Università degli Studi di Napoli Federico II

Giulio Iannello

Dispense di

Algoritmi e Strutture Dati

Febbraio 2004

Indice

Notazioni e convenzioni

Simboli della logica dei predicati del I ordine

$\wedge$	congiunzione
$\vee$	disgiunzione
$\neg$	negazione
$\Rightarrow$	implicazione
$\Leftrightarrow$	equivalenza
$\forall$	quantificatore universale
$\exists$	quantificatore esistenziale

Pseudo codice

Lo pseudo codice usate nelle presenti dispense è sostanzialmente quello usato nel libro:

> T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stain, Introduction to algorithms - second edition, MIT Press, 2001.

Sono state tuttavia aggiunte le seguenti convenzioni, in parte mutuate dalla sintassi del linguaggio Pascal:

1. Le condizioni sono scritte usando i simboli della logica dei predicati del primo ordine (vedi sopra).
2. Le istruzioni possono essere separate da un punto e virgola; nel caso due istruzioni siano scritte sulla stessa linea tale separazione è obbligatoria per evitare ambiguità, altrimenti verrà generalmente omessa.
3. I commenti sono generalmente racchiusi tra parentesi graffe.
4. Gli algoritmi di analisi lessicale acquisiscono l'input operando su un oggetto di tipo *text file*, che rappresenta una sequenza di caratteri terminata da un informazione speciale denominata *end-of-file*.

   Un text file T ha uno stato interno che identifica il *carattere corrente*, cioè il carattere su cui esso è posizionato. Inizialmente, un text file è posizionato sul primo carattere della sequenza di caratteri che lo costituisce.

   Su un text file opera Il sottoprogramma GET(T), che avanza di una posizione il carattere corrente di T.

   Un text file T è infine caratterizzato dagli attributi:

   $eof[T]$ è uguale a TRUE se T è posizionato sull'end-of-file, è uguale a FALSE altrimenti

   $next[T]$ è uguale al carattere corrente; il valore dell'attributo è indefinito quando $eof[T] =$TRUE
5. Nella descrizione degli algoritmi di analisi lessicale viene usato il costrutto **case** che permette di effettuale una selezione tra n alternative. Il costrutto ha la forma:

   case espressione **of**
   valore costante: sequenza di istruzioni
   . . .
   valore costante: sequenza di istruzioni

   e corrisponde ad eseguire esclusivamente la sequenza di istruzioni corrispondente al valore dell'espressione.

Capitolo 3

Algoritmi per l'analisi lessicale (bozze, v. 1.0)

Con il termine *analisi lessicale* si intende il riconoscimento, all'interno di una sequenza di caratteri, di elementi lessicali elementari detti *token*, costituiti ciascuno da una opportuna sequenza di caratteri. Le sequenze di caratteri che formano token validi sono definite da un insieme di *regole lessicali*, che possono essere fornite sia tramite una descrizione informale, che tramite strumenti formali.

L'analisi lessicale è alla base di tutte le forme di elaborazione di testi e in particolare costituisce la prima fase dell'analisi effettuata dai traduttori di linguaggi formali. In questo capitolo illustreremo un metodo per rappresentare in modo semi formale le regole lessicali e per derivare da tale rappresentazione un *analizzatore lessicale* o *scanner*, cioè un programma che scandisce la sequenza di caratteri in ingresso riconoscendo uno alla volta i token presenti nella sequenza.

3.1 Un esempio

Consideriamo il seguente problema:

> *Data in ingresso una sequenza di caratteri terminata dal carattere di fine-file, raggruppare i caratteri in parole costituite da sequenze di lettere dell'alfabeto inglese eventualmente seguite da un accento acuto (') o grave (') nel caso esse terminino con una vocale. Eventuali accenti che seguono una consonante non fanno parte della parola. Le parole possono essere separarte da qualsiasi carattere che non sia una lettera (si noti che nel caso una parola termini con una vocale e un accento, la parola successiva può iniziare immediatamente dopo l'accento).*

Il problema posto richiede di riconoscere particolari sequenze di caratteri (nel nostro caso le parole) in base a determinate regole di aggregazione descritte in modo informale. Una tale situazione si configura evidentemente come un problema di analisi lessicale.

Il primo passo da compiere consiste nel fornire una descrizione più precisa delle regole lessicali che ci permetta di derivare un algoritmo di riconoscimento delle parole. A tal fine risulta efficace usare un formalismo basato sugli *automi a stati finiti*. Più specificamente si costruisce un automa per il quale siano univocamente determinati uno stato di partenza (*stato iniziale*), uno *stato finale* che indica il completamento del riconoscimento di un token, e le cui transizioni (archi) siano etichettate con i caratteri che possono occorrere nella sequenza di ingresso.

Nel caso in esame, l'automa corrispondente alle regole lessicali descritte informalmente nella formulazione del problema è riportato in figura 3.1. Nella figura, l'etichetta `any` riportata su un arco uscente da uno stato S rappresenta l'insieme di tutti i possibili ingressi (i caratteri nel nostro caso), esclusi quelli che etichettano gli altri archi uscenti da S. Ad esempio, l'etichetta sul cappio dello stato 1 rappresenta tutti i caratteri che non siano lettere. Inoltre, nel rappresentare l'automa si è scelto di caratterizzare univocamente lo stato iniziale (finale) mediante un arco non etichettato entrante (uscente).

L'automa di figura 3.1 specifica che, per riconoscere un token (cioè una parola), la scansione della sequenza di ingresso deve procedere nel modo seguente. Inizialmente si devono saltare tutti i caratteri che precedono la prima

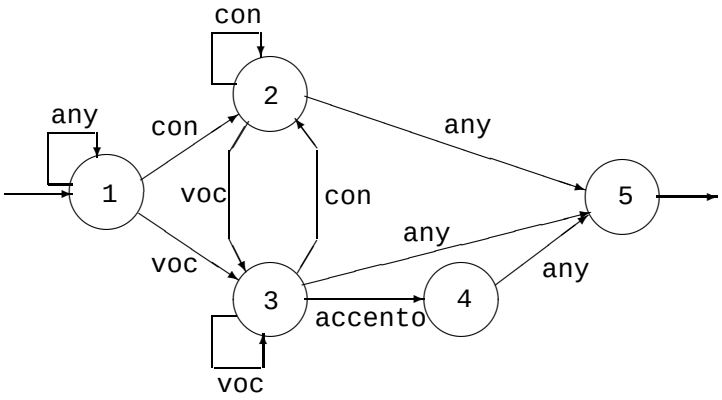


Figura 3.1: Automa a stati finiti che descrive le regole per il riconoscimento di parole in un testo.

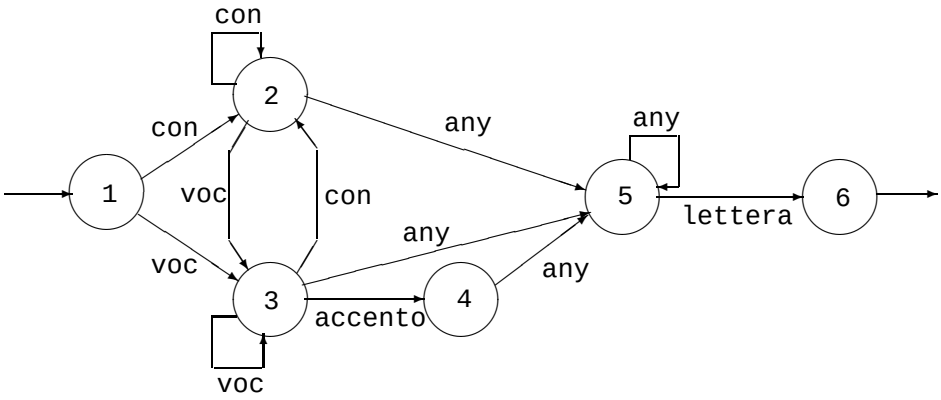


Figura 3.2: Automa che salta i separatori dopo il riconoscimento di una parola.

lettera presente nella sequenza, quindi si acquisiscono le lettere che formano la parola, mantenendo memoria se l'ultimo carattere letto era una vocale o una consonante. Nel primo caso se si incontra un carattere che non sia una lettera e che non sia un accento si termina, mentre se si incontra un accento esso viene considerato parte della parola e quindi si termina. Nel secondo caso, non appena si incontra un carattere che non sia una lettera si termina.

Alternativamente, si potrebbe trasporre le regole lessicali che governano il riconoscimento di una parola nell'automa di figura 3.2. Tale automa descrive evidentemente le stesse regole di quello riportato in figura 3.1, ma prevede di saltare i caratteri che separano due parole consecutive dopo il riconoscimento. È chiaro che l'automa di figura 3.2 funziona correttamente solo se si assume che nello stato iniziale il carattere che segue nella sequenza di ingresso sia una lettera. Poiché però alla fine del riconoscimento di ciascuna parola ci si posiziona all'inizio della parola successiva prima di raggiungere lo stato finale, tale vincolo risulta sempre soddisfatto se si assume che gli eventuali caratteri iniziali che non siano lettere vengano saltati mediante una opportuna fase di inizializzazione.

La prima alternativa è in genere quella preferita, dal momento che risulta la più naturale e non impone precondizioni sulla sequenza di ingresso. Tuttavia vi possono essere motivi per scegliere la seconda soluzione.

In particolare, l'impiego dell'automa di figura 3.2 semplifica la verifica di terminazione del testo prima di estrarre il token successivo. Infatti, se l'estrazione del token precedente ha consumato tutti gli eventuali caratteri separatori, per verificare se il testo è terminato è sufficiente controllare il valore dell'attributo eof. Per questa ragione, seguiamo la seconda soluzione per realizzare l'analizzatore lessicale.

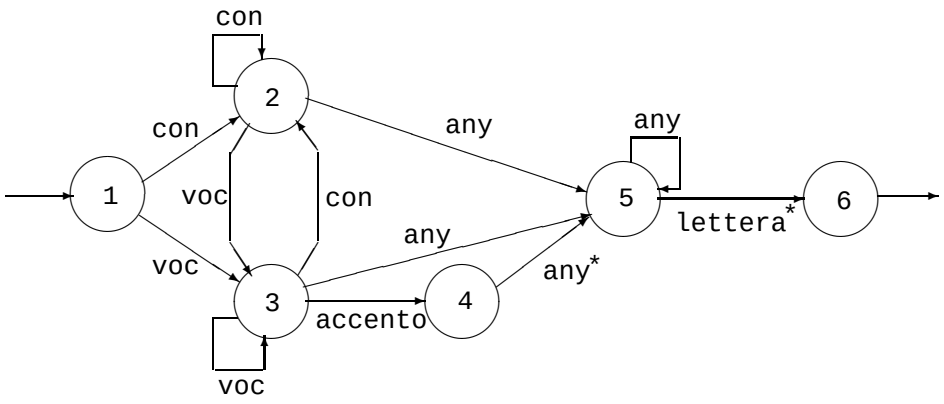


Figura 3.3: Automa che non consuma la prima lettera della parola successiva.

3.2 Avanzamento sulla sequenza di ingresso

In principio, ogni transizione sull'automa "consuma" il carattere che l'ha provocata, facendo avanzare la sequenza di ingresso. Tuttavia, per diverse ragioni[1], è possibile che in corrispondenza di alcune transizioni il carattere in ingresso non vada consumato per garantire il corretto funzionamento dell'analizzatore lessicale.

Per poter facilitare il passaggio dall'automa ad un programma eseguibile, introduciamo la possibilità di marcare con un asterisco le transizioni a cui non deve corrispondere l'avanzamento della sequenza di ingresso. In altre parole, quando viene seguita una transizione asteriscata il carattere che l'ha prodotta non viene consumato e quindi esso determina anche la transizione successiva.

Volendo applicare la convenzione appena introdotta all'automa di figura 3.2, si ottiene l'automa di figura 3.3, dove sono state marcate le transizioni che possono essere prodotte dalla lettera che inizia la parola successiva a quella appena riconosciuta. In questo caso infatti la lettera non va consumata, in quanto essa appartiene al token successivo e dovrà pertanto essere consumata all'inizio della successiva attivazione dell'analizzatore lessicale.

Si noti in particolare che, indipendentemente dal carattere in ingresso, nello stato 4 viene sempre seguita la transizione asteriscata, dal momento che il carattere precedente (accento grave o acuto) ha certamente terminato il token corrente. Il fatto che tale transizione sia stata asteriscata consente di riconoscere il token `sua` nella sequenza:

```
. . .  bonta‘sua  . . .
```

perché la `s` successiva all'accento non viene consumata durante il riconoscimento del token `bonta‘`. Viceversa se la transizione uscente dallo stato 4 non fosse stata asteriscata la `s` verrebbe consumata e un'attivazione successiva dell'automa restituirebbe erroneamente la stringa `ua`.

Viceversa, il cappio sullo stato 5 non è asteriscato in quanto esso serve proprio a consumare tutti i caratteri diversi da una lettera che possono eventualmente separare due token consecutivi. Nello stato 5 la prima lettera che viene letta dalla sequenza di ingresso produce la transizione nello stato finale dell'automa. Tale lettera non viene consumata in quanto essa corrisponde al primo carattere del token successivo.

3.3 Derivazione dell'analizzatore dall'automa

La derivazione dell'analizzatore lessicale dall'automa di figura 3.3 viene effettuata in modo sistematico secondo i principi illustrati di seguito[2]:

[1] Questo comportamento può essere richiesto sia perché normalmente i token vanno elaborati uno alla volta, sospendendo la scansione della sequenza di ingresso tra il riconoscimento di un token e quello del token successivo, sia a motivo delle azioni elaborative che possono accompagnare il riconoscimento, sia infine per particolari esigenze algoritmiche legate all'implementazione dell'analizzatore.

[2] Con analogo procedimento è facile derivare l'analizzatore corrispondente all'automa di figura 3.1. Lasciamo tale derivazine come esercizio.

- L'analizzatore lessicale consiste in generale in una procedura (SCANNER) che restituisce il token successivo presente in ingresso opportunamente rappresentato. Nel caso in esame, il token è sempre una parola e può quindi essere adeguatamente rappresentato mediante un array di caratteri.

- La procedura scanner ha poi una variabile locale $stato$ che tiene memoria dello stato corrente in cui si trova l'automa durante il riconoscimento[3]. La variabile $stato$ dovrà sempre essere inizializzata con il valore che rappresenta lo stato iniziale dell'automa.

- Il corpo della procedura SCANNER è costituito da un ciclo **while** che termina quando la variabile $stato$ assume il valore che rappresenta lo stato finale dell'automa. Il ciclo contiene un costrutto selettivo a più vie (**case**) i cui rami corrispondono ai diversi stati dell'automa. In corrispondenza a ciascun caso del costrutto selettivo, in base al successivo carattere presente nella sequenza di ingresso e alla struttura dell'automa, si determina lo stato successivo.

Sulla base di tali principi, per l'analizzatore corrispondente all'automa di figura 3.3 si deriva il codice riportato in figura 3.4. Si noti la corrispondenza tra i costrutti **if-then-else** che caratterizzano ciascun ramo del **case** e gli archi uscenti dal corrispondente stato dell'automa di figura 3.3. Nel codice, gli identificatori LETTERE, CONSONANTI, VOCALI, e ACCENTI, rappresentano rispettivamente l'insieme delle lettere dell'alfabeto inglese(maiuscole e minuscole), l'insieme delle sole consonanti, l'insieme delle sole vocali, e l'insieme costituito dai due accenti (grave e acuto).

Il codice di figura 3.4 non è però completo. Esso si limita a simulare il comportamento dell'automa in corrispondenza alla sequenza di caratteri data in ingresso, senza produrre alcun risultato utile. In particolare, non viene restituita la parola restituita. Tuttavia, sulla base del codice di figura 3.4, non è difficile aggiungere le istruzioni necessarie a tale scopo. Infatti, il fatto che l'algoritmo sia strutturato secondo quanto specificato dall'automa, consente di individuare immediatamente i punti del codice dove bisogna provvedere ad accumulare in un array $parola$ i caratteri letti. Tali punti sono quelli che corrispondono alle transizioni che riconoscono i caratteri che fanno parte della parola. Si ottiene pertanto il codice di figura 3.5.

Prima di chiudere il paragrafo, vogliamo sottolineare l'importanza di codificare il ciclo che implementa l'automa in modo da mantenere sempre perfetta corrispondenza con gli stati e i casi previsti dall'automa stesso, anche se questo dovesse comportare duplicazioni di codice apparentemente inutili. Il vantaggio di avere un codice ben strutturato risulta evidente quando bisogna aggiungere le azioni elaborative sulle sequenza riconosciute, ovvero quando occorre gestire situazioni di errore o comunque eccezionali (cfr. anche i paragrafi successivi).

3.4 Gestione della condizione di fine-file

Fino ad ora abbiamo supposto che una volta iniziata la scansione di una parola essa termini prima della fine della sequenza di ingresso. Questo significa che ogni parola è seguita da almeno un separatore. D'altra parte, poiché dopo il riconoscimento di una parola vengono consumati tutti i caratteri separatori che precedono la parola successiva, abbiamo anche supposto che dopo tali caratteri vi sia sempre una lettera e quindi un'altra parola. È evidente che tali ipotesi non sono realistiche in quanto corrispondono ad assumere che la sequenza di ingresso sia infinita.

In generale, qualunque analizzatore lessicale deve tenere conto della lunghezza finita della sequenza di ingresso e prevedere una gestione della condizione di fine-file. Una soluzione al problema consiste nell'assumere implicitamente che, oltre alle transizioni esplicitamente rappresentate nell'automa, da ogni stato sia possibile transire nello stato finale nel caso la sequenza di ingresso termini.

In base a tale convenzione, l'automa di figura 3.3 risulta essere una rappresentazione sintetica dell'automa riportato in figura 3.6, dove sono state rappresentate esplicitamente le transizioni causate dal verificarsi della condizione di fine-file. Il codice corrispondente è immediatamente derivabile da quello riportato in figura 3.4. L'unica differenza è la presenza della condizione $\neg eof[T]$ come condizione di uscita aggiuntiva del ciclo **while** (cioè in "and" con la condizione sullo stato). In tal modo quando si verifica la condizione di fine-file, il ciclo termina immediatamente simulando la transizione sullo stato di uscita dell'automa. Naturalmente, la procedura che effettua l'analisi lessicale deve restituire un infomazione logica che indica se è stato effettivamente trovato un token successivo, o la sequenza di token è terminata (vedi anche esempio successivo).

[3]Tale variabile può essere di un tipo a enumerazione se si sceglie di assegnare un nome simbolico a ciascuno stato, ovvero di tipo $integer$ se si sceglie di identificare gli stati con un numero. In questo capitolo, data la semplicità degli automi considerati si seguirà sempre la seconda soluzione.

```
SCANNER (T)
    { riconosce la parola successiva in T }
    stato ← 1
    while stato ≠ 6
        do  case stato of

            1 :  if next[T] ∈ CONSONANTI
                     then stato ← 2
                          GET(T)
                     else if next[T] ∈ VOCALI
                              then stato ← 3
                                   GET(T)

            2 :  if next[T] ∈ CONSONANTI
                     then stato ← 2
                          GET(T)
                     else if next[T] ∈ VOCALI
                              then stato ← 3
                                   GET(T)
                              else { next[T] = any }
                                   stato ← 5
                                   GET(T)

            3 :  if next[T] ∈ CONSONANTI
                     then stato ← 2
                          GET(T)
                     else if next[T] ∈ VOCALI
                              then stato ← 3
                                   GET(T)
                              else if next[T] ∈ ACCENTI
                                       then stato ← 4
                                            GET(T)
                                       else { next[T] = any }
                                            stato ← 5
                                            GET(T)

            4 :  { transizione asteriscata }
                 stato ← 5

            5 :  if next[T] ∈ LETTERE
                     then { transizione asteriscata }
                          stato ← 6
                     else { next[T] = any }
                          stato ← 5
                          GET(T)
```

Figura 3.4: Codice che simula l'esecuzione dell'automa di figura 3.3.

```
SCANNER-PAROLA (T, parola)
    { riconosce e restituisce la parola successiva in T }
    stato ← 1
    inizializza l'array parola come stringa vuota
    while stato ≠ 6
        do  case stato of
                1 :  if next[T] ∈ CONSONANTI
                         then stato ← 2
                              aggiungi next[T] all'array parola
                              GET(T)
                         else if next[T] ∈ VOCALI
                                  then stato ← 3
                                       aggiungi next[T] all'array parola
                                       GET(T)
                2 :  if next[T] ∈ CONSONANTI
                         then stato ← 2
                              aggiungi next[T] all'array parola
                              GET(T)
                         else if next[T] ∈ VOCALI
                                  then stato ← 3
                                       aggiungi next[T] all'array parola
                                       GET(T)
                                  else { next[T] = any }
                                       stato ← 5
                                       GET(T)
                3 :  if next[T] ∈ CONSONANTI
                         then stato ← 2
                              aggiungi next[T] all'array parola
                              GET(T)
                         else if next[T] ∈ VOCALI
                                  then stato ← 3
                                       aggiungi next[T] all'array parola
                                       GET(T)
                                  else if next[T] ∈ ACCENTI
                                           then stato ← 4
                                                aggiungi next[T] all'array parola
                                                GET(T)
                                           else { next[T] = any }
                                                stato ← 5
                                                GET(T)
                4 :  { transizione asteriscata }
                     stato ← 5
                5 :  if next[T] ∈ LETTERE
                         then { transizione asteriscata }
                              stato ← 6
                         else { next[T] = any }
                              stato ← 5
                              GET(T)
```

Figura 3.5: Codice completo dell'analizzatore che riconosce e restituisce le parole in un testo.

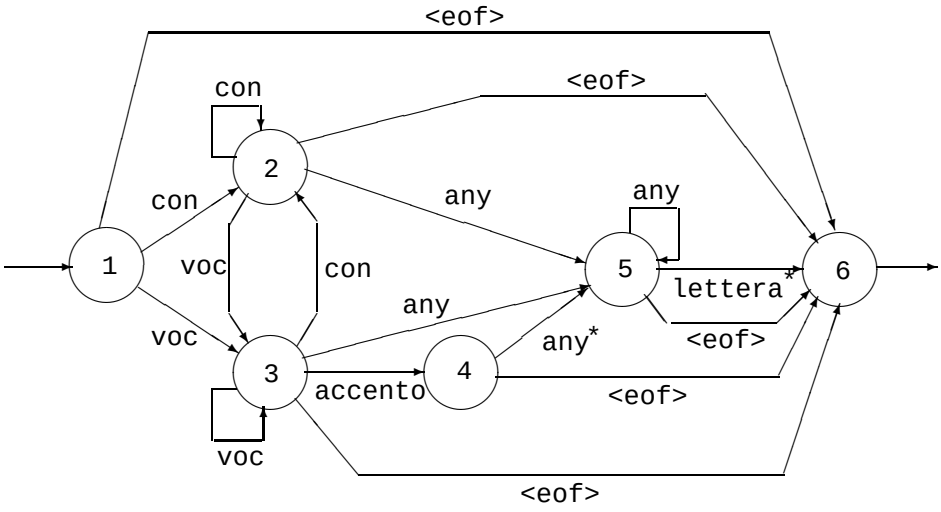


Figura 3.6: Automa con le transizioni corrispondenti alla condizione di fine-file.

3.5 Gestione degli errori lessicali

Durante la scansione della sequenza di ingresso si possono verificare situazioni impreviste che possiamo classificare come *errori lessicali*. Tali situazioni corrispondono al fatto che, trovandosi l'automa che guida il riconoscimento in uno stato s, nessuna delle transizioni uscenti da s risulta etichettata con il successivo carattere in ingresso.

Supponiamo, ad esempio, di voler riconoscere nella sequenza di ingresso costanti numeriche rappresentate in forma decimale e separate da spazi, caratteri di tabulazione (tab) e caratteri di fine linea (*newline/*). Supponiamo inoltre che siano ammissibili le seguenti quattro tipologie di rappresentazione decimale (`xxx` indica una qualsiasi sequenza non vuota di cifre):

```
xxx        xxx.        xxx.xxx        .xxx
```

L'automa in grado di riconoscere tutte e sole le sequenze ammissibili è riportato in figura 3.7. Come è facile verificare, tale automa non prevede il caso di una sequenza costituita da un separatore (spazio, tab o eol), seguito da un punto, seguito ancora da un separatore. Infatti dopo il primo separatore, il punto farebbe transire l'automa nello stato 4, e in tale stato non è previsto altro che una cifra come carattere successivo. In effetti, un punto isolato non rappresenta normalmente alcun valore numerico ed è quindi ragionevole ritenere che una tale evenienza corrisponda ad un errore lessicale nella sequenza di ingresso.

Analogamente al caso della gestione del fine-file, è opportuno rappresentare implicitamente gli stati e le transizioni necessari per la gestione degli errori. In particolare ad ogni automa che rappresenti regole lessicali viene implicitamente aggiunto un ulteriore stato di uscita (*stato di errore*), che rappresenta la terminazione in errore del processo di riconoscimento. Inoltre, per ogni stato s per il quale le transizioni esplicitamente rappresentate non coprono tutti i possibili ingressi, viene implicitamente aggiunta ulteriore transizione sullo stato di errore etichettata con `any`.

In base a tali convenzioni, l'automa di figura 3.7 equivale a quello di figura 3.8. Dal punto di vista algoritmico, l'introduzione dello stato di errore (stato 7 nella figura 3.8) richiede l'aggiunta di un test dopo la conclusione del ciclo **while** per discriminare i casi di terminazione in errore del riconoscimento. Ad esempio, se si decide di arrestare il programma nel caso si verifichi un errore lessicale, il codice da aggiungere dopo il ciclo **while** potrebbe essere il seguente[4]:

if $stato = 7$
    then HALT

[4]La procedura HALT produce la terminazione immediata del programma.

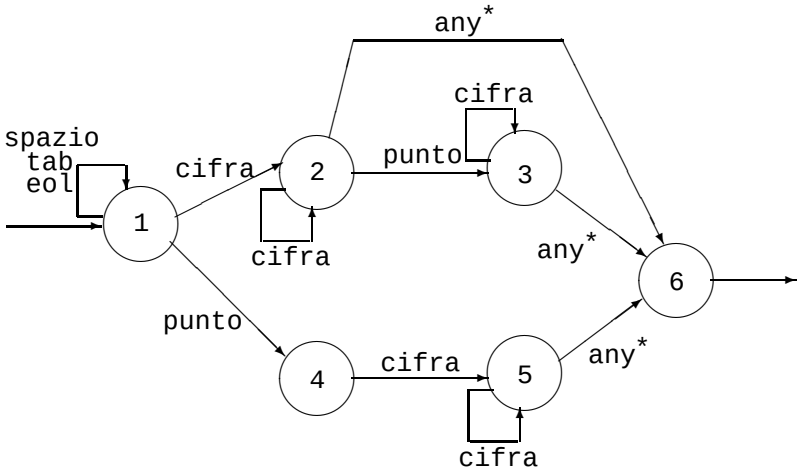


Figura 3.7: Automa che regola il riconoscimento di costanti reali.

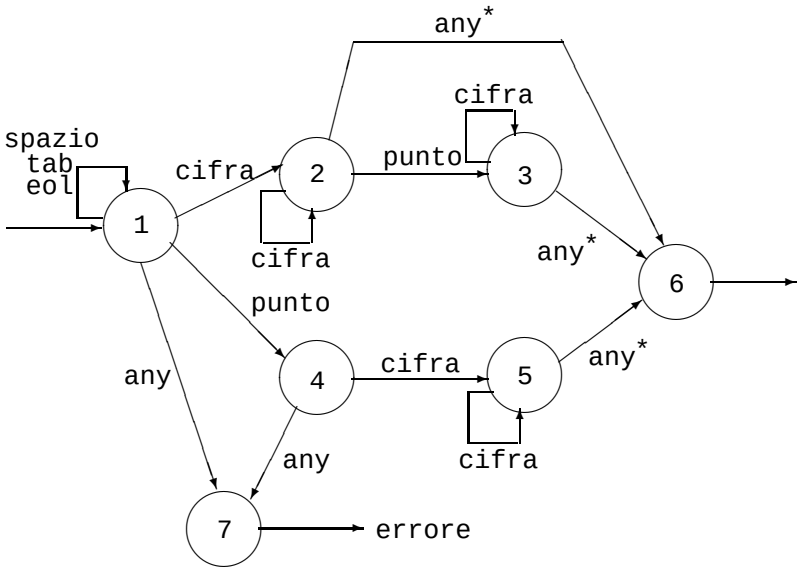


Figura 3.8: Automa con gli stati e le transizioni per la gestione degli errori.

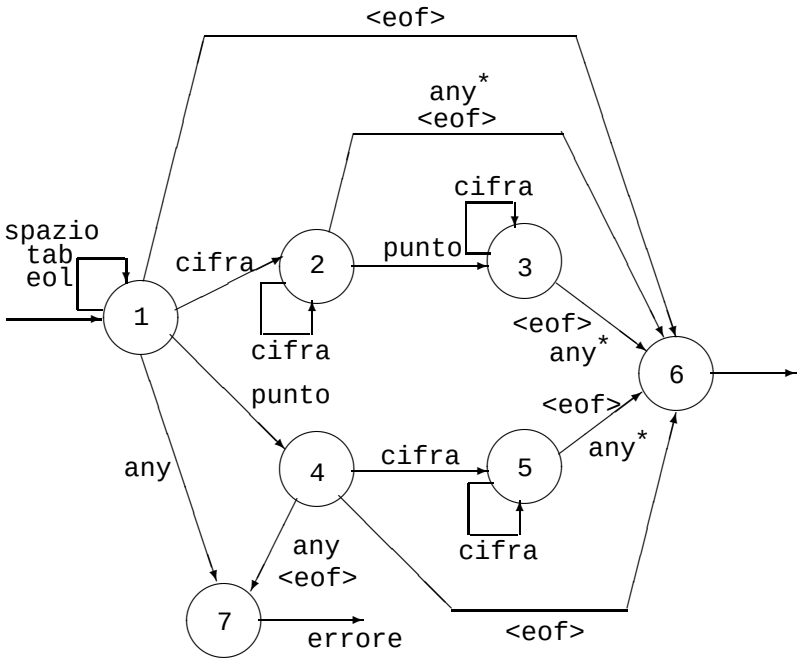


Figura 3.9: Automa completo equivalente alla rappresentazione sintetica di figura 3.7, senza le annotazioni sugli stati 1 e 4 (vedi testo).

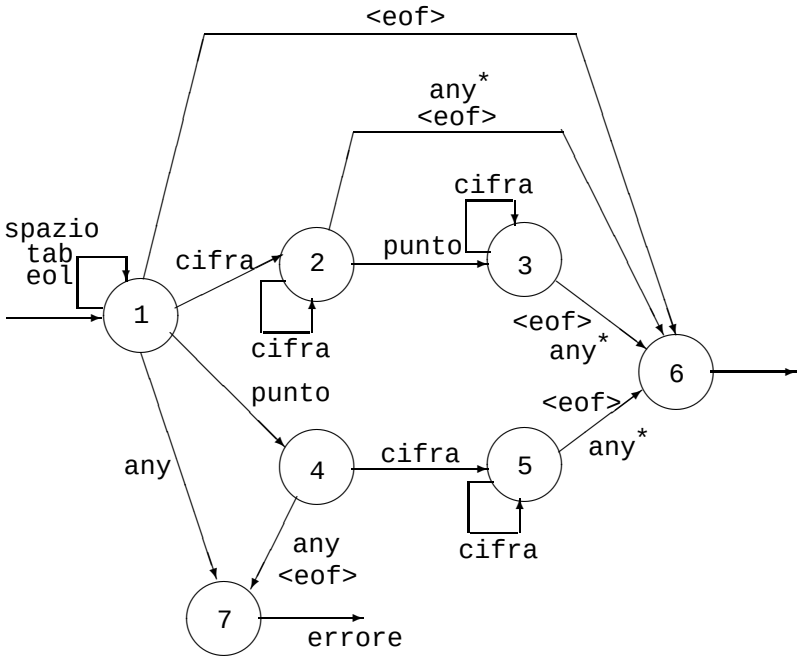


Figura 3.10: Automa completo equivalente alla rappresentazione sintetica di figura 3.7, con le annotazioni sugli stati 1 e 4 (vedi testo).

```
SCANNER-COSTANTE-NUMERICA (T)
    { se esiste, riconosce la costante numerica successiva in T }
    { e restituisce TRUE, altrimenti restituisce FALSE }
    stato ← 1
    while ¬eof[T] ∧ (stato ≠ 6) ∧ (stato ≠ 7)
        do  case stato of
                1 :  if next[T] ∈ CIFRE
                         then stato ← 2
                              GET(T)
                         else if next[T] = PUNTODECIMALE
                                  then stato ← 4
                                       GET(T)
                                  else if next[T] = SEPARATORI
                                           then stato ← 1
                                                GET(T)
                                           else { next[T] = any }
                                                stato ← 7 { errore }
                2 :  if next[T] ∈ CIFRE
                         then stato ← 2
                              GET(T)
                         else if next[T] ∈ PUNTODECIMALE
                                  then stato ← 3
                                       GET(T)
                                  else { next[T] = any }
                                       stato ← 6 { transizione asteriscata }
                3 :  if next[T] ∈ CIFRE
                         then stato ← 3
                              GET(T)
                         else { next[T] = any }
                              stato ← 6 { transizione asteriscata }
                4 :  if next[T] ∈ CIFRE
                         then stato ← 5
                              GET(T)
                         else { next[T] = any }
                              stato ← 7 { errore }
                5 :  if next[T] ∈ CIFRE
                         then stato ← 5
                              GET(T)
                         else { next[T] = any }
                              stato ← 6 { transizione asteriscata }

    if (stato = 4) ∨ (stato = 7)
        then HALT { errore lessicale }
        else if stato = 1
                 then return FALSE { non c'è un'altra costante }
    return TRUE
```

Figura 3.11: Codice dell'analizzatore che riconosce costanti reali.

3.6 Errori causati da condizioni di fine-file impreviste

Applicando anche la convenzione introdotta nel pragrafo 3.4, l'automa di figura 3.7 sarebbe in realtà equivalente all'automa di figura 3.9.

Il comportamento di tale automa non è però quello che ci si aspetterebbe. Infatti, mentre può essere accettabile che si verifichi la condizione di fine-file negli stati 2, 3, 5, tale condizione va certamente considerata un errore se si verifica nello stato 4 (cfr. paragrafo 3.5), mentre può essere accettabile o meno nello stato 1, a seconda delle modalità con cui il programma chiamante utilizza l'analizzatore.

Per questo motivo, quando vi sono stati nei quali il verificarsi della condizione di fine-file corrisponde ad una situazione di errore, occorre specificarlo esplicitamente. Poichè tale situazione non è in pratica frequente, per non appesantire la notazione grafica dell'automa, annoteremo a parte gli stati per i quali il verificarsi della condizione di fine-file non sia accettabile. Assumendo di annotare esplicitamente il caso di errore appena evidenziato dovuto all'improvvisa terminazione dell'input nello stato 4, l'automa di figura 3.7 è quindi equivalente all'automa finale di figura 3.10.

Dal punto di vista algoritmico, le considerazioni svolte in questo paragrafo si traducono in un ulteriore controllo sullo stato corrente all'uscita del ciclo **while**. Ad esempio, se nel caso dell'automa di figura 3.7 occorre specificare che la condizione di fine-file è un errore se si verifica quando l'automa è nello stato 4.

Inoltre, nel caso non si sia verificato alcun errore, l'analizzatore deve restituire un valore logico che indica se è stato acquisito un nuovo token o se la sequenza di ingresso è terminata.

Lo pseudo-codice completo dell'analizzatore risulta pertanto essere quello riportato in figura 3.11, dove gli identificatori CIFRE e SEPARATORI, rappresentano rispettivamente l'insieme delle cifre decimali e l'insieme costituito da spazio bianco, carattere di fine linea e tabulatore, e dove l'identificatore PUNTODECIMALE rappresenta il punto decimale.

Il codice riportato si limita a riconoscere la cosatante successiva senza restituirne il valore numerico. Nel paragrafo 3.7.2 viene presentata la procedura completa in C++ derivata dallo pseudo-codice di figura 3.11.

3.7 Codice C

In questo paragrafo riportiamo il codice C++ completo degli analizzatori corrispondenti agli automi delle figure 3.3 e 3.7. Per le operazioni di lettura è stata utilizzata la gestione degli stream del C++. Il codice verrà riportato sotto forma di file compilabile da collegare ad un programma principale che non viene mostrato per brevità. All'inizio di ciascuna porzione di codice è riportato un commento contente il nome del file e una breve descrizione delle funzionalità implementate.

Il codice verrà corredato da commenti volti a chiarire alcune differenze nella trasposizione dall'automa al codice dovute a specificità del C++. I commenti sono riportati seguendo l'ordine con cui gli aspetti commentati si trovano nel codice.

3.7.1 L'analizzatore di parole

```
/* scanner.cpp
   analizzatore di parole in un testo */

# include <fstream.h>
# include <ctype.h>

void scanner ( ifstream &T, char *parola );
/* riconosce la successiva parola presente nello
   stream T e la assegna a parola
   PRE: il prossimo carattere in ingresso e' una lettera */

int isVocale ( char ch );
/* restituisce 1 se ch e' una vocale, 0 altrimenti */
```

```
void scanner ( ifstream &T, char *parola )
{
  int stato = 1;
  char ch;   // variable temporanea usata solo per consumare i caratteri
  int n = 0; // inizializza la parola come stringa vuota
  while ( !T.eof() && (stato != 6) )
    switch ( stato )
      {
        case 1: if ( isVocale(T.peek()) )
                  { stato = 3; parola[n++] = T.peek(); T.get(ch); }
                else // T.peek() deve essere una consonante
                  { stato = 2; parola[n++] = T.peek(); T.get(ch); }
          break;

        case 2: if ( isVocale(T.peek()) )
                  { stato = 3; parola[n++] = T.peek(); T.get(ch); }
                else if ( isalpha(T.peek()) )
                  // T.peek() deve essere una consonante
                  { stato = 2; parola[n++] = T.peek(); T.get(ch); }
                else // T.peek() = any
                  { stato = 5; T.get(ch); }
          break;

        case 3: if ( isVocale(T.peek()) )
                  { stato = 3; parola[n++] = T.peek(); T.get(ch); }
                else if ( isalpha(T.peek()) )
                  // T.peek() deve essere una consonante
                  { stato = 2; parola[n++] = T.peek(); T.get(ch); }
                else if ( T.peek() == '\'' || T.peek() == ''' )
                  { stato = 4; parola[n++] = T.peek(); T.get(ch); }
                else /* T.peek() = any */
                  { stato = 5; T.get(ch); }
          break;

        case 4: stato = 5;   // transizione asteriscata
          break;

        case 5: if ( isalpha(T.peek()) ) // transizione finale asteriscata
                  stato = 6;
                else /* T.peek() = any */
                  { stato = 5; T.get(ch); }
          break;

      }
  parola[n] = '\0';   // termina la parola riconosciuta, lunga n caratteri
}
```

Nel file `scanner.c` è contenuta la procedura `scanner` il cui prototipo è dichiarato dopo l'inclusione di alcuni header file relativi alle funzioni di libreria utilizzate dal codice. Dopo il prototipo di `scanenr`, è dichiarato il prototipo di una funzione (`isVocale`) che si suppone pure contenuta nel file, ma non è mostrata per brevità.

Il parametro di scambio della procedura `scanner` è un puntatore a carattere. Si suppone che il chiamante passi il puntatore ad un array di caratteri sufficientemente lungo per poter contenere la successiva parola in ingresso. L'allocazione di tale array (che può essere statica o dinamica) è chiaramente a carico del chiamante.

In C++, il text file contenente il testo da analizzare è realizzato mediante un oggetto della classe `ifstream` passato per riferimento alla procedura `scanner`. L'avanzamento sul text file è realizzata mediante il metodo `get` e richiede l'introduzione di una variabile temporanea (`ch`) in cui viene memorizzato il carattere consumato e che non viene mai usata dall'algoritmo di analisi. Gli attributi $next$ e eof sono facilmente realizzati mediante i metodi `peek` e `eof`.

Si noti l'uso della funzione `isalpha` che restituisce 1 (TRUE) se il carattere passato è una lettera e restituisce 0 (FALSE) altrimenti. Si noti anche come nei vari stati si effettua prima il test sulle vocali, più facile da implementare, in modo da semplificare il test sulle consonanti.

Si noti l'uso dell'operatore `++` nella frase `p[n++] = T.peek()`. L'indice `n` indica in ogni momento il numero di caratteri accumulati nell'array puntato da `p`. Poichè gli indici degli array in C partono da 0, il valore di `n` indica sempre anche la successiva cella dell'array da riempire. Quindi il valore restituito da `T.peek()` deve essere dapprima memorizzato nella prima cella libera dell'array e quindi l'indice `n` deve essere incrementato in modo da tener conto del nuovo carattere aggiunto. Questo effetto viene ottenuto specificando l'operatore `++` dopo l'identificatore `n` (post-incremento).

Una volta usciti dal ciclo, la parola riconosciuta è stata copiata nell'array puntato da `p`. Tuttavia perchè tale parola possa essere usata come una normale stringa di caratteri occorre aggiungere il carattere di terminazione `'\0'`. Poichè, come abbiamo visto, l'indice `i` individua la successiva cella libera dell'array, il carattere di terminazione viene assegnato a `p[n]`. Si noti che in questo caso `n` non è stato incrmentato in quanto il carattere di terminazione non fa logicamente parte della stringa[5].

3.7.2 L'analizzatore di costanti reali

```
/* scan-num.c
   analizzatore di costanti reali in notazione decimale */

# include <fstream.h>
# include <ctype.h>
# include <stdlib.h>


bool scan_const_num ( double &v )
/* se esiste, riconosce la successiva costante numerica presente nello
   stream T e ne assegna il valore a v e restituisce true, altrimenti
   restituisce false */
{
  double f; // fattore di scala per la parte decimale
  int stato = 1;
  char ch;   // variable temporanea usata solo per consumare i caratteri
  while ( !T.eof() && (stato != 6) && (stato != 7) )
    switch ( stato )
      {
        case 1: if ( isdigit(T.peek()) ) {
                  stato = 2;
                  v = (double) (T.peek() - '0'); // inizializza v
                  T.get(ch);
                }
                else if ( T.peek() == '.' ) {
                  stato = 4;
                  v = 0;           // la parte intera e' zero
                  f = 1.0 / 10.0;  // fattore della prima cifra decimale
```

[5] Il lettore attento noterà che, poichè la variabile `n` non viene più usata nel seguito, tale ultima osservazione è in realtà inutile; tuttavia conviene osservare che rispettare il ruolo delle variabili ha benefiche ripercussioni sulla comprensibilità e sulla manutenibilità dei programmi. Si pensi per esempio al caso in cui si volesse trasformare la procedura `scanner` in una funzione che restituisce il numero di caratteri da cui è costituita la parola riconosciuta; è evidente che in tal caso si può tranquillamente restituire il valore di `n` senza ulteriori complicazioni.

```
          T.get(ch);
        }
        else if ( isspace(T.peek()) ) {
          stato = 1;
          T.get(ch);
        }
        else // T.peek() = any -- errore
          stato = 7;
  break;

case 2: if ( isdigit(T.peek()) ) {
          stato = 2;
          v = v * 10.0 + (double) (T.peek() - '0');
          T.get(ch);
        }
        else if ( T.peek() == '.' ) {
          stato = 3;
          f = 1.0 / 10.0;  // fattore della prima cifra decimale
          T.get(ch);
        }
        else { // T.peek() = any -- transizione asteriscata finale
          stato = 6;
        }
  break;

case 3: if ( isdigit(T.peek()) ) {
          stato = 3;
          v = v + f * (double) (T.peek() - '0');
          f = f / 10.0;  // fattore della cifra successiva
          T.get(ch);
        }
        else { // T.peek() = any -- transizione asteriscata finale
          stato = 6;
        }
  break;

case 4: if ( isdigit(T.peek()) ) {
          stato = 5;
          v = v + f * (double) (T.peek() - '0');
          f = f / 10.0;  // fattore della cifra successiva
          T.get(ch);
        }
        else // T.peek() = any -- errore
          stato = 7;
  break;

case 5: if ( isdigit(T.peek()) ) {
          stato = 5;
          v = v + f * (double) (T.peek() - '0');
          f = f / 10.0;  // fattore della cifra successiva
          T.get(ch);
        }
        else { // T.peek() = any -- transizione asteriscata finale
```

```
                stato = 6;
              }
          break;

      }
  if ( stato == 4 || stato == 7 )
      exit(1);  // errore lessicale
  else if ( stato = 1 )
      return false;
  return true;
}
```

Ci limitiamo a riportare commenti che non siano doppioni di quelli già fatti nel paragrafo precedente.

Innanzi tutto è bene osservare che, a differenza di quello riportato in figura 3.11, l'analizzatore sopra riportato è completo, nel senso che non solo riconosce le costanti, ma ne calcola il valore numerico e lo restituisce al chiamante. Questo è documentato nell'intestazione della procedura `scan_const_num` che specifica due parametri di tipo puntatore che corrispondono alle informazioni in uscita dalla procedura e nel commento che descrive la sua funzionalità.

Il calcolo del valore numerico della costante letta viene effettuato inserendo le opportune azioni elaborative in corrispondenza delle transizioni che consumano le cifre. Il calcolo è basato sulla cosiddetta regola di Horner per il calcolo di un polinomio:

$$P(x) = \sum_{k=0}^{n} = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n \cdots))$$

dove, nel caso specifico, $x = 10$, e i coefficienti $a_n, a_{n-1}, \ldots, a_0$ sono, nell'ordine, le cifre lette durante la scansione del testo in ingresso.

Quando nello stato 1 si incontra la prima cifra, si provvede a convertirla in numero sfruttando il fatto che in C `char` è un tipo numerico e su di esso si possono pertanto fare direttamente operazioni aritmetiche. Se invece si incontra un punto, si inizializza `v` a 0 essendo nulla la parte intera della costante.

Quando si incontrano le successive cifre della parte intera (stato 2) viene aggiornato il valore di `v` effettuatndo uno scorrimento a sinistra di una cifra (moltiplicazione per 10) e aggiunto il valore corrispondente all'ultima cifra letta.

Per quanto riguarda la parte decimale, essa viene calcolata utilizzando un fattore di scala. Tale fattore viene inizalizzato a $1/10$ in corrispondenza delle transizioni prodotte dal punto e viene via via diviso per 10 man mano che si incontrano cifre dopo il punto decimale. Si noti la necessità di scrivere la frazione $1/10$ nella forma `1/10.0` per evitare che l'operatore `/` indichi la divisione intera che produrrebbe 0 come risultato.

Infine, va osservato l'uso delle funzioni della libreria standard `isdigit` e `isspace`. La prima restituisce 1 se i carattere passato è una cifra, mentre la seconda restituisce 1 se il carattere passato è uno spazio, un tabulatore o un carattere di fine linea. Entrambe le funzioni restituiscono 0 negli altri casi.

Capitolo 4

Analisi sintattica (bozze, v. 1.0)

Come è noto, i linguaggi formali sono caratterizzati da un *lessico*, una *sintassi* e una *semantica*. Il lessico è l'insieme dei *simboli lessicali* o *token* che rappresentano gli elementi base con cui costruire le frasi del linguaggio. Un token è a sua volta costituito da uno o più caratteri, secondo opportune *regole lessicali*, e nel capitolo 3 abbiamo presentato un formalismo per descrivere le regole lessicali e un metodo per derivare da tali regole un *analizzatore lessicale*, cioè un programma in grado di trasformare una sequenza di caratteri in una sequenza di token.

In questo capitolo affronteremo il problema di descrivere la sintassi di un linguaggio, e cioè l'insieme di regole (dette *regole sintattiche*) che stabiliscono il modo con cui i token appartenenti al lessico possono essere composti per formare frasi valide, e di derivare da queste un analizzatore sintattico o parser, cioè un programma in grado di trasformare una sequenza di token in un albero che ne descrive la struttura sintattica. In tutti gli esempi si farà riferimento alla sintassi del linguaggio Pascal perché le regole sintattiche di tale linguaggio risultano particolarmente efficaci sotto l'aspetto didattico. La maggior parte degli esempi possono comunque essere pienamente compresi anche senza avere familiarità con tale linguaggio. Nei pochi casi in cui è necessario verranno fornite le nozioni necessarie a una piena comprensione da parte di chi abbia familiarità con un linguaggio basato sulla sintassi del C.

4.1 Definizione della sintassi

Informalmente siamo abituati a descrivere le modalità di costruzione di una particolare frase di un linguaggio specificando in quale successione devono essere disposti i token corrispondenti. Per esempio, dovendo spiegare la sintassi di una frase **if** in Pascal, diciamo che la frase può assumere una delle due forme seguenti:

if *espressione* **then** *frase*

if *espressione* **then** *frase* **else** *frase*

volendo con questo significare che occorre iniziare con la parola chiave **if**, a cui segue un'espressione (di tipo booleano[1]), a cui segue la parola chiave **then**, a cui segue una qualunque frase valida Pascal, a cui può o meno seguire la parola chiave **else**, a sua volta seguita da un'altra frase Pascal.

4.1.1 Grammatiche libere da contesto

Questo modo informale di procedere, può essere formalizzato introducendo la nozione di *grammatica di un linguaggio formale* che descriva in modo univoco l'insieme delle regole sintattiche. Limiteremo la nostra attenzione alle *grammatiche libere da contasto (context-free)*, che sono le più usate per descrivere la sintassi dei linguaggi di programmazione e per derivare gli analizzatori sintattici necessari per l'analisi e la traduzione dei programmi.

Una grammatica contaxt-free G è una quadrupla:

$$G = (T, N, P, S)$$

[1] In realtà il fatto che l'espressione debba essre di tipo booleano viene solitamente classificato come regola semantica e non sintattica, e pertanto tale specificazione non dovrebbe essere fatta. È tuttavia indicativo che, proprio a causa del carattere informale del modo di procedere che stiamo descrivendo, si tenda spesso a mischiare i due aspetti.

dove:

- T è l'insieme dei token, detti anche *simboli terminali*;
- N è l'insieme dei *simboli non terminali*;
- P è l' insieme delle *regole di produzione*,
- S è un particolare simbolo non terminale detto *simbolo inizale*;

Le regole sintattiche sono propriamente descritte attraverso le regole di produzione della grammatica che hanno la forma:

$$nt \rightarrow s_1 s_2 \dots s_n$$

dove nt è uno dei simboli non terminali e ciascun s_i è un simbolo terminale o non terminale. Si noti che la sequenza di simboli a destra del segno $\rightarrow$ può anche essere vuota (cioè $n = 0$). Per meggiore leggibilità delle regole di produzione, tale sequenza vuota verrà nel seguito indicata con il simbolo speciale ϵ.

Una convenzione che adotteremo per semplificare la specifica delle regole di produzione consiste nel raggruppare le regole che si riferiscono ad uno stesso simbolo non terminale. Più specificamente con la notazione:

$$\begin{aligned} nt \quad &\rightarrow \quad s_1^1 s_2^1 \dots s_{n_1}^1 \\ &\mid \quad s_1^2 s_2^2 \dots s_{n_2}^2 \\ &\dots \\ &\mid \quad s_1^h s_2^h \dots s_{n_h}^h \end{aligned}$$

si intende specificare il seguente insieme di regole:

$$\begin{aligned} nt \quad &\rightarrow \quad s_1^1 s_2^1 \dots s_{n_1}^1 \\ nt \quad &\rightarrow \quad s_1^2 s_2^2 \dots s_{n_2}^2 \\ &\dots \\ nt \quad &\rightarrow \quad s_1^h s_2^h \dots s_{n_h}^h \end{aligned}$$

Inoltre, adotteremo la convenzione di indicare in corsivo gli identificatori che corrispondono a simboli non terminali e di indicare in grassetto quelli che corrispondono a simboli terminali. I caratteri speciali (isolatamente o in brevi sequenze) indicano sempre simboli terminali. In tal modo non è normalmente necessario specificare gli insiemi T e N separatamente dalle regole di produzione in quanto essi possono essere direttamente ricavati esaminando queste ultime.

Infine, nell'elencare le regole di produzione di una grammatica porremo sempre in prima posizione la regola (o le regole) che si riferiscono al simbolo iniziale. Con tale convenzione la grammatica risulta univocamente determinata dall'elenco delle regole.

Una volta definita, una grammatica context-free permette di derivare sequenze di token assumendo inizialmente come sequenza corrente il solo simbolo iniziale, e sostituendo poi ciascun simbolo non terminale presente nella sequenza corrente con il lato destro di una regola per quel simbolo. Il procedimento viene ripetuto ricorsivamente fino a quando la sequenza risulta formata solo da simboli terminali.

Siamo quindi in grado di dare la seguente definizione:

> *Il linguaggio definito da una grammatica context-free corrisponde all'insieme delle sequenze di token che è possibile derivare dalle regole della grammatica, seguendo il procedimento appena descritto.*

Concludiamo il paragrafo con un esempio. Consideriamo la grammatica:

$$\begin{aligned} lista \quad &\rightarrow \quad lista + cifra \mid lista - cifra \mid cifra \\ cifra \quad &\rightarrow \quad \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \end{aligned} \tag{4.1}$$

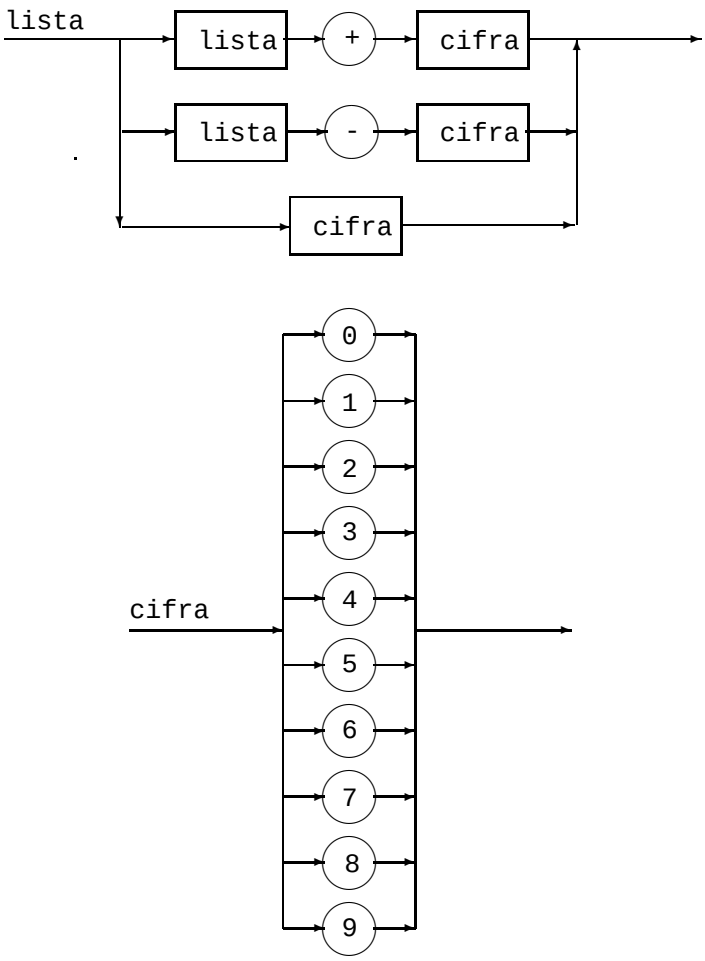


Figura 4.1: Diagrammi sintattici.

Come si può facilmente verificare appartengono al linguaggio definito dalla grammatica 4.1 le sequenti sequenze di token:

```
9 - 5 + 2
```

```
7
```

```
3 + 0 + 2 - 5 + 8 - 2
```

mentre non appartengono al linguaggio le sequenze:

```
19 - 598 + 20
```

```
+
```

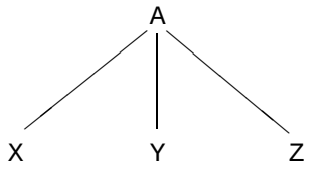
```
3 ++ 2 - 5 + - 8
```

In figura 4.1 è riportata una rappresentazione delle regole alternativa a quella introdotta nel presente paragrafo. Tale rappresentazione prende il nome di *diagrammi sintattici* ed è frequentemente usata per specificare la sintassi dei

linguaggi di programmazione per il fatto che essendo di natura grafica risulta facile da utilizzare. Si può facilmente constatare la perfetta corrispondenza tra le regole 4.1 e i diagrammi di figura 4.1.

4.1.2 Parse Tree

Un *parse tree* è un albero che descrive graficamente il modo con cui viene derivata attraverso le regole della grammatica una particolare sequenza di token appartenente al linuaggio da essa definito. Se per un simbolo non terminale A esiste una regola $A \rightarrow X\ Y\ Z$, allora in corrispondenza dell'applicazione di tale regola il parse tree avrà un nodo etichettato con A con tre figli, etichettati da sinistra a destra, rispettivamente, con X, Y e Z:



Più formalmente, data una grammatica context-free, un parse tree è un albero con le seguenti proprietà:

- la radice è etichettata con il simbolo iniziale;
- ogni foglia è etichettata con un token o con ϵ,
- ogni altro nodo è etichettato con un simbolo non terminale;
- i figli di ciascun nodo devono corrispondere a una regola di produzione.

Per esempio, in figura 4.2 è riportato il parse tree relativo alla sequenza `9-5+2` appartenente al linguaggio definito dalla grammatica di figura 4.1.

Il processo di derivare il parse tree per una sequenza di token data è detto *parsing* della sequenza.

4.1.3 Ambiguità di una grammatica

Dato un parse tree, risulta evidentemente determinata la corrispondente sequenza di token. In generale non è però vero il vicecersa. Ad esempio, se consideriamo la grammatica:

$$string \quad \rightarrow \quad string + string \mid string - string \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \tag{4.2}$$

esistono due diversi parse tree (cfr. figura 4.3) per la sequenza di token:

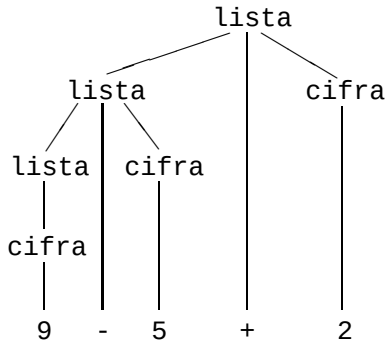


Figura 4.2: Parse tree relativo alla sequenza `9-5+2`, secondo la grammatica 4.1.

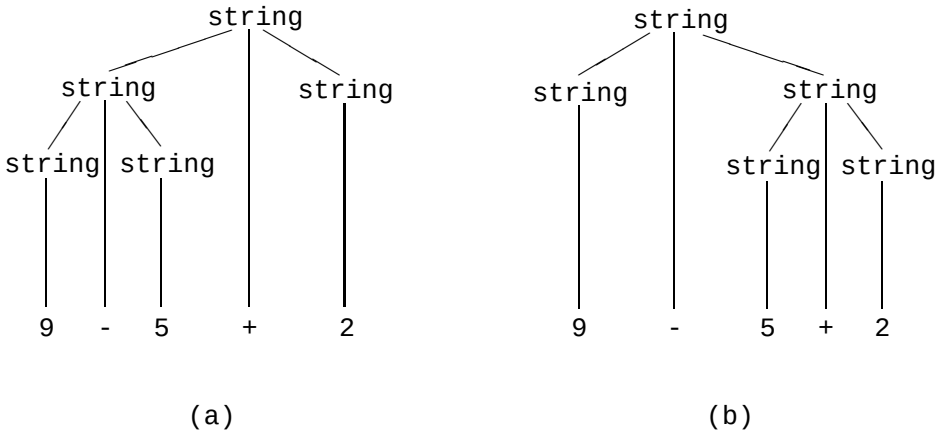


Figura 4.3: Due parse tree per la sequenza `9-5+2`.

```
9 - 5 + 2
```

Una grammatica come la 4.2 per la quale possono esistere parse tree differenti per una stessa sequenza di token sono dette *ambigue*. È facile invece verificare che tale eventualità non può verificarsi per la grammatica (4.1). Una tale grammatica è detta quindi *non ambigua*.

Poiché l'elaborazione di una sequenza di token (come ad esempio l'analisi e la traduzione di un programma) è generalmente legata alla struttura sintattica della sequenza, nella maggior parte dei casi pratici vengono esclusivamente utilizzate grammatiche non ambigue. Tutti gli esempi che considereremo nel prosieguo del capitolo godranno di questa proprietà.

4.1.4 Associatività degli operatori

Si consideri un operatore binario $\odot$ non necessariamente associativo. L'espressione:

$$a \odot b \odot c \tag{4.3}$$

risulta ambigua se non si specifica l'associatività dell'operatore. Non è chiaro infatti se debba essere effettuata per prima l'oparazione $a \odot b$ e il poi il suo risultato debba essere combinato con c, ovvero se debba essere effettuata prima l'operazione $b \odot c$ e poi a debba essre combinato col risultato ottenuto.

Sebbene questo tipo di ambiguità possa essere sempre risolta imponendo l'uso delle parentesi per specificare il significato dell'espressione, in genere si preferisce indicare esplicitamente il tipo di associatività dell'operatore, e cioè se esso è *associativo a sinistra* o *associativo a destra*. Nel primo caso, l'espressione 4.3 equivale all'espressione $(a \odot b) \odot c$ in quanto si applica prima l'operatore alla sinistra di b. Nel secondo caso, l'espressione 4.3 equivale all'espressione $a \odot (b \odot c)$ in quanto si applica prima l'operatore alla destra di b.

Per convenzione, i quattro operatori aritmetici $+$, $-$, $*$ e $/$ sono considerati associativi a sinistra[2]. Si noti tuttavia che esistono esempi di operatori associativi a destra. Un esempio è quello dell'elevamanto a potenza. Infatti l'espressione:

$$a^{b^c}$$

va interpretata come:

$$a^{(b^c)}$$

[2]Tali operatori, dal punto di vista matematico godono della proprietà associativa (da non confondere con la nozione di associatività di un operatore che è l'oggetto di questo paragrafo) e dunque per essi non avrebbe importanza specificare il tipo di associatività. Tuttavia, come è noto, gli operatori dell'aritmetica dei calcolatori non sono in realtà associativi e dunque è importante stabilire l'ordine di applicazione degli operatori nel caso di espressioni contenenti più di due operandi.

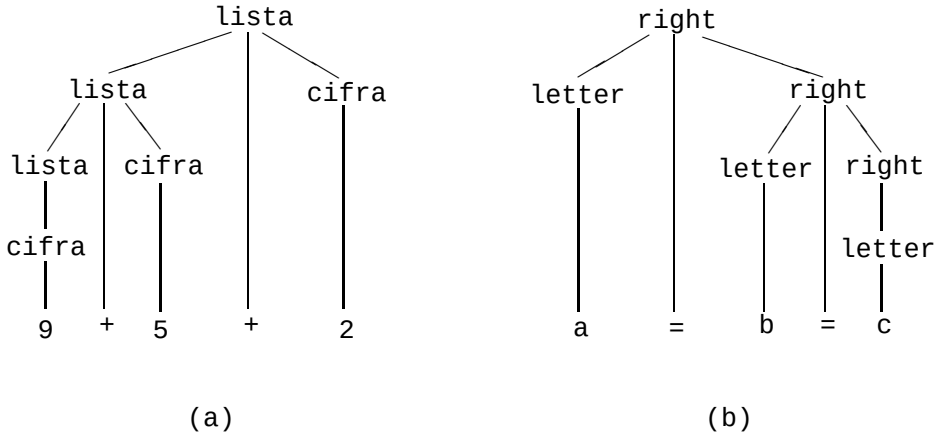


Figura 4.4: Parse tree per operatrori: (a) associativi a sinistra e (b) associativi a destra.

che è cosa ben diversa da:

$$\left(a^b\right)^c = a^{(b \cdot c)}$$

Un altro esempio di operatore che associa a destra è l'operatore di assegnazione nel linguaggio C. Infatti, la frase:

```
a = b = c
```

significa che il valore di `c` va assegnato a `b` e che poi il risultato dell'assegnazione (il valore assegnato a `b`) va assegnato a `a`. Si noti come in questo caso l'espressione:

```
(a = b) = c
```

non avrebbe senso in quanto l'espressione tra parentesi non è un *lvalue* valido (non rappresenta cioè una locazione di memoria).

L'associatività di un operatore si riflette nel modo di definire le regole sintattiche che descrivono la formazione di espressioni corrette. Nel caso di operatori associativi a sinistra, le regole sono del tipo di quelle della grammatica 4.1. Tali regole sono cioè *ricorsive a sinistra* (*left-recursive*). Al contrario, nel caso di operatori come l'assegnazione in C, le regole sintattiche assumono la forma[3]:

$$right \rightarrow letter = right \mid letter$$

$$letter \rightarrow \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \ldots \mid \texttt{z}$$

e cioè sono *ricorsive a destra* (*right-recursive*).

In figura 4.4, sono mostrati i parse tree relativi a due espressioni che contengono rispettivamente operatori associativi a sinistra e a destra.

4.1.5 Precedenza degli operatori

Nelle espressioni contenenti operatori diversi, la conoscenza del tipo di associatività di tali operatori non è in generale sufficiente per risolvere le potenziali ambiguità. Per esempio l'espressione:

```
9 + 5 * 2
```

potrebbe essere interpretata come:

[3]Riportiamo qui ovviamente una sintassi molto semplificata in cui le espressioni atomiche sono costituite esclusivamente da singole lettere dell'alfabeto inglese.

```
(9 + 5) * 2
```

oppure come:

```
9 + (5 * 2)
```

Il fatto che noi convenzionalmente attribuiamo all'espressione senza parentesi il secondo significato dipende dal fatto che si sono introdotti delle *precedenze* tra gli operatori e che in particolare l'operatore $*$ ha precedenza sull'operatore $+$.

L'esistenza di livelli di precedenza tra gli operatori si rispecchia sul modo di definire le regole sintattiche che governano la formazione di espressioni complesse.

Per illustrare come si deve procedere, consideriamo il caso di espressioni che contengano i quattro operatori aritmetici fondamentali. Le regole di precedenza e di asociatività di tali operatori sono riportate nella tabella seguente:

operatore	associativo a	precedenza
`+ -`	sinistra	0
`* /`	sinistra	1

Per definire una grammatica per le espressioni, si introduce un simbolo non terminale ($factor$) che rappresenta l'unità base delle espressioni. Nel nostro caso, tale unità base può essere una costante (che assumiamo per semplicità essere esclusivamente numeri interi da `0` a `9`) o una espressione (rappresentata dal simbolo non terminale $expr$) tra parentesi. Di conseguenza, la regola per il simbolo non terminale $factor$ è:

$$factor \quad \rightarrow \quad cifra \mid (\ expr\ ) \tag{4.4}$$

Per ogni livello di precedenza si introduce poi un ulteriore simbolo non terminale con le associate regole di produzione, che risultano ricorsive a sinistra o a destra a seconda del tipo di associatività degli operatori appartenenti a quel livello. Il simbolo non terminale $expr$ utilizzato nella regola per l'unità base delle espressioni viene associato al livello di precedenza più basso. Indicando con $term$ il simbolo non terminale associato agli operatori `*` e `/` si hanno pertanto le regole:

$$\begin{aligned} expr \quad &\rightarrow \quad expr + term \mid expr - term \mid term \\ term \quad &\rightarrow \quad term{*}factor \mid term/factor \mid factor \end{aligned} \tag{4.5}$$

È facile verificare, sulla base delle regole sintattiche appena introdotte che il parse tree corrispondente alla sequenza di token `9+5*2` riflette correttamente la diversa precedenza degli operatori $+$ e $*$.

Per chiarire ulteriormente come devono essere costruite le regole sintattiche per le espressioni, proviamo a vedere come si modificano le regole 4.4 e 4.5 se ai quattro operatori aritmetici aggiungiamo l'elevamento a potenza (rappresentato dal simbolo `'^'`). In questo caso, ricordando che tale operatore è associativo a destra si ottiene:

$$\begin{aligned} expr \quad &\rightarrow \quad expr + term \mid expr - term \mid term \\ term \quad &\rightarrow \quad term * exponent \mid term\ /\ exponent \mid exponent \\ exponent \quad &\rightarrow \quad factor\ \hat{}\ exponent \mid factor \\ factor \quad &\rightarrow \quad cifra \mid (\ expr\ ) \end{aligned}$$

4.1.6 Un esempio di sintassi per le frasi di un linguaggio

Con riferimento anche alle regole sintattiche per le espressioni riportate nel paragrafo precedente, diamo di seguito le regole di produzione relative ad un sottoinsieme delle frasi Pascal costituito dall'assegnazione, la frase **if**, il ciclo **while** e la frase composta.

La sintassi semplificata di queste frasi è descritta dalle [4]:

$$
\begin{array}{lcl}
statement & \rightarrow & \mathbf{id} := expr \\
 & | & \mathbf{if}\ expr\ \mathbf{then}\ statement \\
 & | & \mathbf{if}\ expr\ \mathbf{then}\ statement\ \mathbf{else}\ statement \\
 & | & \mathbf{while}\ expr\ \mathbf{do}\ statement \\
 & | & \mathbf{begin}\ opt\_statement\ \mathbf{end} \\
opt\_statement & \rightarrow & statement\_list \mid \epsilon \\
statement\_list & \rightarrow & statement\_list\ ;\ statement \\
 & | & statement
\end{array}
\tag{4.6}
$$

È interessante notare come è stato risolto il problema di specificare che in una frase composta tra il **begin** e l'**end** vi è una lista di frasi che può essere anche vuota. A questo scopo è stato introdotto il simbolo non terminale $statement\_list$ che definisce una lista non vuota attraverso regole del tutto simili a quelle usate nei paragrafi precedenti per definire le espressioni. È stato poi introdotto un altro simbolo non termiale $opt\_statement$ per consentire di scegliere tra una lista non vuota (rappresentata dal simbolo non terminale $statement\_list$) e una lista vuota (rappresentata attraverso la sequenza di token vuota).

4.2 Analisi sintattica (Parsing)

Un *analizzatore sintattico* o *parser* è un programma che analizza se la sequenza di token fornita in ingresso appartiene al linguaggio definito da una grammatica e, in caso affermativo, costruisce il corrispondente parse tree [5].

A partire dalla grammatica context-free, il parser può essere sempre derivato in modo sistematico ed esistono diversi strumenti software che effettuano la generazione del parser automaticamente a partire da una specifica della grammatica. In pratica tuttavia ci si limita a prendere in considerazione solo grammatiche che consentono di effettuare il parsing in modo efficiente. In pratica, algoritmi con complessità lineare rispetto alla lunghezza della sequenza di ingresso sono sempre sufficienti per tutti i linguaggi di interesse pratico.

I metodi con cui può essere effettuato il parsing possono essere classificati in due grandi categorie. I metodi *top-down* costruiscono il parse tree partendo dalla radice e per la loro semplicità sono adatti ad essere usati per la generazione manuale del parser. I metodi *bottom-up* costruiscono invece il parse tree a partire dalle foglie e possono essere usati per trattare una classe di grammatiche più ampia di quelle trattabili con i metodi top-down. Per questo motivo i metodi bottom-up sono i più usati dai generatori automatici di parser.

4.2.1 Analisi top-down

Poichè siamo qui interessati a presentare algoritmi per la codifica manuale di analizzatori sintattici, ci limiteremo ad esaminare in dettaglio il parsing top-down. Nelle sue linee generali il metodo può essere descritto come l'esecuzione ripetuta dei seguenti due passi:

1. relativamente al nodo n del parse tree, etichettato con il simbolo non terminale A, si sceglie una delle regole per A e si generano i figli di n etichettati con i simboli presenti nella parte destra della regola scelta;

2. si individua il successivo nodo da esaminare tra quelli etichettati con simboli non terminali.

Il procedimento inizia generando la radice del parse tree etichettata con il simbolo iniziale della grammatica, e termina quando tutti i nodi sono etichettati da simboli terminali.

[4] Nelle regole compare il token **id** che rappresenta un generico identificatore. Dal punto di vista sintattico tutti gli identificatori svolgono infatti lo stesso ruolo e ad essi corrisponde un unico simbolo terminale. Tuttavia, poiché da un punto di vista semantico occorre distinguere i diversi identificatori, al token **id** viene normalmente associato un attibuto che contiene l'informazione corrispondente alla stringa di caratteri che costituisce l'identificatore stesso (vedi anche la successiva nota 9). L'analizztore lessicale dovrà pertanto restituire tale attibuto tutte le volte che riconosce nella sequenza di caratteri in ingresso un token di tipo **id**.

[5] In realtà in molti casi non è necessario che il parser costruisca materialmente il parse tree. Tuttavia, il processo di parsing è sempre equivalente a una visita del parse tree, anche se non viene generata una struttura dati ad albero.

Il punto cruciale del metodo è la scelta della regola da applicare al passo 1. Si ricordi infatti che l'obiettivo del parsing è quello di riconoscere se la sequenza di token in ingresso appartiene alla grammatica. È allora evidente che tale scelta dipende da tale sequenza e che, nel caso esistano più regole per un determinato simbolo non terminale, si può dare il caso di dover ripetere l'analisi di una porzione della sequenza nel caso ci si accorga di aver scelto una regola errata[6]. Per alcune grammatiche, è possibile scegliere la regola da applicare in modo che non si debba mai tornare sulle scelte effettuate. In questi casi il parsing prende il nome di *predictive parsing* e può essere realizzato scandendo una sola volta la sequenza di ingresso da sinistra verso destra utilizzando esclusivamente il token successivo in ingresso (*lookahead token*) per decidere quale regola applicare nel caso in cui ci siano alternative.

Illustriamo come si svolge il predictive parsing con un esempio. Si consideri la seguente grammatica:

$$\begin{array}{lcl} type & \rightarrow & simple \\ & | & \uparrow \textbf{id} \\ & | & \textbf{array}\ [\ simple\ ]\ \textbf{of}\ type \\ & & \\ simple & \rightarrow & \textbf{integer} \\ & | & \textbf{char} \\ & | & \textbf{num dotdot num} \end{array} \tag{4.7}$$

dove **num** rappresenta il token corrispondente a una costante intera[7] e **dotdot** rappresenta il token "`..`" che separa i limiti inferiore e superiore della dimensione di un array.

Se in ingresso viene fornita la sequenza di caratteri:

```
array [1..100] of integer
```

dopo l'analisi lessicale viene fornita in ingresso al parser la sequenza di token:

array `[` **num dotdot num** `]` **of integer**

A questo punto, il parser, seguendo il procedimento illustrato in precedenza, costruisce passo-passo il parse tree corrispondente alla sequenza di token in ingresso. In figura 4.5 sono riportati i diversi passi relativi alla costruzione del parse tree. Ogni passo corrisponde all'applicazione di una regola e la corrispondente scelta viene di volta in volta effettuata in base al lookahead token. Più specificamente:

passo a: Viene generata la radice etichettata con il simbolo iniziale $type$.

passo b: Per espandere la radice viene scelta la terza regola per il simbolo $type$ in quanto è l'unica che inizia con il lookahead token, che nel nostro caso è il primo token della sequenza di ingresso (**array**); la scansione della sequenza di ingresso avanza, "consumando" i token **array** e `[` che corrispondono alle prime due foglie del parse tree.

passo c: Viene incontrato il successivo nodo da espandere, etichettato con il simbolo non terminale $simple$; per espandere tale nodo viene scelta la terza regola per tale simbolo in quanto è l'unica che inizia con il lookahead token (**num**); la scansione della sequenza di ingresso avanza, "consumando" i token **num**, **dotdot**, **num**, `]`, e **of** che corrispondono alle foglie del parse tree.

passo d: Viene incontrato il successivo nodo da espandere, etichettato con il simbolo non terminale $type$; per espandere tale nodo viene scelta la prima regola per tale simbolo in quanto è l'unica che può iniziare con il lookahead token (**integer**).

passo e: Viene incontrato il successivo nodo da espandere, etichettato con il simbolo non terminale $simple$; per espandere tale nodo viene scelta la prima regola per tale simbolo in quanto è l'unica che inizia con il lookahead token (**integer**). Non vi sono altri nodi etichettati con simboli non terminali da espandere e la sequenza in ingresso è terminata: il processo di parsing termina con successo.

[6] Questa eventualità che viene generalmente denominata *backtracking*, equivale a ripetere in tutto o in parte la visita del parse tree.

[7] Anche in questo caso, analogamente al caso degli identificatori (cfr. nota 4) si tratta di un token con un attributo che fornisce l'informazione relativa la valore numerico della costante.

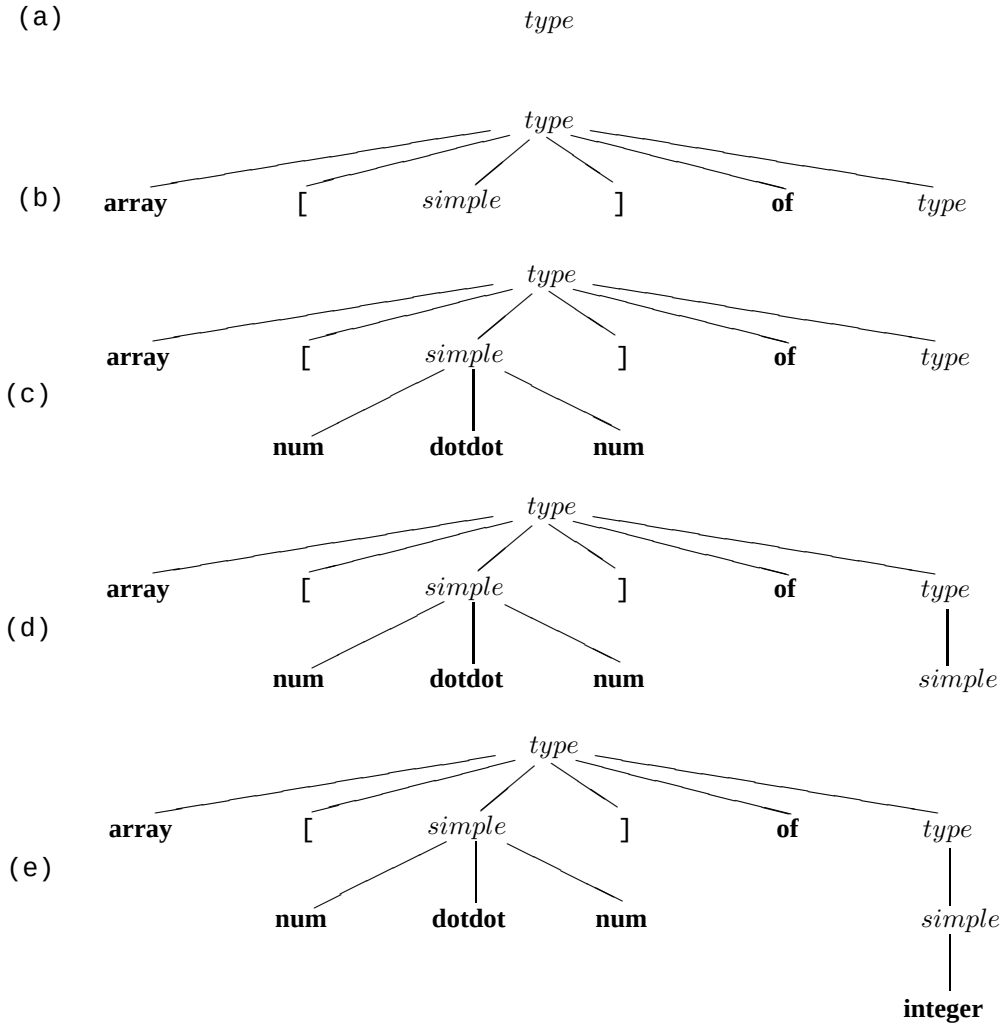


Figura 4.5: Passi della costruzione top-down di un parse tree.

Per dare una caratterizzazione precisa delle grammatiche per le quali è possibile applicare il predictive parsing, consideriamo una grammatica context-free, e sia α il lato destro di una regola per il simbolo non terminale A. Sia poi $FIRST(\alpha)$ l'insieme dei token che compaiono come primo simbolo delle sequenze che possono essere generate a partire da α. Se α è ϵ o può generare ϵ, allora ϵ fa parte di $FIRST(\alpha)$. Ad esempio, con riferimento alle regole 4.7:

$$\begin{aligned} FIRST(simple) &= \{\mathbf{integer}, \mathbf{char}, \mathbf{num}\} \\ FIRST(\uparrow \mathbf{id}) &= \{\uparrow\} \\ FIRST(\mathbf{array}\ [\ simple\ ]\ \mathbf{of}\ type) &= \{\mathbf{array}\} \end{aligned}$$

Sulla base di queste definizioni, possiamo affermare che si può realizzare un predictive parsing se, ogni qual volta vi sono due regole $A \rightarrow \alpha$ e $A \rightarrow \beta$ per il simbolo non terminale A, si ha che:

$$FIRST(\alpha) \cap FIRST(\beta) = \emptyset$$

In questo caso infatti il lookahead token consente sempre di scegliere in modo univoco la regola da applicare, e se nel seguito del procedimento si giunge ad un vicolo cieco (il lookahead token non trova corrispondenza con la succssiva foglia del parse tree) si può concludere che la sequenza in ingresso non appartiene al linguaggio senza bisogno di effettuare backtracking.

Prima di chiudere il paragrafo, è opportuno esaminare come ci si comporta in presenza di regole che hanno ϵ come lato destro (ϵ-produzioni). Il criterio di scelta da adottare è in questo caso quello di scegliere la ϵ-produzione come default quando il lookahead token non consente di scegliere alcun altra regola alternativa.

Ad esempio, dovendo applicare le regole 4.6 alla sequenza:

begin end

il primo token ci fa scegliere l'ultima regola per $statement$ e ci porta quindi a dover applicare la regola:

$$opt\_statement \quad \rightarrow \quad statement\_list \mid \epsilon$$

con **end** come lookahead token. Poichè **end** non appartiene a $FIRST(statement\_list)$, viene scelta la ϵ-produzione e l'analisi termina correttamente. Si noti che se al posto del token **end** vi fosse stato un altro token non appartenente a $FIRST(statement\_list)$, si sarebbe verificato un errore sintattico e la sequenza non sarebbe stata riconosciuta come appartenente al linguaggio.

4.2.2 Parsing ricorsivo discendente

Il *parsing ricorsivo discendente* (*recursive descent parsing*) è un metodo di tipo top-down che può essere facilmente codificato manualmente se la grammatica consente di utilizzare il predictive parsing. Un parser discendente ricorsivo è costituito da un insieme di procedure mutuamente ricorsive, con una diversa procedura per ciascun simbolo non terminale della grammatica.

Ogni procedura compie due azioni:

1. decide in base al lookahead token quale regola usare per il simbolo non terminale ad essa associato;
2. applica la regola scelta "mimando" il suo lato destro: per ogni simbolo non terminale chiama la corrispondente procedura e per ogni simbolo terminale verifica la corrispondenza con il lookahead token, facendo avanzare la sequenza di ingresso; se il lookahead token non corrisponde con il token previsto dalla regola viene rilevato un errore sintattico..

La sequenza dinamica di chiamate ricorsive che corrisponde alla scansione della sequenza di token in ingresso determina implicitamente il parse tree corrispondente. Una volta quindi che le procedure sono state derivare dalle regole della grammatica è possibile aggiungere in corrispondenza delle diverse chiamate il codice che effettua la costruzione del parse tree o qualunque altro tipo di analisi si voglia effettuare.

Prima di illustrare nel paragrafo successivo il metodo con un esempio, occorre risolvere un problema che si può presentare nel derivare le procedure del parser dalle regole grammaticali. Se la grammatica contiene una regola del tipo:

$$expr \quad \rightarrow \quad expr + term \mid expr - term \mid term \tag{4.8}$$

e se la procedura associata al simbolo non terminale $expr$, una volta attivata, sceglie la prima alterativa della regola, verrà eseguita una chiamata ricorsiva alla stessa procedura; poiché tale chiamata non produce avanzamento della sequenza di ingresso la nuova istanza della procedura selezionerà nuovamente la stessa alternativa, innescando una sequenza infinita di chiamate ricorsive!

Si riconosce facilmente che il problema è causato dal fatto che la regola è ricorsiva a sinistra. Infatti, negli altri casi, l'eventuale chiamata ricorsiva avviene dopo un avanzamento sulla sequenza di ingresso e quindi in condizioni diverse da quelle iniziali.

Queste osservazioni comportano che il parsing ricorsivo discendente non può essere impiegato su una grammatica che contenga regole ricorsive a sinistra. Fortunatamente però tali regole possono sempre essere opportunamente trasformate in modo da eliminare la ricorsione a sinistra senza modificare il linguaggio definito della grammatica. Nel caso più semplice, le regole vanno trasformate nel modo di seguito illustrato. Si supponga di avere la produzione ricorsiva a sinistra:

$$A \rightarrow A\,\alpha \mid \beta$$

Tale regola può essere sostituita dalla coppia di regole:

$$A \rightarrow \beta\,R$$

$$R \rightarrow \alpha\,R \mid \epsilon$$

senza modificare il linguaggio definito dalla grammatica. Infatti in entrambi i casi il linguaggio è costituito da sequenze che iniziano con β sequito da zero o più occorrenze di α. Si noti che la trasformazione ha comportato l'introduzione di un nuovo simbolo non terminale e il passaggio da una regola ricorsiva a sinistra a una regola ricorsiva a destra.

Per esemplificate il procedimento, consideriamo la grammatica che regola la formazione delle espressioni (cfr. paragrafo 4.1.4). Se osserviamo, con riferimento alla regola 4.8, che il simbolo $expr$ corrisponde ad A, le sequenze $+\ term$ corrispondono ad α nei due casi ricorsivi, e il simbolo $term$ corrisponde β, si ottiene la grammatica:

$$\begin{array}{lcl}
expr & \rightarrow & term\ \ rest_sum \\
 & & \\
rest_sum & \rightarrow & +\ term\ \ rest_sum \\
 & \mid & -\ term\ \ rest_sum \\
 & \mid & \epsilon \\
 & & \\
term & \rightarrow & factor\ \ rest_prod \\
 & & \\
rest_prod & \rightarrow & *\ factor\ \ rest_prod \\
 & \mid & /\ factor\ \ rest_prod \\
 & \mid & \epsilon \\
 & & \\
factor & \rightarrow & cifra \mid (\ expr\ )
\end{array}$$

Con tale nuova grammatica, il parse tree della sequenza `9-5+2` è quello riportato in figura 4.6

4.3 Un esempio di analisi basata su parsing discendente

Si voglia realizzare un sottoprogramma che analizzi espressioni aritmetiche contenenti gli operatori `+` e `-` in notazione in-fissa e le ristampi in notazione post-fissa. Si supponga per semplicità che gli operandi delle espressioni siano costanti intere da `0` a `9` e che non vi siano spazi all'interno delle espressioni[8].

Il problema richiede evidentemente di effettuare un analisi sintattica delle espressione e di compiere opportune azioni in corrispondenza del riconoscimento delle varie sotto-espressioni. Si tratta in effetti di operare una forma di traduzione dell'ingresso, del tutto analoga (sebbene di gran lunga più semplice) alla traduzione che deve effettuare

[8]L'ipotesi consente di risolvere il problema senza impiegare un analizzatore lessicale ed è stata introdotta solo per mantenere l'esempio entro dimensioni limitate. Non vi è nessuna difficoltà nel rilasciare tali vincoli purché si sostituisca la lettura diretta del file di ingresso con la chiamata ad un analizzatore lessicale. Il lettore può operare tale generalizzazione per esercizio.

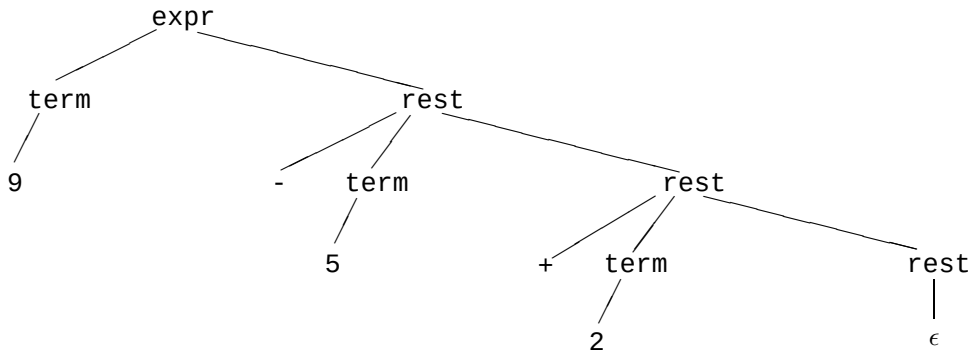


Figura 4.6: Parse tree della sequenza `9-5+2` secondo una grammatica non ricorsiva a sinistra.

il compilatore di un linguaggio di programmazione. In tali casi viene adoperata una tecnica denominata *traduzione diretta dalla sintassi* che consiste nello specificare le azioni elaborative che operano la traduzione (*azioni semantiche*) in corrispondenza delle regole sintattiche della grammatica context-free, e nell'immaginare che tali azioni vengano eseguite man mano che procede la creazione/visita del parse tree. L'insieme di regole così ottenuto prende il nome di *schema di traduzione*. L'ordine con cui le azioni semantiche di uno schema di traduzione devono essere eseguite corrisponde ad una visita in post-odine del parse tree.

Nel nostro caso, se indichiamo tra parentesi graffe le azioni semantiche, possiamo specificare la soluzione al nostro problema mediante il seguente schema di traduzione:

$$
\begin{aligned}
expr &\rightarrow expr{+}term\ \{print('+')\} \\
expr &\rightarrow expr{-}term\ \{print('-')\} \\
expr &\rightarrow term \\
term &\rightarrow \texttt{0}\ \{print('\texttt{0}')\} \\
term &\rightarrow \texttt{1}\ \{print('\texttt{1}')\} \\
&\ldots \\
term &\rightarrow \texttt{9}\ \{print('\texttt{9}')\}
\end{aligned}
$$

dove le azioni semantiche vanno interpretate come ulteriori foglie del parse tree che vengono eseguite quando vengono visitate (secondo una visita in post-ordine). Ad esempio, se in ingresso abbiamo la sequenza `9-5+2`, il parse tree integrato dalle azioni semantiche risulta essere quello di figura 4.7. È facile verificare che visitando in post-ordine tale albero viene stamapata la stringa di caratteri `95-2+` che è proprio l'espressione di partenza in notazione post-fissa.

Volendo tradurre questa soluzione in un programma che usi un parsing ricorsivo discendente, occorre innanzi tutto trasformare la sintassi con i metodi esposti nel paragrafo 4.2.2. Trattando le azioni semantiche come normali token si

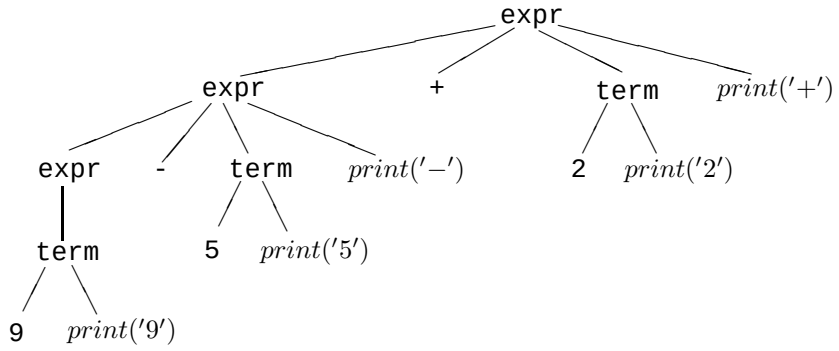


Figura 4.7: Parse tree contenente azioni semantiche.

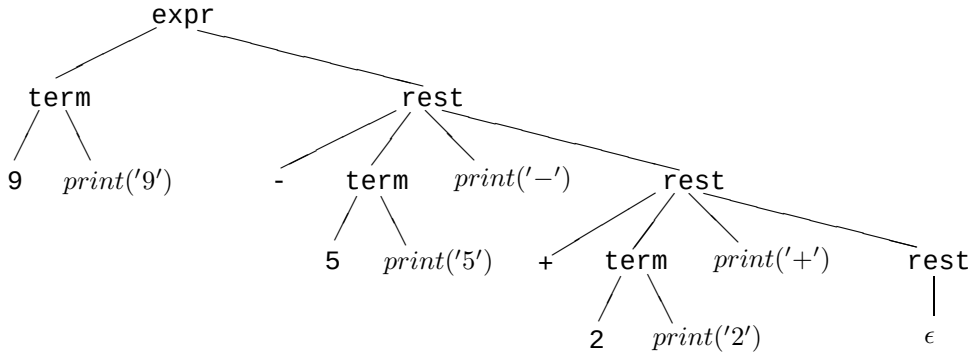


Figura 4.8: Parse tree contenente azioni semantiche e relativo a un grammatica non ricorsiva a sinistra.

ottiene lo schema di traduzione:

$$
\begin{array}{lcl}
expr & \rightarrow & term\ rest \\
 & & \\
rest & \rightarrow & +\ term\ \{print('+')\}\ rest \\
 & | & -\ term\ \{print('-')\}\ rest \\
 & | & \epsilon \\
 & & \\
term & \rightarrow & \texttt{0}\ \{print('\texttt{0}')\} \\
 & | & \texttt{1}\ \{print('\texttt{1}')\} \\
 & \ldots & \\
 & | & \texttt{9}\ \{print('\texttt{9}')\}
\end{array}
\tag{4.9}
$$

Si noti che il parsing della sequenza `9-5+2` genera questa volta il parse tree di figura 4.8, ma che visitando tale albero in post-ordine si ottiene ancora la stringa di usita `95-2+`.

A questo punto possiamo procedere a codificare le procedure da associare ai simboli non terminali $expr$, $term$ e $rest$. Per semplificare la codifica introduciamo la procedura $match$ che provvede a verificare che il lookahead token corrisponda al token previsto t, e che avanza sulla sequenza in ingresso in caso di successo e segnala un errore in caso di insuccesso:

```
TERM ( )
    if symbol[lookahead] ∈ CIFRE
        then stampa symbol[lookahead]
             MATCH(symbol[lookahead])
        else MATCH(symbol[lookahead]) { errore! }

REST ( )
    if symbol[lookahead] = PLUS
        then MATCH(PLUS)
             TERM()
             stampa PLUS
             REST()
        else if symbol[lookahead] = MINUS
                then MATCH(MINUS)
                     TERM()
                     stampa MINUS
                     REST()rest
                else { sequenza vuota di token }

EXPR ( )
    TERM()
    REST()
```

Figura 4.9: Procedure associate ai simboli non terminali della grammatica 4.9.

```
MATCH (s)
    { verifica che il prossimo token (attributo symbol dell'oggetto globale lookahead) coincida con s; }
    { avanza sulla sequenza di ingresso in caso affermativo, segnala errore altrimenti }
    if s = symbol[lookahead]
        then lookahead ← GETTOKEN()
        else HALT { errore sintattico }
```

La procedura match assume che l'analizzatore sintattico sia realizzato in modo da acquisire il token successivo in ingresso e di memorizzarlo nell'oggetto $lookahead$, gestito come variabile globale. Nel caso la sequenza di ingresso termini viene restituito un token speciale.

Abbiamo anche assunto che un oggetto di tipo token abbia sempre l'attributo $symbol$ che identifica il token. Nel caso di alcuni token specifici, l'oggetto restituito dalla'analizzatore lessicale deve avere anche altri attributi, generalmente non utilizzati dall'analizzatore sintattico, ma necessari alle fasi successive (analisi semantica, generazione di codice, ecc.). Ad esempio, nel caso di costanti numeriche, oltre all'informazione che il token acquisito è una costante, l'analizzatore lessicale solitamente restituisce anche il suo valore numerico. Analogamente nel caso di identificatori, è opportuno che l'analizzatore restituisca un'informazione che indichi quale identificatore è stato acquisito [9].

Il codice delle procedure che compongono l'analizzatore sintattico è riportato in figura 4.9. Nello pseudo-codice,

[9] La cosa più naturale sarebbe restituire la stringa di caratteri corrispondente all'identificatore, tuttavia, poiché il numero di identificatori diversi in un programma è generalmente molto limitato, solitamente, per semplificare l'implementazione e velocizzare gli accessi alla tabella dei simboli, si può procede nel seguente modo:

- l'analizzatore lessicale mantiene una tabella dinamica che associa agli identificatori trovati durante la scansione un identificativo unico (tipicamente un intero senza segno);
- quando l'analizzatore acquisisce come token un identificatore, restituisce un oggetto contenente l'informazione sul tipo di token acquisito e il valore intero ad esso associato;
- il valore intero associato all'identificatore viene usato per l'accesso alla tabella dei simboli e per qualsiasi altra elaborazione che richieda di distinguere un identificatore dall'altro.

```
REST ( )
    repeat
        if symbol[lookahead] = PLUS
            then MATCH(PLUS)
                 TERM()
                 stampa PLUS
            else if symbol[lookahead] = MINUS
                     then MATCH(MINUS)
                          TERM()
                          stampa MINUS
                     else { sequenza vuota di token }
    until (symbol[lookahead] ≠ PLUS) ∧ (symbol[lookahead] ≠ MINUS)
```

Figura 4.10: Eliminazione della tail-recursion dalla procedura $rest$.

```
EXPR ( )
    TERM()
    repeat
        if symbol[lookahead] = PLUS
            then MATCH(PLUS)
                 TERM()
                 stampa PLUS
            else if symbol[lookahead] = MINUS
                     then MATCH(MINUS)
                          TERM()
                          stampa MINUS
                     else { sequenza vuota di token }
    until (symbol[lookahead] ≠ PLUS) ∧ (symbol[lookahead] ≠ MINUS)
```

Figura 4.11: Procedura $expr$ ottimizzata che sostituisce le procedure $expr$ e $rest$ di figura 4.9.

l'identificatore CIFRE rappresenta l'insieme dei valori che corrispondono ai token `0`, `1`, ..., `9`, mentre gli identificatori PLUS e MINUS corrispondono rispettivamente ai token + e `-`.

Prima di concludere il paragrafo, con riferimento alle procedure di figura 4.9, vogliamo fare un'ultima osservazione che ci permette di ottimizzare in alcuni casi molto frequenti il codice del parser. La procedura ricorsiva $rest$ è caratterizzata dal fatto che in ogni caso la chiamata ricorsiva risulta essere l'ultima azione della procedura stessa. In altre parole al rientro da una chiamata ricorsiva la procedura termina sempre immediatamente senza compiere ulteriori azioni elaborative. Questa situazione viene detta *tail recursion* (letteralmente ricorsione "di coda") e consente una facile trasformazione della procedura da ricorsiva a iterativa. È chiaro infatti che la chiamata ricorsiva può essere sostituita da una ripetizione dell'intero corpo della procedura stessa (figura 4.10). Ricordando poi che la procedura $rest$ viene invocata sempre dalla procedura $expr$ (cfr. figura 4.9), tale ultima procedura può essere riscritta come in figura 4.11, eliminando completamente la procedura $rest$.

In questo modo la ricorsione viene sostituita da un ciclo con evidenti vantaggi sul piano dlla complessità spaziale (cfr. paragrafo 2.2). È chiaro poi che le trasformazioni considerate sono applicabili ogni volta che nella derivazione del parser si ottengono procedure come la $expr$ e la $rest$ di figura 4.9, tipiche del processo di eliminazione della ricorsione a sinistra.

Capitolo 5

Un Semplice Interprete (bozze, v. 1.0)

In questo capitolo viene presentata la struttura completa di un interprete di un semplice linguaggio che permette di calcolare espressioni aritmetiche. Lo scopo del capitolo è quello di applicare le nozioni sull'analisi lessicale e sintattica illustrate nei capitoli precedenti. Un ulteriore scopo è quello di chiarire mediante un esempio reale, sebbene necessariamente limitato, la natura degli interpreti.

5.1 Il linguaggio

Il linguaggio da interpretare è molto semplice. Esso consente di scrivere due tipi di frase:

- le *assegnazioni* che hanno la forma:

  $espressione$ `=>` $identificatore$`;`

- e le *valutazioni* che hanno forma:

  $espressione$`;`

Nelle assegnazioni e nelle valutazioni $espressione$ è una espressione aritmetica che può comprendere le operazioni di somma (`+`), sottrazione (`-`), prodotto (`*`) e divisione (`/`) tra costanti reali prive di segno e variabili espresse mediante un identificatore. È possibile anche usare parentesi tonde per modificare la precedenza delle operazioni.

La assegnazioni associano alla variabile identificata dall'$identificatore$ l'espressione a sinistra del simbolo di assegnazione (`=>`). Si noti che ciò che viene assegnato è *il testo dell'espressione* e non il suo valore. In altre parole l'assegnazione non produce alcuna valutazione dell'espressione assegnata, ma la variabile diviene un suo *alias*.

Le valutazioni producono una valutazione dell'espressione specificata e la visualizzazione del risultato. Durante la valutazione, ciascuna variabile viene sostituita dall'espressione di cui è alias.

Per quanto riguarda la dichiarazone delle variabili, il linguaggio non le prevede esplicitamente, ma l'occorrenza di un nuovo identificatore in un assegnazione introduce implicitamente una nuova variabile. Nelle valutazioni possono comparire solo variabili già implicitamente dichiarate e che sono quindi alias di una qualche espressione.

Per chiarire ulteriormente quanto detto, l'esecuzione del seguente programma nel linguaggio appena descritto:

```
100 => capitale;
12.5 => interesse;
capitale; interesse;
capitale => saldo;
capitale * (1 + interesse/100) => saldo1;
saldo1 * (1 + interesse/100) => saldo2;
saldo; saldo1; saldo2;
```

produce il seguente output:

```
100.00000
12.50000
100.00000
112.50000
126.56250
```

5.2 Analisi lessicale

Come è già stato accennato nel capitolo 3, per descrivere le regole lessicali in modo univoco e conciso si può utilizzare un formalismo basato sugli automi a stati finiti. Tale soluzione ha anche il vantaggio di consentire una facile codifica del sottoprogramma che realizza l'analisi lessicale, detto *analizzatore lessicale* o *scanner*.

La codifica dell'analizzatore lessicale può essere realizzata seguendo due diversi metodi.

Uno, più sistematico, è quello gia ampiamente descritto nel già citato capitolo 3. Tale metodo fa fa ricorso ad una *variabile di stato* che rappresenta lo stato dell'automa, e consiste in un ciclo while che, in base al valore della variabile di stato, seleziona la porzione di codice che realizza le transizioni previste per quello stato, ed effettua le operazioni richieste per la generazione del token corrente.

Un secondo metodo consiste nel programmare direttamente il flusso di controllo sulla base dell'automa. Tale metodo è meno sistematico e può essere facilmente utilizzato solo quando il grafo che descrive l'automa non contiene cicli troppo complessi come nel caso in esame.

5.2.1 Lessico del linguaggio

Nella realizzazione di uno scanner il primo passo è quello di definire in modo preciso il lessico, cioè l'insiene dei token che devono essere riconosciuti.

Nel caso del linguaggio che stiamo considerando i token sono i seguenti simboli costituiti da un solo carattere:

```
+ - * / ( ) ;
```

il simbolo:

```
=>
```

costituito da due caratteri, le costanti numeriche e gli identificatori, che seguono le regole mostrate dai diagrammi di figura 5.1 e il simbolo corrisipondente al carattere di fine file che indicheremo con:

```
eof
```

Gli spazi e i caratteri di fine linea sono caratteri che possono separare token diversi, mentre i caratteri speciali non esplicitamente menzionati in precedenza non devono occorrere in un programma corretto.

5.2.2 Automa dell'analizzatore lessicale

Dall'elenco riportato nel precedente paragrafo si possono facilmente determinare quali sono i casi che si possono verificare durante il riconoscimento dei token. Tale analisi consente di costruire l'automa a stati che descrive il comportamento dell'analizzatore lessicale. L'automa corrispondente al lessico del linguaggio in esame è mostrato in figura 5.2.

Nell'automa in figura 5.2 non è stato considerato il caso in cui si presenti in input un carattere errato. I casi di errore sono tuttavia implicitamente specificati come quei casi non esplicitamente previsti. In fase di codifica dello scanner si potrà procedere, se il caso, a introdurre una gestione dei possibili errori.

5.2.3 L'analizzatore lessicale

La stesura del codice completo dell'analizzatore lessicale dell'interprete che vogliamo implementare è lasciata come esercizio. Aggiungiamo solo alcuni suggerimenti che possono facilitare il compito.

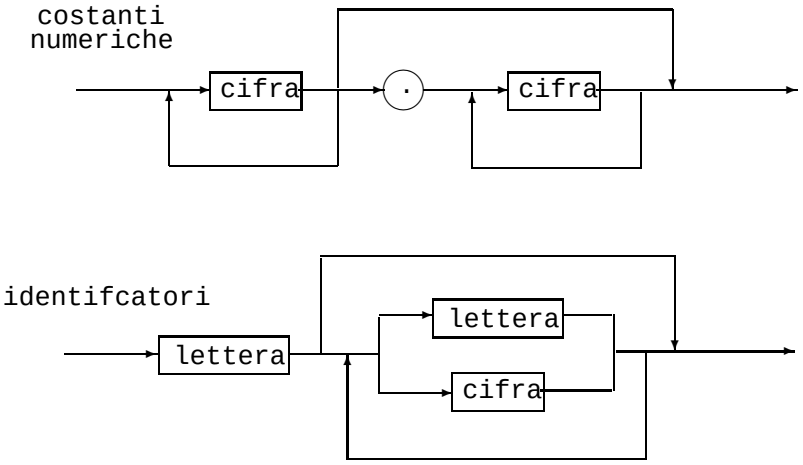


Figura 5.1: Regole lessicali per le costanti numeriche e gli identificatori.

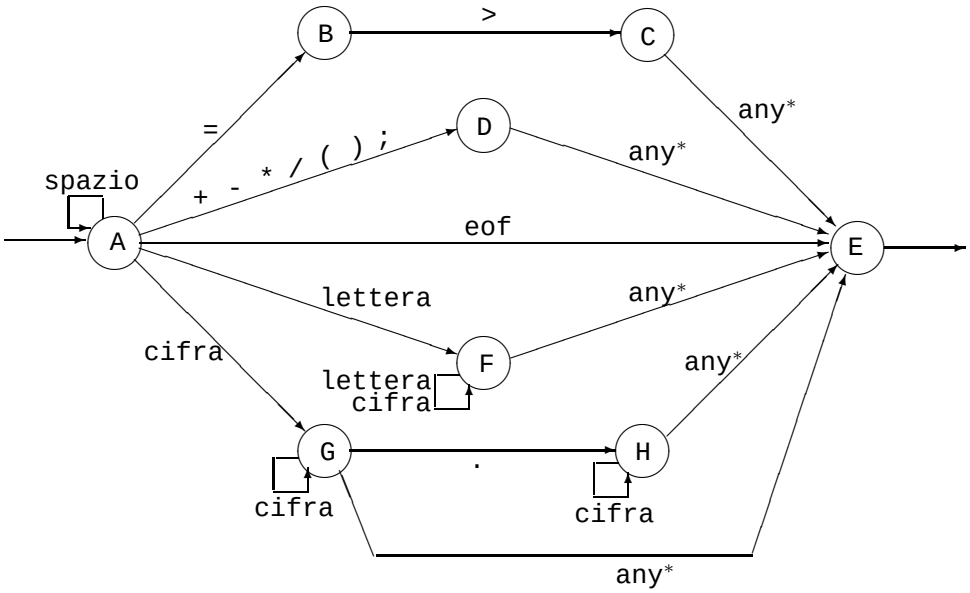


Figura 5.2: Automa a stati finiti dell'analizzatore lessicale.

Oltre alla procedura che implementa l'automa di figura 5.2, conviene implementare anche una procedura di inizializzazione e una procedura che provvede alla effettiva acquisizione dei caratteri. In tali procedure conviene leggere la sequenza in ingresso una riga per volta invece che un carattere per volta per consentire di visualizzare l'intera linea prima di procedere alla sua analisi. Questo consente di segnalare eventuali errori mostrando all'utente il punto in cui l'errore è stato rilevato.

5.3 Analisi sintattica

Come sappiamo, per implementare un analizzatore sintattico occorre disporre di una descrizione non ambigua della sintassi del linguaggio da analizzare. Useremo a tal fine i diagrammi sintattici che pur essendo del tutto equivalenti ad altri formalismi, come ad esempio la forma normale di Bakus basata su produzioni, ha il vantaggio di risultare di più immediata lettura.

5.3.1 Diagrammi sintattici

Nella figura 5.3 sono riportati i diagrammi sintattici corrispondenti al linguaggio che stiamo considerando. Nei diagrammi, i *simboli terminali* del linguaggio (cioè i simboli coincidenti con i simboli lessicali) sono racchiusi in cerchi o ovali, mentre i *simboli non terminali* sono racchiusi in rettangoli e a ciascuno di essi corrisponde un diagramma sintattico che ne descrive la struttura. Come si può facilmente osservare la struttura sintattica del linguaggio è in parte ricorsiva. È questa una caratterisitica generale delle sintassi (almeno di quelle non banali) che impedisce di utilizzare gli automi a stati finiti per realizzare i corrispondenti analizzatori.

Una caratteristica della sintassi descritta dai diagrammi di figura 5.3 è che ad ogni biforcazione la conoscenza del solo token successivo è sufficiente a decidere quale via prendere. Nel capitolo 4 abbiamo visto che quando i diagrammi che descrivono la sintassi di un linguaggio verificano tale proprietà è possibile realizzare facilmente il corrispondente parser sotto forma di un insieme di procedure e funzioni mutuamente ricorsive, una per ciascun diagramma.

5.3.2 L'albero sintattico del programma

L'obiettivo del parser è quello di costruire il cosiddetto *albero sintattico* del programma. L'albero sintattico è un albero che descrive la struttura sintattica analizzata. Esso rappresenta la base su cui i compilatori e gli interpreti operano per giungere, rispettivamente, alla traduzione e all'esecuzione del programma in input.

Ciascun nodo di un albero sintattico corrisponde a un simbolo (terminale o non) della sintassi. Nel caso il simbolo sia terminale il nodo è una foglia che contiene tutte le informazioni necessarie ad identificare il simbolo rappresentato. Nel caso che il simbolo sia non terminale il nodo contiene ancora le informazioni necessarie a identificare il simbolo rappresentato, ma ha un figlio per ogni componente della sua struttura secondo quando specificato dal corrispondente diagramma[1]. Ad esempio, con riferimento al diagramma sintattico corrispondente al simbolo non terminale $statement$, l'albero sintattico corrispondente sarà:

- una foglia contenente il simbolo `eof` nel caso si sia seguita la via in alto;
- un nodo contenente il simbolo `=>` con due figli corrispondenti rispettivamente alla variabile e all'espressione, nel caso si sia seguita la via al centro;
- un albero vuoto nel caso si sia seguita la via in basso[2].

Per concludere questa breve intrduzione all'analisi sintattica mostriamo nella figura 5.4 una espressione che segue la sintassi del nostro linguaggio e il corrispondente albero generato dall'analizzatore sintattico. Si noti che in generale

[1] In realtà tale regola ammette numerose eccezioni essendo possibili in pratica diverse semplificazioni. Un caso che vale la pena sottolineare è quello in cui la struttura sintattica di un simbolo non terminale è una lista. In tal caso il simbolo viene generalmente rappresentato da un nodo con due figli, il primo corrispondente al primo componente della struttura e il secondo corrispondente alla lista dei successivi. Una tale soluzione è stata ad esempio impiegata nel nostro caso per rappresentare le espressioni.

[2] In questo caso il simbolo lessicale `;` non viene rappresentato nell'albero sintattico perché esso ha in realtà solo il compito di separare le frasi tra loro. Simboli come questo vengono anche detti in gergo "zucchero sintattico" perché, una volta che l'analisi sintattica è stata completata e i diversi componenti sintattici sono stati identificati e rappresentati esplicitamente mediante alberi sintattici distinti, essi possono essere eliminati senza problemi. Nel nostro caso sono zucchero sintattico anche le parentesi nelle espressioni, che infatti non sono rappresentate nei corrispondenti alberi sintattici.

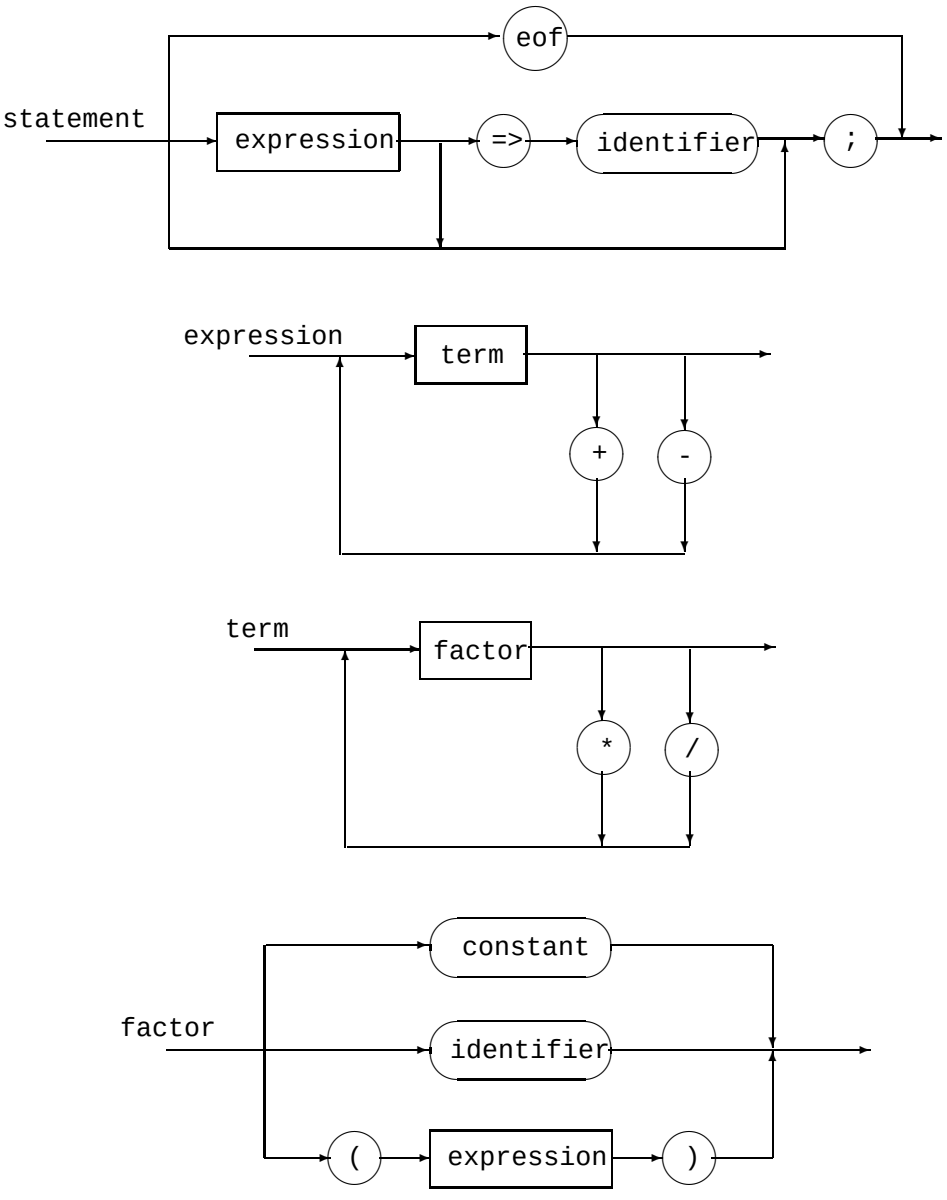


Figura 5.3: Descrizione della sintassi mediante diagrammi sintattici.

```
(5 + a) * 3.2 - b/2 => c
```

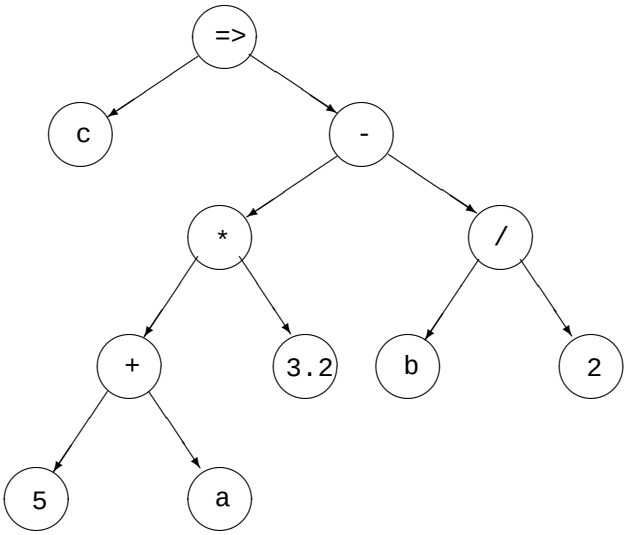


Figura 5.4: Albero sintattico di una frase.

l'albero sintattico non è necessariamente un albero binario, anche se nel caso che stiamo considerando è sufficiente impiegare un albero di questo tipo.

5.3.3 L'analizzatore sintattico

Anche l'implementazione dell'analizzatore sintattico dell'interprete è lasciata come esercizio. Per ottenere un programma con funzionalità realistiche nell'implementare i sottoprogrammi corrispondenti ai diagrammi di figura 5.3 è opportuno includere il codice per la gestione degli errori. Tale inclusione non modifica però la struttura dell'analizzatore che, come illustrato nel capitolo 4, segue rigorosamente la struttura delle regole sintattiche.

5.4 Interpretazione del programma

Una volta realizzati gli analizzatori lessicale e sintattico, l'interprete consiste in un programma che:

- inizializza una tabella dinamica che implementa la memoria dell'interprete;
- per ogni istruzione presente nel programma:
  - costruisce il corrispondente albero sintattico
  - visita tale albero eseguendo le azioni elaborative da esso descritte.

La visita dell'albero sintattico di un'istruzione può comportare la creazione di nuove variabili (aggiunta di un'entry nella memoria), la valutazione di un'espressione (visita in post-ordine del corrispondente albero sintattico) e l'assegnazione di un'espressione a una variabile (aggiornamento della memora). Quest'ultima azione si può realizzare associando alla variabile la rappresentazione dell'albero sintattico dell'espressione.

Anche l'impementazione di questa parte dell'interprete è lasciata come esercizio.

Capitolo 6

Progetto di un analizzatore (bozze, v. 1.0)

6.1 Introduzione

In questo capitolo viene suggerito lo svolgimento di un progetto consistente nell'implementare le componenti base di un compilatore. Viene suggerito come linguaggio da analizzare un sottoinsieme del linguaggio Pascal (vedi ad esempio il riferimento [Jensen & Wirth 1975] in bibliografia). Il sottoinsieme è minimo, ma sufficiente a consentire lo sviluppo di strumenti di un certo interesse didattico. Essendo un sottoinsieme del Pascal, la semantica del linguaggio è determinata dalla semantica del Pascal.

6.2 Un esempio

Quello che segue è un programma di esempio.

```
program example ( input, output );
  var x, y: integer;
  function gcd ( a, b: integer ) : integer;
    begin
      if b = 0 then gcd := a
      else gcd := gcd(b, a mod b)
    end;
begin
  read(x, y);
  write(gcd(x, y))
end.
```

6.3 Sintassi

Quella che segue è una grammatica per il sottoinsieme del Pascal considerato. La grammatica può essere modificata in modo da eliminare la ricorsione a sinistra e consentire così l'impiego di un analizzatore sintattico discendente ricorsivo.

program → **program id** (*identifier_list*) ;
declarations
subprogram_declarations
compound_statement
.

identifier_list → **id** | *identifier_list* , **id**

declarations → *declarations* **var** *identifier_list* : *type*
| ϵ

type → *standard_type*
| **array** [**num** . . **num**]

standard_type → **integer**
| **real**

subprogram_declarations → *subprogram_declarations subprogram_declaration* ;
| ϵ

subprogram_declaration → *subprogram_head declarations compound_statement*

subprogram_head → **function id** *arguments* : *standard_type* ;
| **procedure id** *arguments*

arguments → (*parameter_list*)
| ϵ

parameter_list → *identifier_list* : *type*
| *parameter_list* ; *identifier_list* : *type*

compound_statement → **begin** *optional_statements* **end**

optional_statement → *statement_list*
| *epsilon*

statement_list → *statement*
| *statement_list* ; *statement*

statement → *variable* **assignop** *expression*
| *procedure_statement*
| *compound_statement*
| **if** *exprexpression* **then** *statement* **else** *statement*
| **while** *expression* **do** *statement*

variable → **id**
| **id** [*expression*]

procedure_statement → **id**
| **id** (*expression_list*)

$$
\begin{array}{lcl}
expression\_list & \rightarrow & expression \\
 & | & expression\_list\ ,\ expression \\
 & & \\
expression & \rightarrow & simple\_expression \\
 & | & simple\_expression\ \mathbf{relop}\ simple\_expression \\
 & & \\
simple\_expression & \rightarrow & term \\
 & | & sign\ term \\
 & | & simple\_expression\ \mathbf{addop}\ term \\
 & & \\
term & \rightarrow & factor \\
 & | & term\ \mathbf{mulop}\ factor \\
 & & \\
factor & \rightarrow & \mathbf{id} \\
 & | & \mathbf{id}\ (\ expression\_list\ ) \\
 & | & \mathbf{num} \\
 & | & (\ expression\ ) \\
 & | & \mathbf{not}\ factor \\
 & & \\
sign & \rightarrow & + \mid - \\
\end{array}
$$

6.4 Regole lessicali

Per la definizione dei token, valgono le seguenti regole lessicali:

1. I commenti sono delimitati dalle parentedi graffe aperta e chiusa. Un commento non può contenere una parentesi graffa aperta. I commenti possono comparire prima o dopo un token qualsiasi.
2. Gli spazi bianchi tra i token sono opzionali, con l'eccezione che le parole chiave devono essere precedute e seguite da almeno uno spazio bianco, un carattere di fine linea, l'inizio del programma, o il punto finale.
3. Il token **id** che rappresenta un generico identificatore è costituito da una lettera dell'alfabeto inglese seguita da lettere o cifre. Si può decidere di limitare la massima lunghezza di un identificatore per esigenze implementative.
4. Il token **num** che rappresenta una costante numerica, è costituito da una sequenza di cifre senza segno. È possibile prevedere anche il caso più complesso di costanti reali, anche in forma esponenziale.
5. Le parole chiave sono riservate (non possone essere usate come identificatori) ed appaiono in grassetto nella grammatica.
6. Gli operatori di relazione (**relop**) sono: `=`, `<>`, `<`, `<=`, `>=`, e `>`. Il token `<>` denota l'operatore $\neq$.
7. Gli operatori **addop** sono `+`, `-`, e `or`.
8. Gli operatori **mulop** sono `*`, `/`, `div`, `mod`, e `and`.
9. Il lessico di **assignop** è: `:=`.

Bibliografia

Sono riportati di seguito alcuni testi che possono essere consultati per approfondire i principali argomenti trattati.

1. T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stain, *Introduction to Algorithms - Second Edition*, MIT Press, 2001.
2. T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduzione agli Algoritmi*, Jackson Libri, 1999
3. N. Wirth, *Principi di Programmazione Strutturata*, ISEDI.
4. N. Wirth, *Algoritmi + Strutture Dati = Programmi*, Tecniche Nuove.
5. F. Preparata, R. Yeh, *Introduzione alle Strutture Discrete*, Boringhieri.
6. A.H. Aho, R. Sethi, J.D. Ullman, *Compilers - Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts, USA, 1986.
7. K. Jensen, N. Wirth, *Pascal. Manuale e standard del linguaggio*, Mc-Grow Hill, 1975.