *fattore*. L'elemento

terminale `or` denota la disgiunzione logica; gli elementi terminali `true` e `false` denotano i valori di verità.

La struttura sintattica implica associatività a sinistra dell'operatore `or`. In altre parole, la frase

```
true or false or false
```

appartiene al linguaggio `L(E)` generato dalla produzione `E` e va letta come la successione della frase `true or false` (a sua volta una espressione `E`) e della frase `or false` (in cui `false` è un `F`).

Le produzioni sintattiche si possono riscrivere eliminando la ricorsione sinistra della prima produzione (a `VT` si aggiunge la variabile sintattica `RestE`):

```
P = {
    E ::=  F | F RestE
    RestE := or F | or F RestE
    F ::= true | false
  }
```

Questa riscrittura induce a pensare l'operatore `or` come associativo a destra; essa agevola tuttavia il progetto e la realizzazione di un riconoscitore sintattico con tecniche di analisi ricorsiva discendente.

1.1.3 Analisi lessicale

Il lessico delle frasi del linguaggio `L(E)` può essere fatto coincidere con il linguaggio `L(F)`. L'analizzatore lessicale (*lexer*) del linguaggio `L(E)` si può costruire facilmente in Java, utilizzando la classe `java.io.StreamTokenizer`:

```
String frase ="false or true or false";
java.io.StringReader strReader  = new java.io.StringReader( frase );
java.io.StreamTokenizer strTokenizer strTokenizer =
                new java.io.StreamTokenizer( strReader );
```

La invocazione

```
strTokenizer.nextToken();
```

provoca un'azione di analisi che memorizza in una variabile locale dell'analizzatore il primo token e restituisce un intero che rappresenta il tipo del token (a sua volta memorizzato nel campo `ttype`).

Il tokenizer Java

`strTokenizer.nextToken()` restituisce il valore intero `-1` se la frase è terminata o non vi sono token, il valore `-3` se individua una stringa e il valore `-2` se infividua un numero. Se il tipo del token è

- `StreamTokenizer.TT_WORD (-3)`, il token è un alfanumerico e il suo valore è memorizzato nel campo `sval` di tipo `String`;
- Se il tipo del token è `StreamTokenizer.TT_NUMBER (-2)`, il token è un numero e il suo valore è memorizzato nel campo `nval` di tipo `double`.

Pertanto, con riferimento alla frase `false or true or false` l'espressione:

```
(strTokenizer.sval != null) ?  strTokenizer.sval : "eof";
```

alla prima scansione denota il token `false` in forma di stringa. Iterando le invocazioni `strTokenizer.nextToken()`, la variabile `StreamTokenizer.sval` denoterà, in sequenza le stringhe `or`, `true`, `or`, `false` e infine `eof`.

L'analizzatore lessicale fornito da Java opera sulla base di standard comuni a quasi tutti i linguaggi. In particolare, se la frase di ingresso fosse stata `false -2 + 1.5` l'analizzatore lessicale avrebbe fornito, uno dopo l'altro i token `false` (in `sval`), `-2.0` (in `nval`), `43`(valore ASCII o UNICODE di `+` in `sval`) `1.5` (in `nval`).

Ovviamente vi sono modi per personalizzare il comportanento di oggetti della classe `java.io.StreamTokenizer`.

1.1.4 Prototipo Prolog di un lexer

Il codice *tuProlog* che segue crea un un lexer Java utilizzabile all'interno di un programma Prolog:

```
createTokenizer( S,Tokenizer ) :-
      java_object('java.io.StringReader',[S], Reader ),
      java_object('java.io.StreamTokenizer',[Reader], Tokenizer ).
```

La clausola `getToken` che segue permette di ottenere un `Token` da un `Tokenizer` (ovviamente l'invertibilità non è applicabile: rimane la sola intepretazione procedurale, in cui la variabile `Tokenizer` denota il valore di un argomento di ingresso e la variabile `Token` il valore di uscita).

```
%getToken( @Tokenizer, ?Token )
getToken( Tokenizer, Token )  :-
      Tokenizer <- nextToken returns TypeOfToken,
      resolveToken( Tokenizer, TypeOfToken, Token ).

resolveToken( Tokenizer, V, token(symbol,eof) ) :- V is -1, !.
resolveToken( Tokenizer, V, token(number,Token) ) :- V is -2, !,
      Tokenizer.nval <- get( Token ).     %get valore del  campo nval
resolveToken( Tokenizer, V , token(string,Token) ) :- V is -3,
      Tokenizer.sval <- get( TokenObj ), %aget valore del  campo sval
      TokenObj <- toString returns Token.

resolveToken( Tokenizer, V, token(symbol,V) ). %default
```

In questa versione si è scelta come rappresentazione del token di uscita la forma

```
token(Type,Value)
```

ove `Type`, che può assumere i valori `symbol` o `number`, denota il tipo del token e `Value` denota il suo valore.

L'ultima regola gestisce tutti i simboli riconosciuti dal lexer che non rientrano nelle categorie di interesse.

1.1.5 Generazione di una lista di token

Un primo collaudo dell'analizzatore può essere impostato definendo una clausola (procedura) che, dato un `Tokenizer` fornisce la lista dei token presenti nella frase di ingresso:

```
buildTokenList( Tokenizer, CurList, ListOfToken):-
 getToken( Tokenizer, Token ),
 (  Token = token(_,eof), !, reverse( CurList, ListOfToken );
     buildTokenList( Tokenizer, [Token|CurList], ListOfToken).
```

Nell'eseguire la procedura `buildTokenList` l'interprete Prolog effettua un *processo iterativo* in forma di *tail recursion*. Il ruolo di accumulatore del risultato è svolto dalla variabile `CurList` che si presuma assuma inizialmente il valore di lista vuota. Al termine della iterazione la lista risulta ordinata in senso inverso rispetto alla sequenza di ingresso; pertanto il risultato si ottiene effettuando l'operazione `reverse` di una lista:

```
reverse( L,R ) :-  reverse( L, [], R ).
reverse( [], R,R ).
reverse( [H|T], CurList, Res ):- reverse(T, [H|CurList],Res).
```

Collaudo

Il collaudo dell'analizzatore può essere impostato come segue:

```
testLex0 :-  %caso di stringa vuota
  createTokenizer( "",Tokenizer ),
  buildTokenList( Tokenizer, [ ], [ ] ).

testLex1 :-  %caso generico
  createTokenizer( "a -2 true",Tokenizer ),
  buildTokenList( Tokenizer, [ ], ListOfToken ),
  ListOfToken = [token(symbol,a),token(number,-2),token(symbol,true) ].

testLex :- testLex1, testLex2.
```

Numeri
In `testLex1`, se si visualizza la lista `ListOfToken` si ottiene

```
[token(symbol,a),token(number,-2.0),token(symbol,true)].
```

Ciò per due ragioni:

- il valore numerico restituito dal lexer è sempre di tipo `double`;
- il Prolog interpreta la notazione `-2` come la definizione di un numero (*meno due*) che può unificare con il numero reale di parte intera *due* e parte frazionario *zero*.

1.1.6 Analisi ricorsiva discendente

La produzione grammaticale `E`

```
P = {
    E ::=  F | F RestE
    RestE := or F | or F RestE
    F ::= true | false
  }
```

induce al seguente ragionamento: data una frase (stringa) `S`, questa appartiene al linguaggio `L(E)` se esiste un *prefisso* `SF` di `S` che appartiene al linguaggio `L(F)`. Se il prefisso `SF` è seguito dal simbolo `or`, allora `S` appartiene al linguaggio `L(E)` se dopo `or` esiste un altro segmento di frase che appartiene al linguaggio `L(F)` e così via.

Analisi top down ricorsiva discendente

La tecnica di analisi top-down ricorsiva discendente consiste nell'associare ad ogni produzione `P` una procedura di riconoscimento con la seguente signature :

```
ricP( Source, Rest, Result )
```

ove `Source` denota la frase di ingresso, `Rest` denota il segmento di frase che rimane dopo la scansione e `Result` il risultato della scansione.

La procedura ha l'obiettivo di determinare il più lungo prefisso di `Source` che costituisce una frase di `L(P)`, associando alla variabile `Rest` la frase che rimane.

Si noti che invocando la procedura nella forma `ricP(..., [ ],Result)` si impone il requisito che **tutta** la frase appartenda a `L(P)`.

Pro e contro della analisi ricorsiva discendente

La tecnica di associare una procedura ad ogni produzione può sembrare inutilmente onerosa per il caso corrente. Infatti il linguaggio `L(E)` può essere denotato dalla espressione regolare:

```
(true | false) (or (true | false))*
```

riconoscibile da un semplicissimo automa a stati finiti. Dunque l'onere di memoria e di tempo insito nel meccanismo di chiamata di procedura potrebbe essere evitato usando una delle molte tecniche per il progetto e la realizzazione di automi a stati finiti.

Tuttavia l'introduzione di una procedura di riconoscimento per ciascuna produzione permette di creare in modo immediato una soluzione software modulare e riusabile strettamente correlata alla specifica dei requisiti (le produzioni grammatcali) e cioè tale da poter essere modificata in modo semplice e comprensibile alla modifica dei requisiti stessi (cioè alla modifica delle produzioni).

1.1.7 Prototipo Prolog di un parser

Nel caso delle espressioni logiche definite dalla grammatica `G` introdotta nella sezione 1.1.2, pag. 7 si tratta di scrivere le procedure di riconoscimento della produzione `E` (`ricE`) e della produzione `F` (`ricF`) una volta che si sia stabilita la forma che si vuole far assumere al risultato.

In questa prima versione supporremo che il risultato consista in una riscrittura della frase in forma di *albero*, rappresentato dalla seguente struttura Prolog:

```
exp( op(Op), Opnd1, Opnd2 )
```

ove `Op` denota un operatore, `Opnd1` il primo operando e `Opnd2` il secondo operando. Ricordando che gli operandi sono particolari forme di espressioni, stabiliamo che la rappresentazione di un valore sia la struttura:

```
exp( Type, Value )
```

ove `Type` ne denota il tipo e `Value` il valore. Nel caso specifico `Type` può assumere un unico valore, che fissiamo nell'atomo `bool`.

Pianificazione del collaudo

In queste ipotesi, il risultato di un analizzatore sintattico (**parser**) per la frase `false or true or false` deve essere:

```
exp(op(or),exp(op(or),exp(bool,false),exp(bool,true)),exp(bool,false))
```

Si ricorda infatti che la grammatica originaria stabilisce l'associatività a sinistra dell'operatore `or`.

Per la frase `false or true and false or true` il risutato sarà:

```
exp(op(or),exp(bool,false),exp(bool,true)).
```

La scansione infatti si deve arresta al primo simbolo che non appartiene a `L(E)`, cioè `and`. Il parser lascia come resto non analizzato la lista di token corrispondente alla frase `and false or true`.

Impostazione del parser

La procedura di riconoscimento del linguaggio `L(E)` viene definita partendo dalla ipotesi che la frase di ingresso sia denotata da una lista di token prodotta utilizzando l'operazione `buildTokenList` definita nella sezione 1.1.5, pag. 10 :

```
% E ::=  F | F RestE
ricE( Source, Rest, Result ) :-
     ricF( Source, RestOfF, ResultOfF ),
     ricRestE( RestOfF, Rest, ResultOfF, Result ).
```

In pratica la prima produzione grammaticale è stata riscritta in forma di clausole Prolog.

La operazione `ricRestE` ha il compito di riconoscere la frase dopo il primo prefisso avendo memoria del risultato ottenuto fino a quel momento assicurando che nella forma del risultato l'operatore `or` risulti associativo a sinistra, come voleva la forma originaria della grammatica.

```
% RestE := or F | or F RestE

ricRestE( Source, Rest, CurRes, Result ) :-
% Source: frase corrente di ingresso
% Rest: parte della frase di ingresso che rimane dopo la scansione
% CurRes: risultato noto fino a questo momento
% Result: risultato finale al termine della scansione
     (   Source = [ token(symbol,or) | RestAfterOr ], !,
         ricF( RestAfterOr, RestAfterF, ResultOfF ),
         ricRestE(RestAfterF,Rest,exp(op(or),CurRes,ResultOfF),Result );
         Rest = Source, Result = CurRes
     ).
```

La struttura sintattica del codice è ricorsiva ma il processo computazionale è iterativo (ricorsione tail). Riferendoci alla interpretazione procedurale del Prolog, l'interprete alloca per la procedura `ricRestE` un solo record di attivazione.

Effettuando un ragionamento analogo sulla produzione `F`, il riconoscitore per `L(F)` avrà la forma:

```
% F ::= true | false
ricF(  [token(symbol,true)| Rest], Rest, exp(bool,true) ).
ricF(  [token(symbol,false)| Rest], Rest, exp(bool,false) ).
```

Collaudo

Impostiamo alcune operazioni di collaudo:

```
testParser1 :-%frase di un solo elemento
  createTokenizer( "false ",Tokenizer ),
  buildTokenList( Tokenizer, [ ], ListOfToken ),
  ricE( ListOfToken, [], Result ),
  Result = exp(bool, false).

testParser2 :-%frase generica senza resto
  Source = "false or true or false",
  createTokenizer( Source,Tokenizer ),
  buildTokenList( Tokenizer, [ ], ListOfToken ),
  ricE( ListOfToken, Rest, Result ),
  %writeln( answer(rest(Rest), result(Result)) ),
  Rest = [],
  Result =
  exp(op(or),exp(op(or),exp(bool,false),exp(bool, true)),exp(bool,false)).

testParser3 :-%frase generica con resto
  Source = "false and true",
  createTokenizer( Source,Tokenizer ),
  buildTokenList( Tokenizer, [ ], ListOfToken ),
  ricE( ListOfToken, Rest, Result ),
  Rest = [token(symbol,and),token(symbol,true)],
  Result =  exp(bool, false).
```

1.1.8 Produzioni con self embedding

Da un linguaggio per esprimere espressioni ci si aspetta la possibilità di introdurre parentesi per modificare l'ordine di valutazione degli operatori che compaiono nella frase. Questo obiettivo è ottenuto introducendo una nuova produzione grammmaticale a livello di fattore:

```
F ::= ( E )
```

L'inserimento di questa regola imnplica l'estensione di `VT` della grammatica `G` di sezione 1.1.2, pag. 7 con i due nuovi simboli terminali `(`, `)`. Le nuove produzioni grammaticali:

```
P = {
    E ::=  F | E or F
    F ::= true | false | ( E )
  }
```

permettono ora di scrivere frasi del tipo:

```
 true or (false or true)
```

che inducono a valutare il secondo `or` prima della valutazione del primo.

Grazie alla impostazione del parser suggerita dalla tecnica di analisi ricorsiva discendente e alla modularità intrinseca allo stile di programmazione dichiarativo dle Prolog, l'aggiornamento del prototipo per tenere conto della nuova regole è immediato.

Il lexer viene aggiornato inserendo prima delle altre le seguenti regole:

```
resolveToken( Tokenizer, 40, token(symbol,lp) ) :- !.  % (
resolveToken( Tokenizer, 41, token(symbol,rp) ) :- !.  % )
```

Per il parser basta aggiungere una regola specializzata per il caso delle parentesi a livello di riconoscimento dei fattori:

```
%F ::= ( E )
ricF( [token(symbol,lp)| RE ] , Rest, Result )  :-
    ricE( RE, [token(symbol,rp)|Rest], Result ).
```

La nuova versione del parser riconosce frasi con qualunque disposizione ben formata di parentesi, come ad esempio:

```
true or (true or false)
(true or (true or false))
(true or ((true or false)))
```

1.1.9 Operatore not

La produzione `F ::= ( E )` rivela la sua utlità quando si estendano le produzioni per permettere anche l'uso dell'operatore `not` che di norma è prioritario rispetto all'`or`:

```
P = {
    E ::=  F | F RestE
    RestE := or F | or F RestE
    F ::= true | false | ( E ) | not F
  }
```

Per tenere conto della nuova regola di produzione definiamo una nuova clausola di riconoscimento per `L(F)`:

```
ricF( [token(symbol,not)| RE ] , Rest, exp(bool,not(Result)) )  :-
    ricF( RE, Rest, Result ).
```

Collaudo

Definiamo una clausola che permetta la valutazioni di frasi immesse come ingresso:

```
testParser( Source ) :-
  createTokenizer( Source,Tokenizer ),
  buildTokenList( Tokenizer, [ ], ListOfToken ),
  ricE( ListOfToken, Rest, Result ),
  writeln( answer(rest(Rest), result(Result)) ).
```

Il goal `testParser( false or not true or true )` emette un messaggio così formato:

```
answer(
 rest([]),
 result(
 exp(op(or),
  exp(op(or),
        exp(bool,false),
        exp(bool,not(exp(bool,true) ))),
  exp(bool,true))
  ))
```

Il goal `testParser( false or not (true or true) )` emette un messaggio così formato:

```
answer(
 rest([]),
 result(
 exp(op(or),
   exp(bool,false),
```

```
   exp(bool,not(  exp(op(or),
                          exp(bool,true),
                          exp(bool,true) )
   )))
 ))
```

1.2 Impostazione del valutatore

1.2.1 Parser con uscita non cablata

Il codice Prolog che segue costituisce un raffinamento del parser sviluppato nella sezione precedente. La nuova versione mira a rendere il parser più flessibile riguardo alla costruzione della risposta.

Lo scopo viene raggiunto incapsulando la costruzione della risposta nella clausola `eval` che, dati come ingressi i token riconosciuti corretti dal parser, fornisce strutture di uscita diverse in funzione del valore corrente del fatto `mode/1` definito nella base di conoscenza.

Ad esempio se la base di conoscenza include il fatto

```
mode(postfissa).
```

il parser fornirà come risposta una rappresentazione in forma polacca postfissa della frase.

La clausola `eval` assume tre forme:

```
eval(Symbol,Result).                              %operando
eval(Operator,Operand,Result).                 %operatore unario
eval(Operator,Operand1,Operand2,Result). %operatore binario
```

Il parser come client del lexer

La nuova versione del parser non assume più di ricevere in ingresso tutta la frase in forma di lista di token, ma utilizza esplicitamente il *lexer* (invocando `getToken` definita nella sezione 1.1.4, pag. 9) per ottenere un token quando ne ha bisogno.

Poichè il lexer fa uso di un dispositivo di ingresso le cui operazioni di lettura non possono essere revocate una volta effettuate, nella clausola che rappresenta il parser viene introdotto l'argomento `CurToken` per denotare il token corrente e l'argomento `NextToken` per denotare il primo token non consumato. Ciascuna procedura di riconoscimento `P` ha la forma

```
ricP(@CurToken, @Tokenizer, ?Result, ?NextToken)
```

ove

- `CurToken` rappresenta il primo token disponibile, già prelevato dalla sorgente dati;
- `Tokenizer` rappresenta la sorgente dati, dotata di stato;
- `Result` rappresenta il risultato prodotto dal riconoscimento;
- `NextToken` rappresenta il primo token (già prelevato dalla sorgente dati) che il riconoscimento non ha coperto.

La procedura di riconoscimento procede consumando tutti i token della frase di ingresso che corrispondono a `L(P)`; la procedura `P` trasferisce ad eventuali procedure chiamate il valore del primo token non consumato.

```
%E ::=  F | F RestE
parserE( CurToken, Tokenizer, Result, NextToken ) :-
     parserF( CurToken, Tokenizer, ResultOfF, AfterFToken ),
     parserRestE( AfterFToken, Tokenizer, ResultOfF, Result, NextToken).

%RestE := or F | or F RestE
parserRestE( token(symbol,or), Tokenizer, CurResult, Result, NextToken ) :-
        getToken( Tokenizer, AfterOpToken ),
        parserF( AfterOpToken, Tokenizer, ResultOfF, AfterFToken ),
        eval( op(or), CurResult, ResultOfF, NewResult ),
        parserRestE( AfterFToken, Tokenizer, NewResult, Result, NextToken ).
parserRestE( CurToken, Tokenizer, CurResult, CurResult, CurToken ).

%F ::= ( E )
parserF(  token(symbol,lp), Tokenizer, ResultOfE, NextToken )  :- !,
    getToken( Tokenizer, AfterLpToken ),
    parserE( AfterLpToken, Tokenizer, ResultOfE,  token(symbol,rp) ),
    getToken( Tokenizer, NextToken ).

%F ::= not F
parserF(  token(symbol,not), Tokenizer, Result, NextToken )  :- !,
    getToken( Tokenizer, AfterNotToken ),
    parserF( AfterNotToken, Tokenizer, ResultOfF,  NextToken ),
    eval( op(not), ResultOfF, Result ).

%F ::= true | false
parserF(  CurToken, Tokenizer, Res, NextToken )  :-
     CurToken = token(symbol,SYM),
    (SYM = true,! ; SYM = false ),!,
    eval( token(symbol,SYM), Res ),
    getToken( Tokenizer, NextToken ).
```

Per verificare il comportamento del riconoscitore faremo uso della seguente clausola (procedura) di prova interattiva:

```
testParser( Mode, Source ) :-
    clean,
    assert( mode( Mode ) ),
    createTokenizer( Source,Tokenizer ),
    getToken( Tokenizer, CurToken ),
    parserE( CurToken, Tokenizer, Res, NextToken),
    writeln( result(Res, next(NextToken) ) ).

clean :- (retract( mode(_) ),!;true).
```

Nel codice che segue si definiscono tante versioni della clausola `eval` quanti i modi di valutazione usando il `cut` per impedire che l'interprete Prolog esplori inutilmente altre versioni di `eval` in caso di fallimento.

1.2.2 Forma polacca postfissa

```
% ------------------------------------------------------------
% Evaluation mode polacca postfissa
% ------------------------------------------------------------
eval(op(OP), V1, V2, R ) :- mode(postfissa), !,
     append(V1,V2,R1),
     append(R1,[OP],R).

eval(op(OP), V, R ) :- mode(postfissa), !,
     append(V,[OP],R).

eval( token(symbol,true),  [true]  ) :- mode(postfissa),!.
eval( token(symbol,false), [false]  ) :- mode(postfissa),!.
```

La invocazione

```
 testParser( postfissa, "false or not true or true" )
```

provoca la emissione del messaggio:

```
result(
 [false,true,not,or,true,or],
 next(token(symbol,eof))
)
```

La invocazione

```
testParser( postfissa, "false or not (true or true)" )
```

provoca la emissione del messaggio:

```
result(
  [false,true,true,or,not,or],
  next(token(symbol,eof))
)
```

La invocazione

```
testParser( postfissa, "not true and false)" )
```

provoca la emissione del messaggio:

```
result(
  [true,not],
  next(token(symbol,and))
)
```

1.2.3 Valutazione del valore della espressione

```
% ----------------------------------------------------------------
% Evaluation mode calcolo
% ----------------------------------------------------------------
eval(op(OP), bool(V1), bool(V2), bool(Result) ) :-
 mode(calcolo), !,
 evalOp(OP,V1,V2,Result).

eval(op(OP), bool(V1), bool(Result) ) :-
 mode(calcolo), !,
 evalOp(OP,V1,Result).

eval( token(symbol,true),  bool(true)   ) :- mode(calcolo),!.
eval( token(symbol,false),bool(false)  ) :- mode(calcolo),!.

evalOp(or,false,false,false):-!.
evalOp(or,_,_,true) :- !.
evalOp(not,true,false):-!.
evalOp(not,false,true):-!.
```

La invocazione

```
testParser( calcolo, "false or not true or true" )
```

provoca la emissione del messaggio:

```
result(bool(true),next(token(symbol,eof)))
```

La invocazione

```
testParser( calcolo, "false or not (true or true)" )
```

provoca la emissione del messaggio:

```
result(bool(false),next(token(symbol,eof)))
```

La invocazione

```
testParser( calcolo, "not true and false)" )
```

provoca la emissione del messaggio:

```
result(bool(false),next(token(symbol,and)))
```

1.2.4 Costruzione di un abstract parse tree

Una delle prime e più critiche decisioni di progetto di un sistema software riguarda il modo con cui rappresentare le informazioni del dominio applicativo. Quando queste informazioni sono le frasi di un linguaggio, una scelta può essere quella di rappresentare una frase con una stringa. Ma questa rappresentazione non sarebbe così espressiva come una rappresentazione ad albero. Infatti l'albero è una struttura bidimensionale che può esprimere ciò che la monodimensionalità delle stringhe rende meno immediato: i componenti elementari di una frasi e la loro relazione gerarchica.

Se infatti consideriamo la stringa `true or false or true` essa non ci dice nulla su come leggere la frase in termini strutturali. Già meglio sarebbe la stringa `(true or (false or true))`, in cui le parentesi tonde danno informazione su quale operatore debba essere eseguito per primo.

L'organizzazione strutturale di una frase è compiutamente catturata dall'albero di derivazione (detto anche *parse tree*) così organizzato per una grammatica `G=(VN,VT,S,P)`:

- ciascun nodo dell'albero corrisponde ad un simbolo del vocabolario $V = VN \cup VT$ della grammatica;

- la radice è lo scopo `S` della grammatica;
- se i nodi $A_1, A_2, ...A_k$ sono i figli ordinati di un nodo di etichetta A , allora $A ::= A_1, A_2, ...A_k$ è una produzione in `P` di `G`.

Sintassi astratta e albero sintattico astratto

Una rappresentazione delle frasi in termini di albero di derivazione è ridondante in quanto, come detto, parte della sintassi concreta non è legata a nulla di significativo sul piano semantico e in quanto i simboli non terminali, indispensabili per la costruzione, possono essere omessi nella rappresentazione interna delle frasi.

L'eliminazione dalla sintassi concreta di un linguaggio di tutti gli elementi inutili produce una descrizione del linguaggio più semplice, detta *sintassi astratta*. La rappresentazione della sintassi astratta di una frase espressa in forma di albero si dice *abstract parse tree* (**apt**).

In base a quanto detto fno ad ora, un *interprete* può essere definito come un automa che stabilisce la semantica operazionale di un linguaggio eseguendo le frasi del linguaggio descritte in termini di sintassi astratta.

La sintasi astratta può essere descritta ancora con notazioni `BNF`, riducendo le informazioni all'essenziale. Ad esempio, la sintassi astratta delle espressioni ogiche può essere definita dalle seguenti produzioni:

```
Exp ::= Exp or Exp  //Disgiunzione
Exp ::= Exp and Exp  //Congiunzione
Exp ::= not Exp //Negazione
```

Quste sono regole ambigue, non convenienti per la descrizione della struttura concreta frasi. Tuttavia, ai fini della specifica di una rappresentazione interna, esse esprimono gli elementi essenziali della struttura delle frasi.

Un **interprete** può essere definito come un automa che stabilisce la semantica operazionale di un linguaggio eseguendo le frasi del linguaggio descritte in termini di sintassi astratta.

Ciascun simbolo non terminale definisce un diverso *tipo di struttura*. La parte destra di ciscuna produzione indica un possibile modo di costruzione di quel tipo in termini di altri componenti. Nel caso specifico, una *Exp* può essere formata in tre modi diversi.

1.2.5 Uscita in forma di albero

Il codice che segue definisce la fase di valutazione dell'automa riconoscitore del linguaggio (parser) sviluppato nelle sezioni precedenti in modo che il risultato del riconoscimento sia un apt della frase. Gli apt sono frequentemente

usati per aumentare la modularità di compilatori ed interpreti, in quanto consentono di separare la parte di parsing dalla parte di analisi semantica (type-checking, generazione di codice, etc).

```
 % ----------------------------------------------------------------
 % Evaluation mode albero
 % ----------------------------------------------------------------
   eval( op(Op), Opnd1, Opnd2, exp( op(Op), Opnd1, Opnd2) ) :- mode(albero),!.
   eval( op(Op), Opnd, exp( op(Op), Opnd) ) :- mode(albero),!.
   eval( token(symbol,true),  exp(bool,true)   ) :- mode(albero),!.
   eval( token(symbol,false),exp(bool,false) ) :- mode(albero),!.
```

La invocazione

```
 testParser( albero, "false or not true or true" )
```

provoca la emissione del messaggio:

```
result(
  exp(op(or),
      exp(op(or),
         exp(bool,false),
         exp(op(not),
                    exp(bool,true))),
      exp(bool,true)),
  next(token(symbol,eof))
)
```

La invocazione

```
 testParser( albero, "false or not (true or true)" )
```

provoca la emissione del messaggio:

```
result(
   exp(op(or),
       exp(bool,false),
       exp(op(not),
           exp(op(or),
               exp(bool,true),
               exp(bool,true))
   )),
   next(token(symbol,eof))
)
 )
```

Le informazioni dello apt

L'esclusione dall'apt delle informazioni che costituiscono lo zucchero sintattico che rende più intellegibile il linguaggio, come i simboli di puteggiatura, le parole chiave, etc. rende più complicata la produzione di significativi messaggi di errore da parte del compilatore. Pertanto l'apt è di solito organizzato in modo da mantenere memoria nei vari nodi della posizione della frase iniziale dei caratteri cha hanno dato luogo alla struttura rappresentata dal nodo.

1.2.6 Apt come abstract data type

L'uso del Prolog ha indotto a risolvere il problema della costruzione dell'apt al problema della costruzione di una corrispondente struttura Prolog che assume una delle due seguenti forme:

```
exp( Value )
exp( Op, Opnd1, Opnd2 )
```

La riduzione della informazione di uscita ai termini Prolog è utile per la realizzazione di un primo prototipo che permetta di validare la correttezza della logica di elaborazione. Nel sistema finale è tuttavia preferibile costruire un albero i cui elementi siano istanze di tipi del dominio del problema e non del linguaggio di implementazione.

La definizione strutturale espressa dalle regole di produzione della sintassi astratta può essere traslata direttamente nella definizione di tipi strutturati di dati, seguendo una metodologia in cui:

- si introducono tanti tipi di dato astratto quanti i simboli non terminali;
- ciascuna tipo di dato astratto viene specializzata da uno o più sottotipi (concreti), tanti quante le regole di produzione;
- ogni simbolo (non terminale) che compare nella parte destra di una produzione definisce un componente della struttura del sottotipo che corrisponde a quella produzione.

Nel caso delle espressioni logiche esprimibili con la grammatica definita nella sezione 1.1.9, pag. 15 , il tipo di dato astratto *Exp* può essere concretizzato dai due sottotipi: `OrExp` e `NotExp`, cui si può pensare in prosettiva di aggiungere i sottotipi `AndExp` (introducendo l'operatore `and`), `ImplExp` (introducendo l'operatore `=>`) e coì via.

Utilizzando un linguaggio ad oggetti, ciascun tipo di dato astratto può essere espresso mediante una classe astratta (o da una interfaccia) tenendo conto che:

- La relazione di ereditarietà tra classi può essere usata per esprimere la relazione tipo-sottotipo.
- La struttura concreta dei sottotipi può essere ottenuta introducendo nelle classi corrispondenti tanti campi quanto i componenti del sottotipo.
- Per ciascuna classe concreta viene definito un costruttore che inizializza tutti i campi della classe;
- Le strutture vengono completamente inizializzate all'atto della costruzione e mai più modificate.

Ad esempio, nel caso della grammatica di sezione 1.1.9, pag. 15

```
P = {
   E ::=  F | E or F
   F ::= true | false | ( E ) | not F
 }
```

si possono introdurre le classi:

```
abstract class Exp{} //la classe-base della grammatica

abstract class OpExp extends Exp{
protected Exp left; Exp right;
protected OpExp( Exp l, Exp r){
// invariante: left e right sono sempre non null
left=l; right=r;}
abstract String myOp(); //ciascuna sottoclasse ne d una versione specializzata
public String toString(){
return left.toString() + myOp() + right.toString(); }
public String op(){ return myOp(); }
}


class OrExp extends OpExp{
public OrExp( Exp left, Exp right){ super(left,right);}
public String myOp() { return "or" ; }
}
```

La classe astratta `OpExp` è stata introdotta per fattorizzare le proprietà comuni alle frasi con operatore. Questa classe ridefinisce il metodo `toString()` in modo da permettere la visualizzazione degli oggetti in termini

di stringa. Il metodo `myOp` rappresenta il contenuto di conoscenza specifico di ogni sottoclasse di `OpExp`.

Ogni oggetto di classe `OpExp` ha la seguente proprietà strutturale (invariante di classe `IC`): i componenti `left` e `right` esistono sempre, cioè le corrispondenti variabili denotano riferimenti non `null` ad oggetti.

1.2.7 Altre architetture

Per ottenere dal parser diversi tipi di risposta sono utilizzabili anche altre tecniche, cui corrispondono diverse impostazioni architetturali del sistema di interpretazione. In particolare:

- nello stile funzionale si può introdurre un *argomento* in più (di nome `mode`) nella procedura di parsing. Il valore dell'argomento `mode` denota il tipo di risposta che si desidera e va trasmesso alla procedura eval, che può condizionare il suo comportamento in funzione del `mode` corrente;

- nello stile ad oggetti si può definire una *classe astratta* di valutazione e diverse specializzazioni, ciascuna delle quali costruisce uno specifico tipo di risposta. Il parser deve ricevere come informazione di ingresso un oggetto di valutazione istanza di una delle classi specializzate;

- la definizione della entità da usare come valutatore può essere definita in un *file di configurazione*. Il parser deve esplicitamente consultare questo file per ottenere il valutatore da usare;

- nello stile ad oggetti, in accordo al *pattern Factory*, si deifnisce una classe di costruzione di valutatori che il parser utilizza per ottenere una istanza dell'oggetti di valutazione da usare

Una ulteriore impostazione può essere quella di ottenere il programma Prolog come *configurazione dinamica di teorie*. Si supponga ad esempio di includere tutte le clausole relative al parsing in una base di conoscenza $Parse$ e tutte le clausole relative alla valutazione in una base di conoscenza $Eval$.

Il programma di riconoscimento e valutazione formato da $Parse \cup Eval$ darebbe le risposte relative alle conoscenze incluse in $Eval$. Risposte diverse si possono ottenere componendo uno stesso $Parse$ con teorie diverse di $Eval$.

Questa tecnica è stata studiata e realizzata nella *programmazione logica contestuale*.

Citiamo anche la possibilità di impostare il rapporto tra parser e valutatore seguendo lo schema del *pattern observer*. Questo tipo di architettura è oggi piuttosto diffusa e ad esso dedicheremo la sezione **??**, pag. **??** .

1.2.8 Produzione di un documento XML

In questa sezione definiamo una versione specializzata della clausola di valutazione `eval` in modo che l'uscita del parser sia un apt in forma di documento `XML` secondo lo standard `DOM` del `w3c`.

```
eval( op(Op), Opnd1, Opnd2, EXPNode ) :-
     mode(xml, Doc),
     text_term( Op, OpStr),
     Doc <- createElement( "EXP" ) returns EXPNode,
     EXPNode <- setAttribute( "type", op ),
     EXPNode <- setAttribute( "value", v(OpStr) ),
     EXPNode <- appendChild( Opnd1 ) returns ChildNode1,
     EXPNode <- appendChild( Opnd2 ) returns ChildNode2.

eval( op(Op), Opnd, EXPNode ) :-
     mode(xml, Doc),!,
     text_term( Op, OpStr),
     Doc <- createElement( "EXP" ) returns EXPNode,
     EXPNode <- setAttribute( "type", op ),
     EXPNode <- setAttribute( "value", v(OpStr) ),
     EXPNode <- appendChild( Opnd ) returns ChildNode.

eval( token(symbol,V), EXPNode   ) :-
     mode(xml, Doc),
     text_term( V, VStr),
     Doc <- createElement( "EXP" ) returns EXPNode,
     EXPNode <- setAttribute( "type", "bool" ),
     EXPNode <- setAttribute( "value", v(VStr) ).
```

Collaudo

```
cleanMode1:- (retract( mode(_) ),!;true).
cleanMode2:- (retract( mode(_,_) ),!;true).

clean :- cleanMode1, cleanMode2.

testParserXML(Source) :-
  clean,
  createDocument( xml, Doc),
  assert( mode( xml, Doc ) ),
  createTokenizer( Source,Tokenizer ),
  getToken( Tokenizer, CurToken ),
  parserE( CurToken, Tokenizer, Result, NextToken ),
  writeln( result(Result, next(NextToken)) ),
  Doc <- appendChild( Result ),
  showDoc( Doc ),
  showResultOfParsing( Result ).
```

Il predicato `showDoc/1` visualizza un documento rappresentato da un oggetto che realizza la interfaccia Java `org.w3c.dom.Document` invocando il metodo `showDocument` di istanza della classe `uniboEnv.DocTree`:

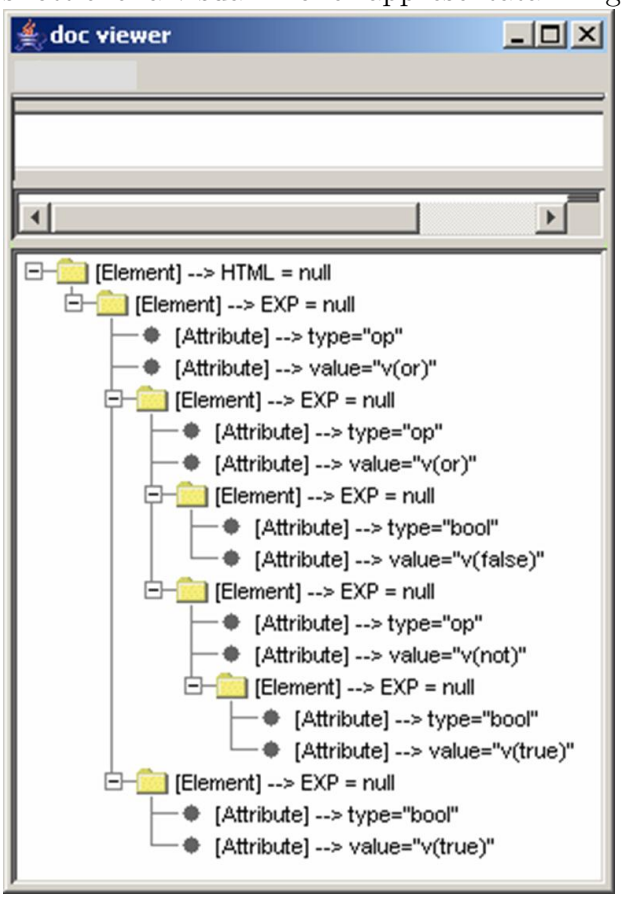
```
showDoc( Doc ):-
 java_object('uniboEnv.DocTree',["doc viewer"], DocTree ),
 DocTree <- showDocument( Doc ).
```

Esempio

Impostando la query

```
testParserXML("false or not true or true")
```

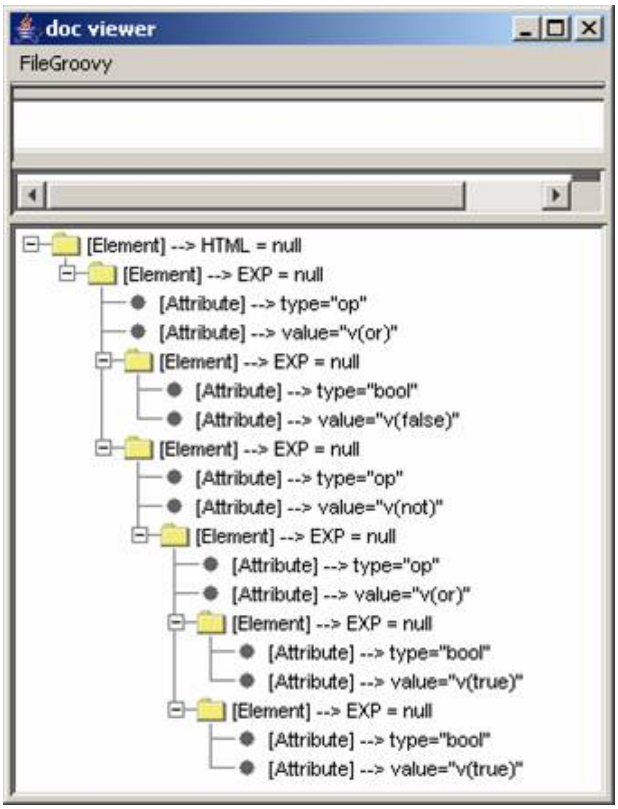
si ottiene la visualizzione rappresentata in figura:



Impostando la query

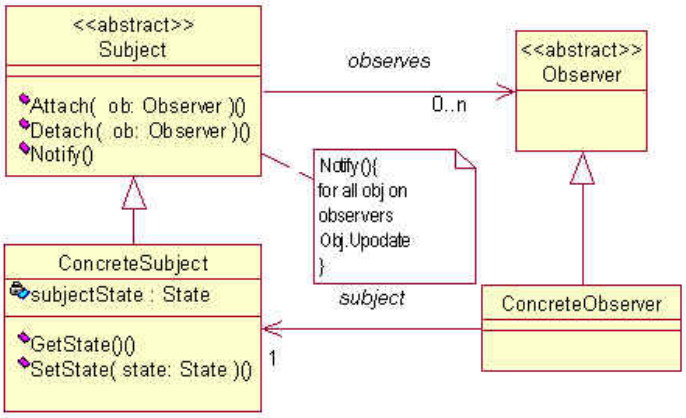
```
testParserXML("false or not (true or true)")
```

si ottiene la visualizzione rappresentata in figura:



1.2.9 Un parser generatore di eventi

Una impostazione oggi diffusa, soprattutto nel contesto del riconoscimento di testi scritti in `XML`, consiste nell'organizzare il rapporto tra parser e valutatore secondo lo schema di progettazione costituito dal *pattern observer*.



In questo schema il parser è concettualmente visto come un oggetto (`ConcreteSubject` del pattern) capace di generare *eventi* che possono venire percepiti da uno o più oggetti osservatori (`ConcreteObserver` del pattern, talvolta detti anche *listener*).

Associando (attraverso il metodo `Attach` del pattern) ad un parser più osservatori si possono effettuare valutazioni diverse in corrispondenza a una singola scansione sintattica, senza che il parser sia consapevole di quanti e quali siano i valutatori. Tenendo memoria degli eventi è possibile anche fare in modo che un valutatore sia temporalmente disaccoppiato dalla attività del parser.

Il codice che segue definisce la fase di `eval` in modo che la valutazione consista nella generazione di eventi che vengono memorizzati in forma di fatti in una base di conoscenza Prolog.

```
% ----------------------------------------------------------------
% Evaluation mode eventi
% ----------------------------------------------------------------
eval(op(OP),_,_,done ) :-
   mode(eventi,NameSpace,Engine ),!,
   generateEvent(Engine,event( NameSpace, op(OP))).
eval( token(symbol,true), done  ) :-
    mode(eventi, NameSpace, Engine ),!,
    generateEvent(Engine,event( NameSpace, bool(true))).
eval( token(symbol,false), done ) :-
    mode(eventi, NameSpace, Engine ),!,
    generateEvent(Engine,event(NameSpace,bool(false))).
```

Eventi come fatti di una base di conoscenza

Il predicato `generateEvent` è introdotto per incapsulare i dettagli di generazione di eventi.

Nel caso specifico, gli eventi sono espressi da clausole ground della forma:

```
event(NameSpace,Value)
```

La variabile `NameSpace` denota una struttura utile a partizionare gli eventi in gruppi diversi. La variabile `Value` denota il contenuto informativo associato all'evento.

Ammesso che la variabile logica `Engine` sia legata al riferimento a un oggetto Java di classe `alice.tuprolog.Prolog` che rappresenta un risolutore tuProlog, esso inserisce nella base di conoscenza un nuovo fatto, che rappresenta l'evento:

```
generateEvent( Engine, Event ) :-
     solve( Engine, G, Res  ).

solve( Engine, Goal, Res ) :-
    makeGoalStr( Goal, G  ),
    Engine <- solve( G ) returns Res.
```

I fatti che rappresentano gli eventi sono scritti nella sequenza in cui sono generati dal parser e quindi nell'ordine tipico della forma polacca postifissa. In particolare, per la frase

```
true or false or false
```

la teoria degli eventi generata è:

```
event(e5,bool(true)).
event(e5,bool(false)).
event(e5,op(or)).
event(e5,bool(false)).
event(e5,op(or)).
```

Il predicato `makeGoalStr` costruisce un goal nella forma attesa da `solve/1`, dato un termine Prolog (`Goal`):

```
makeGoalStr( Goal, GoalStr ):-
  text_term(GStr,Goal),  %text_term defined in tuProlog BasicLibrary
  text_concat(GStr,".",GoalStr). %text_concat defined in tuProlog BasicLibrary
```

Osservatori come enti autonomi

Un osservatore è concettualmente una entità interessata a essere informata ad ogni generazione di evento da parte di una determinata sorgente. Nel pattern *observer* l'interesse su un evento si concretizza nella invocazione del metodo `Attach` della sorgente da parte dell'osservatore, al fine di memorizzare presso la sorgente l'oggetto invocante (osservatore) come ente interessato a ricevere notifiche di uno specifico evento. Inoltre il pattern prevede che l'osservatore sia un oggetto capace di rispondere alla invocazione del metodo `update` da parte della sorgente, ogni qualvolta questa intende notificare un evento ai propri osservatori.

Nel caso sviluppato in questa sezione non occorre alcuna forma di generazione o di notifica. Gli eventi generati dalla sorgente (parser) sono infatti tradotti in assiomi di una teoria logica; chiunque abbia accesso a quella teoria può ricavarne informazioni effettuando deduzioni logiche.

La teoria degli eventi è costituita dall'insieme di clausole che forma la base di conoscenza associata al risolutore Prolog usato dal predicato `generateEvent`. Questi viene acquisito consultando il fatto:

```
mode(eventi, NameSpace, Engine )
```

che associa al modo di valutazione ad eventi uno specifico spazio di nomi e uno specifico risolutore.

Collaudo

Il codice che segue imposta una procedura di collaudo interattivo che esemplifica un possibile protocollo di uso:

```
testParserEventi( Source, Space ) :-
  clean,
  getEngineFromJava( Engine ),
  assertz( mode(eventi, Space, Engine ) ),
  createTokenizer( Source,Tokenizer ),
  getToken( Tokenizer, CurToken ),
  parserE( CurToken, Tokenizer, Result, NextToken ),
  showEvents( Engine, Space ).

clean :- cleanMode1, cleanMode2, cleanMode3.

cleanMode1:- (retract( mode(_) ),!;true).
cleanMode2:- (retract( mode(_,_) ),!;true).
cleanMode3:- (retract( mode(_,_,_) ),!;true).
```

La clausola:

```
getEngineFromJava( Engine ):-
  class('uniboEnv.JavaToProlog') <- getNewEngine returns Engine.
```

costituisce una utility per acquisire un risolutore Prolog diverso da quello utilizzato per il sistema di interpretazione. Il metodo `getNewEngine` della classe `uniboEnv.JavaToProlog` è definito in modo da creare un nuovo risolutore o da restituire un risolutore creato in precedenza:

```
public static Prolog getNewEngine(){
 if (newEngine == null) newEngine = new Prolog();
 return newEngine;
}
protected static Prolog newEngine = null;
```

Visualizzazione degli eventi

La clausola `showEvents` costituisce una utility per visualizzare gli eventi generati:

```
showEvents( Engine, NameSpace ) :-
  writeln( "showEvents starts ----------- " ),
  solve( Engine, event( NameSpace, Value ), SolInfo),!,
  SolInfo <- getSolution returns Res,
  Res <- toString returns R,
  writeln( R ),
  showNextEvents( Engine ).

showNextEvents( Engine ) :-
  Engine <- solveNext returns SolInfo,
  SolInfo <- getSolution returns Res,
  java_object_string(Res,R),
  writeln( R ),
  showNextEvents( Engine ).

showNextEvents( Engine ) :-
  writeln( "showEvents ends ----------- " ).
```

La clausola `showNextEvents` si occupa di visualizzare gli eventi successivi al primo, impostando una iterazione di invocazione al metodo `solveNext()` del risolutore tuProlog la cui ri-esecuzione non può essere ottenuta tramite backtracking essendo esso un predicato extra logico.

Una clausola quale:

```
allEvents( Engine, R ) :-
  solve(Engine,
            bagof(event(NameSpace,Ev),event(NameSpace,Ev),All),
            SolInfo),
  SolInfo <- getSolution returns Sol,
  Sol <- toString returns R.
```

permette di costruire una lista di tutti gli eventi presenti in una teoria degli eventi.

1.2.10 Agenti di valutazione

Osservatori come valutatori

Una volta disponibile una teoria degli eventi si possono definire osservatori che effettuano le diverse forme di valutazione definite in precedenza. Ad esempio, la logica di elaborazione di un osservatore che costruisce la forma polacca postfissa può essere definita dalla clausola che segue;

Progetto individuale

Questa ed altra forme di osservtaori-valutatori costituiscono materia per un possibile progetto finale da concordare con il docente in sostitutizione della prova standard di esame.

1.3 Espressioni artimetico logiche

1.3.1 Operatore and

Estendiamo la grammatica delle espressioni logiche in modo da permettere di esprimere anche operatori di congiunzione logica (`and`):

```
P = {
    E1 ::=  T | E1 or T
    T ::=  F | T and F
    F ::= true | false | ( E ) | not F
  }
```

Nel linguaggio `L(E1)` una espressione è formata da un termine (`T`) oppure da una espressione seguita da un `or` seguito da un termine. Un termine è formato da un fattore (`F`) oppure da un termine seguito da un `and` seguito da un fattore. Gli operatori `or` e `and` sono associativi a sinistra. L'operatore `and` è prioritario rispetto all'operatore `or` in quanto la frase;

```
true or false and true
```

viene derivata (derivazione canonica sinistra) come segue:

```
E1 -> E1 or T -> T or T -> F or T ->
true or T -> true or T and F -> true or F and F ->
true or false and F ->  true or false and true
```

La frase

```
(true or false) and true
```

viene riconosciuta da una derivazione che inizia con la riscrittura

```
E1 -> T -> T and F
```

In questa riscrittura `T` corrisponde alla frase `(true or false)` che, costituendo il primo operando dell'operatore `and`, deve essere valutata per prima. Se il linguaggio, come spesso accade, adotta una forma di *short circuit evaluation*, il secondo operando dell'`and` non viene nemmeno valutato se la valutazione del primo operando dà il risultato `false`,

Grazie alla impostazione del parser suggerita dalla tecnica di analisi ricorsiva discendente, l'aggiornamento del prototipo per tenere conto delle nuove regole è immediato, una volta riscritta la grammatica nella forma che elimina la ricorsione sinistra:

```
P = {
    E ::=  T | T RestE
    RestE := or T | or T RestE
    T ::=  F | F RestT
    RestT := and T | and T RestT
    F ::= true | false | ( E )  | not F
  }
```

Il parser non subisce alcuna modifica strutturale; basta tenere conto delle nuove produzioni:

```
%E ::=  T | T RestE
parserexpE( CurToken, Tokenizer, Result, NextToken ) :-
     parserexpT( CurToken, Tokenizer, ResultOfF, AfterFToken ),
     parserexpRestE( AfterFToken, Tokenizer, ResultOfF, Result, NextToken).

%RestE := or T | or T RestE
parserexpRestE( CurToken, Tokenizer, CurResult, Result, NextToken ) :-
        isoperator(CurToken, op(or)), !,
        getToken( Tokenizer, AfterOpToken ),
```

```
        parserexpT( AfterOpToken, Tokenizer, ResultOfT, AfterTToken ),
        eval( op(or), CurResult, ResultOfT, NewResult ),
        parserexpRestE( AfterTToken, Tokenizer, NewResult, Result, NextToken ).
parserexpRestE( CurToken, Tokenizer, CurResult, CurResult, CurToken ).

%T ::=  F | F RestT
parserexpT( CurToken, Tokenizer, Result, NextToken ) :-
     parserexpF( CurToken, Tokenizer, ResultOfF, AfterFToken ),
     parserexpRestT( AfterFToken, Tokenizer, ResultOfF, Result, NextToken).

%RestT := and F | and F RestT
parserexpRestT( CurToken, Tokenizer, CurResult, Result, NextToken ) :-
        isoperator(CurToken, op(and)), !,
        getToken( Tokenizer, AfterOpToken ),
        parserexpF( AfterOpToken, Tokenizer, ResultOfF, AfterFToken ),
        eval( op(and), CurResult, ResultOfF, NewResult ),
        parserexpRestT( AfterFToken, Tokenizer, NewResult, Result, NextToken ).
parserexpRestT( CurToken, Tokenizer, CurResult, CurResult, CurToken ).

%F ::= ( E )
parserexpF(  token(symbol,lp), Tokenizer, ResultOfE, NextToken )  :- !,
    getToken( Tokenizer, AfterLpToken ),
    parserexpE( AfterLpToken, Tokenizer, ResultOfE,  token(symbol,rp) ),
    getToken( Tokenizer, NextToken ).

%F ::= not E
parserexpF(  token(symbol,not), Tokenizer, ResultOfE, NextToken )  :- !,
    getToken( Tokenizer, AfterNotToken ),
    parserexpE( AfterNotToken, Tokenizer, ResOfE,  NextToken ),
    eval( op(not), ResOfE, ResultOfE ).

%F ::= true | false
parserexpF(  CurToken, Tokenizer, Res, NextToken )  :-
    CurToken = token(symbol,SYM),
    (SYM = true,! ; SYM = false ),!,
    eval( token(symbol,SYM), Res ),
    getToken( Tokenizer, NextToken ).
```

Le valutazioni devono essere aggiornate introducendo le regole per i nuovi operatori. Nel caso specifico:

```
%Valutazione che determina il valore
evalOp(and,true,true,true):-!.
evalOp(and,_,_,false) :- !.
```

1.3.2 Un parser a livelli

Le produzioni grammaticali delle espressioni artimetico-logiche

```
E ::=  T | T RestE
RestE := or T | or T RestE                %livello 2
T ::=  F | F RestT
RestT := and T | and T RestT          %livello 1
F ::= true | false | ( E ) | not F           %livello 0
```

presentano una caratteristica ripetitva a livelli, in cui ogni livello superiore al livello 0 è strutturalmente simile agli altri; unica differenza consiste nell'operatore caratteristico di quel livello. Tenendo conto di ciò, si può organizzare un parser top-down ricorsivo discendente in modo più compatto di quanto fatto fino ad ora:

```
exp( CurToken, Tokenizer, Res, NextToken  ) :-
            subexp( 2, CurToken, Tokenizer, Res, NextToken ).

subexp( N, CurToken, Tokenizer, Res, NextToken ) :-
 N > 0,!,
 N1 is N - 1,
 subexp( N1, CurToken, Tokenizer, ResTemp, NextTokenTemp ),
 restexp( N, NextTokenTemp, Tokenizer, ResTemp, Res,NextToken ).

% F ::= ( E )
subexp( 0, token(symbol,lp), Tokenizer, ResultOfE, NextToken ):- !,
    getToken( Tokenizer, AfterLpToken ),
    exp( AfterLpToken, Tokenizer, ResultOfE,  token(symbol,rp) ),
    getToken( Tokenizer, NextToken ).

%F ::= true | false
subexp(  0, token(symbol,SYM), Tokenizer, Res, NextToken )  :-
  (SYM = true,! ; SYM = false ),!,
  eval( token(symbol,SYM), Res ),
  getToken( Tokenizer, NextToken ).

%RestE := OP X | OP X RestE
restexp( N, token(symbol,Op), Tokenizer, CurResult, Res, NextToken ) :-
 operatore(N, Op, RepOp), !,
 getToken( Tokenizer, AfterOpToken ),
 N1 is N - 1,
 subexp( N1,AfterOpToken, Tokenizer, Res1, NextToken1 ),
 eval( RepOp, CurResult, Res1, NewRes ),
 restexp( N, NextToken1, Tokenizer, NewRes, Res, NextToken ).

restexp( _, CurToken, Tokenizer, CurResult, CurResult, CurToken ).
```

```
/* ============================    */
/* LIVELLI DEGLI OPERATORI                    */
/* ============================  */
operatore( 1, and, op(and )  ).
operatore( 2, or,  op(or)  ).
operatore( 0, lp,  op(lp)  ).
operatore( 0, rp,  op(rp)  ).
```

1.3.3 Procedure di collaudo del secondo ordine

Il codice che segue imposta procedure di collaudo interattivo che ricevono come argomento il funtore del parser da usare.

Produzione di un apt

```
testExpTree(Source, Parser) :-
    clean,
    assert( mode( albero ) ),
    createTokenizer( Source,Tokenizer ),
    getToken( Tokenizer, CurToken ),
    writeln(firstToken(CurToken) ),!,
    Goal =.. [Parser,CurToken, Tokenizer, Result, NextToken],
    Goal,
    writeln( result(Result, next(NextToken)) ).
```

Produzione del valore

```
testExpValue( Source, Parser ) :-
  clean,
  assert( mode(calcolo) ),
  createTokenizer( Source,Tokenizer ),
  getToken( Tokenizer, CurToken ),
  Goal =.. [Parser,CurToken, Tokenizer, Result, NextToken],
  Goal,
  writeln( answer(Result, NextToken ) ).
```

Produzione di un apt XML

```
 testExpXML(Source, Parser) :-
  clean,
  createDocument( xml, Doc),
  assert( mode( xml, Doc ) ),
  createTokenizer( Source,Tokenizer ),
  getToken( Tokenizer, CurToken ),
```

```
  Goal =.. [Parser,CurToken, Tokenizer, Result, NextToken],
  Goal,
  writeln( result(Result, next(NextToken)) ),
  Doc <- appendChild( Result ),
  showDoc( Doc ),%visualizzazione grafica
  showResultOfParsing( Result ). %visualizzazione Prolog
```

Produzione di eventi

```
testExpEventi( Source, Space, Parser ) :-
  clean,
  createObserver( Space, Engine ),
  createTokenizer( Source,Tokenizer ),
  getToken( Tokenizer, CurToken ),
  Goal =.. [Parser,CurToken, Tokenizer, Result, NextToken],
  Goal,
   showAnswer( Source, Space ).

showAnswer( Source, Space ) :-
  writeln( "======================== " ),
  %mode( eventi, Space, Engine ),  %funziona solo on the fly
  getEngineFromJava( Engine ),
  solve( Engine, showEvents(Space), Result ),
  writeln( "======================== " ).
```

Possibili procedure di attivazione nel caso dell'ultima versione del parser, che permette la gestione di operatori logici sono:

```
expTree( Source ):-  testExpTree( Source, exp).
expValue( Source ):-  testExpValue( Source, exp).
expXML( Source ):-  testExpXML( Source, exp).
expEvent( Source ):-  testExpEventi( Source, exp).
```

Possibili procedure di attivazione nel caso della versione del parser che gestisce le sole espressioni logiche `and-or-not` sono:

```
andorExpTree( Source ):-  testExpTree( Source, parserexpE).
andorExpValue( Source ):-  testExpValue( Source, parserexpE).
andorExpXML( Source ):-  testExpXML( Source, parserexpE).
andorExpEvent( Source ):-  testExpEventi( Source, parserexpE).
```

1.3.4 Espressioni aritmetiche

In questa sezione estendiamo il sistema di riconoscimento e valutazione definito nelle sezioni precedenti in modo che sia possibile riconoscere e valutare **anche** frasi generate dalle seguenti produzioni grammaticali *context free*:

```
P = {
    E ::=  T | E + T | E - T
    T ::=  F | T * F | T / F
    F ::= + F | - F | ( E ) | N
    N ::=   ....
  }
```

I simboli terminali `+`, `-`, `*`, `/` denotano i convenzionali operatori binari di somma, differenza, prodotto e divisione. I simboli `+`, `-` denotano anche il segno di un fattore `F`.

Il tokenizer

La variabile sintattica `N` denota un numero intero o reale, che viene già riconosciuto dall'analizzatore lessicale. L'unica avvertenza è di definire il *tokenizer* Java in modo che possa trattare il simbolo `/` (codice `ASCII` o `UNICODE` `47`) come un carattere ordinario:

```
createTokenizer( S,Tokenizer ) :-
      java_object('java.io.StringReader',[S], Reader ),
      java_object('java.io.StreamTokenizer',[Reader], Tokenizer ),
      Tokenizer <- ordinaryChar(47).
```

Il lexer

L'analizzatore lessicale deve definire clausole specializzate per gestire i nuovi simboli terminali:

```
resolveToken(_, 42, token(symbol,mulop) ) :- !. % *
resolveToken(_, 43, token(symbol,plusop) ) :- !. % +
resolveToken(_, 45, token(symbol,minusop) ) :- !. % -
resolveToken(_, 47, token(symbol,slashop) ) :- !. % /
```

La valutazione

Le diverse forme di valutazione si ottengono introducendo regole specializzate:

```
% ------------------------------------------------------------
% Valutazione dell'apt
% ------------------------------------------------------------
```

```
eval( token(number,N),  exp(num,N)   ) :- mode(albero),!.


% ------------------------------------------------------------
% Valutazione della forma polacca postfissa
% ------------------------------------------------------------
eval( token(number,N),  [N]   ) :- mode(postfissa),!.
```

Il parser

il parser a livelli definito in 1.3.2, pag. 37 è già strutturalmente adeguato a gestire le regole di produzione riscritte per evitare la ricorsione a sinistra:

```
P = {
    E ::=  T | T RestE
    RestE := + T | + T RestE  | - T | - T RestE
    T ::=  F | F RestT
    RestT := * T | * T RestT  | / T | / T RestT
    F ::= +F | -F | ( E ) | N
  }
```

Occorre però modificare (estendere) le regole per la gestione degli operatori unari:

```
% operatori unari (not F, + F, - F )
subexp( 0, token(symbol,SYM), Tokenizer, Result, NextToken ):- !,
    (SYM = not,!; SYM = plusop,!;SYM = minusop,!),
    getToken( Tokenizer, AfterUnOpToken ),
    subexp( 0, AfterUnOpToken, Tokenizer, ResultOfF,  NextToken ),
    eval( op(SYM), ResultOfF, Result ).
```

e introdurre nuove regole per la gestione di numeri:

```
% F ::= N
subexp(  0, token(number,N), Tokenizer, Result, NextToken )  :-
  eval( token(number,N), Result ),
  getToken( Tokenizer, NextToken ).
```

1.3.5 Calcolo del valore della espressione

Per ottenere in uscita il valore della espressione data come ingresso, occorre definire nuove regole per la valutazione di operatori binari ed unari:

```
% ------------------------------------------------------------
% Valutazione del risultato
% ------------------------------------------------------------
eval(op(OP), num(V1), num(V2), num(Result) ) :-
 mode(calcolo), !,
 evalOpNum(OP,V1,V2,Result).

eval(op(plusop),N, N):-  mode(calcolo),!.
eval(op(minusop),num(V1), num(R)):-
     mode(calcolo),!,
     R is 0 - V1.

eval( token(number,N),  num(N)  ) :- mode(calcolo),!.
```

Le regole specializzate relative agli operatori gestiscono il caso in cui la divisione abbia come denominatore `0`. In tal caso viene restituito il valore convenzionale `nAn` che denota un *non-numero*:

```
evalOpNum(_,_,nAn,nAn):-!.
evalOpNum(_,nAn,_,nAn):-!.
evalOpNum(plus,V1,V2,R):-!, R is V1 + V2.
evalOpNum(minus,V1,V2,R):-!, R is V1 - V2.
evalOpNum(mul,V1,V2,R):-!, R is V1 * V2.
evalOpNum(div,V1,0,nAn):-!.
evalOpNum(div,V1,V2,R):-!, R is V1 / V2.
```

1.3.6 Problemi di semantica

Sottoponendo al riconoscitore esteso il goal

```
testExpValue( "2 + 3 * 6 - (4 - 2) - 1", exp )
```

il sistema si pone in modo valutazione (`mode(calcolo)`) e fornisce come risposta il valore `17`.

Sottoponendo al riconoscitore esteso il goal

```
testExpTree( "2 + 3 * 6 - (4 - 2) - 1", exp )
```

il sistema si pone in modo valutazione (`mode(albero)`) e fornisce come risposta la struttura:

```
exp( op(minus),
       exp( op(minus),
                exp(  op(plus),
                         num(2.0),
```

```
                        exp(  op(mul),num(3.0),num(6.0))),
                exp(op(minus),num(4.0),num(2.0))),
        num(1.0) )
```

Il riconoscitore accetta come sintatticamente corretta anche una frase del tipo:

```
2 + 3 * true - (4 - false) - 1
```

Infatti

```
testExpTree( "2 + 3 * true - (4 - false) - 1", exp )
```

fonisce in risposta la struttura:

```
exp( op(minus),
        exp( op(minus),
                exp( op(plus),
                        num(2.0),
                        exp(op(mul),num(3.0),exp(bool,true))),
                exp(op(minus),num(4.0),
                exp(bool,false))),
        num(1.0))
```

Tuttavia invocando `testExpValue(2 + 3 * true - (4 - false) - 1)` si ottiene un fallimento.

La fase di analisi semantica è dunque implicitamente contenuta nelle clausole di valutazione degli operatori che sono definite in modo da applicare gli operatori logici (`and`, `or`, `not`) a operandi della forma `exp(bool,V)`e gli operatori algebrici a operandi della forma `exp(num,V)`. In un sistema di interpretazione reale, la parte di analisi semantica andrebbe sviluppata in modo più organico, tenendo conto delle regole che fomano il *type system* del linguaggio.

Capitolo 2

La semantica

2.1 Introduzione

2.1.1 Descrizioni della semantica

Per quanto riguarda la caratterizzazione della semantica di un linguaggio di programmazione, il metodo che viene spontaneo alla mente è quello **operazionale**: visto che un linguaggio di programmazione serve per descrivere azioni eseguibili da una macchina, la semantica operazionale può venire descritta scegliendo un automa esecutore e definendo il comportamento dell'automa in corrispondenza ad ogni classe di frasi del linguaggio.

Ma la struttura e il comprotamento dell'automa esecutore devono venire espresse a loro volta in modo formale, usando un linguaggio. Dunque come stabilire la semantica del linguaggio con cui descrivere l'automa? Per evitare un regresso all'infinito si deve assumere un livello di formalizzazione primitivo il cui significato è lasciato all'accordo intuitivo (ammesso che ciò sia possibile) tra chi legge e chi scrive, facilitato dalla semplicità delle mosse dell'automa prescelto.

L'approccio operazionale è tuttavia da molti criticato in quanto troppo di basso livello, poichè consente di pervenire alla comprensione di un costrutto linguistico solo dopo una analisi dettagliata e spesso complicata. Molti preferiscono definire il significato di un costrutto linguistico stabilendo una *funzione di interpretazione* che associ a quel costrutto una funzione matematica che denota una specifica relazione ingresso-uscita. Questo approccio, detto **denotazionale**, consente di caratterizzare la semantica con un maggior livello di astrazione rispetto all'approccio operazionale in quanto astrae dalla nozione di stato.

Una prospettiva ancora diversa è quella che imposta la semantica di un linguaggio in termini di una *teoria* dei programmi scritti in quel linguaggio.

Questo approccio, detto **assiomatico**, eleva ancora più il livello di astrazione in quanto si prefigge di costruire una teoria matematica per quel linguaggio, cioè un sistema formale nel quale è possibile esprimere fatti interessanti intorno ai programmi e dimostrarne o confutarne formalmente la verità. Il metodo assiomatico risulta particolarmente attraente dal punto di vista dell'ingegneria del software in quanto il programmatore non è tanto interessato ad avere un modello formale dei programmi che scrive, quanto a ragionare sui programmi e provarne proprietà.

L'analizzatore semantico

Un *analizzatore semantico* è un componente che, data la rappresentazione intermedia prodotta dall'analizzatore sintattico, controlla la corenza logica interna del programma (ad esempio se le variabili sono usate dopo essere state definite, se sono rispettate le regole di compatibilità in tipo, etc). Nel caso di un compilatore, l'analizzatore semantico può trasformare la rappresentazione delle frasi in modo più adatto per la generazione di codice.

2.1.2 Sematica denotazionale

L'approccio denotazionale mira a stabilire il significato delle frasi di un linguaggio introducendo opportune funzioni di interpretazione per stabilire la corrispondenza tra ogni categoria sintattica e il dominio inteso.

Poichè il concetto di funzione è noto dalla matematica, è possibile fare un esempio di questo metodo nel caso della grammatica delle espressioni definita nel capitolo precedente (sezione 1.3.4, pag. 40). Le categorie sintattiche sono qui le frasi generate dalle produzioni relative ai non-terminali `E`, `T` e `F`. Le funzione di interpretazione possono essre definite come segue (i simboli `+, -, *, /` denotano ora le operazioni di somma, sottrazione, prodotto e divisione tra numeri interi):

```
E_toNAT[ T ]            = T_toNAT[ T ]
E_toNAT[ E + T ]     = E_toNAT[ E ] + E_toNAT[ T ]
E_toNAT[ E - T ]      = E_toNAT[ E ] - E_toNAT[ T ]
T_toNAT[ T ]            = F_toNAT[ F ]
T_toNAT[ T * F]        = T_toNAT[ T ] * T_toNAT[ T ]
T_toNAT[ T / F]        = T_toNAT[ T ] / T_toNAT[ T ]
F_toNAT[ E ]            = E_toNAT[ E ]
F_toNAT[ N ]            = valueOf( N )
```

Il concetto alla base di queste definizioni è di specificare un comportamento per ogni possibile struttura di frase. Nel caso una espressione abbia la forma `E+T`, la funzione di interpretazione da eseguire sarà:

```
E_toNAT[ E + T ] = E_toNAT[ E ] + E_toNAT[ T ]
```

Questa funzione stabilisce che il valore denotato dalla espressione che compare prima del simbolo + deve essere sommato al valore del termine che compare dopo il simbolo +. Poichè un termine è definito come un fattore o un prodotto di fattori, eventuali moltiplicazioni presenti nella espressione verranno eseguite prima delle operazioni di somma.