Capitolo 3

Analisi Sintattica

Ogni linguaggio di programmazione ha delle regole che prescrivono la struttura sintattica dei programmi ben formati del linguaggio. Ad esempio in Pascal un programma corretto è formato da blocchi, i blocchi sono formati da statements, gli statements da espressioni, le espressioni dai token e così via.

La sintassi dei linguaggi di programmazione può essere ed è convenientemente specificata tramite grammatiche libere dal contesto (*context-free grammars*). Lo stesso formalismo viene a volte chiamato anche notazione BNF (Backus-Naur Form).

Le grammatiche offrono vantaggi significativi sia ai progettisti dei linguaggi di programmazione che agli implementatori dei corrispondenti compilatori. Vediamo i principali:

- Una grammatica dà una specifica sintattica precisa e facile da capire di un linguaggio.

- Per alcune classi di grammatiche possiamo automaticamente costruire dei parser efficienti che determinano se un certo programma sorgente è sintatticamente ben formato e, in caso positivo, ne rendono disponibile la struttura gerarchica ad albero. Inoltre, il processo di costruzione del parser stesso riesce a rilevare ambiguità nella definizione della grammatica o ad individuare alcuni costrutti critici difficili da analizzare. Questo tipo di problemi possono facilmente passare inosservati nella fase iniziale di progetto di un linguaggio e questa possibilità offerta dai generatori di parser è un valido supporto per i progettisti.

- Una grammatica ben progettata impone una struttura al linguaggio e questa struttura è utile per la traduzione dei costrutti in codice oggetto

(sia intermedio che della macchina ospite). Questo processo di traduzione può essere fatto automaticamente dando le specifiche giuste ai generatori di parser.

- Un linguaggio può evolvere nel tempo acquisendo nuovi costrutti e/o eseguendo nuove operazioni. Questi nuovi costrutti possono essere aggiunti più facilmente se il linguaggio è stato implementato sulla base di una grammatica.

In questo capitolo introdurremo le grammatiche libere dal contesto e le relative nozioni di albero di derivazione, derivazioni, ambiguità. Poi ci dedicheremo a studiare alcune metodologie di parsing per alcune classi di grammatiche che sono sufficientemente espressive da specificare i costrutti tipici di un linguaggio di programmazione.

3.1 Il ruolo del parser

Come abbiamo già visto nel Capitolo 1 il parser ottiene una stringa di token dall'analizzatore lessicale e tenta di costruire l'albero di derivazione o parse tree della stringa di token in accordo alla grammatica del linguaggio. La Figura 3.1 mostra le interazioni del parser. Ricordiamo che in una reale implementazione molto difficilmente il parse tree sarà creato esplicitamente. Solo per comodità e chiarezza di esposizione assumiamo che il risultato della fase di analisi sia un parse tree e che poi questo venga dato come input al resto delle fasi.

Nella figura viene indicato che il parse tree è l'input del resto del front-end del compilatore. Il front-end di un programma, in generale, è la parte che si occupa del reperimento e dell'analisi dell'input in modo da porlo in una rappresentazione interna adatta per essere processato dalla parte di "calcolo" dei risultati del programma stesso (chiamata back-end). Nel caso del compilatore il front-end si occupa di trasformare il codice sorgente in codice tradotto nel linguaggio intermedio scelto (ad esempio il codice a tre indirizzi). Quindi fanno parte del front-end le prime quattro fasi della compilazione secondo la suddivisione che abbiamo dato nel Capitolo 1.

Durante il parsing, in una reale implementazione, possono essere effettuate molte altre operazioni "in parallelo" che ricadono concettualmente in altre fasi:

- Reperimento delle informazioni relative ai token e loro memorizzazione nella tabella dei simboli.

- Type Checking e altre analisi semantiche statiche.

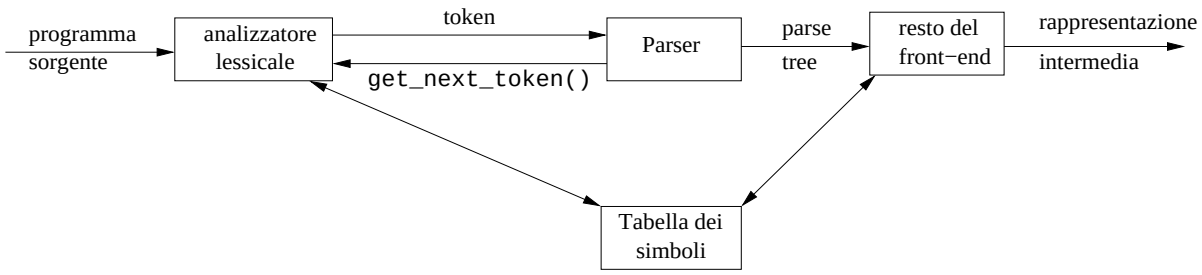


Figura 3.1: Il ruolo del parser.

- Generazione del codice intermedio.

Ci aspettiamo che il parser riporti eventuali errori di sintassi se il programma sorgente non rispetta le regole della grammatica. La gestione degli errori in questa fase è molto importante dato che la maggior parte degli errori che un compilatore può scoprire sono di tipo sintattico. Noi non ci occuperemo dettagliatamente di questo aspetto. Tuttavia nel libro di riferimento si possono trovare diverse tecniche di recovering in caso di errore.

Ci sono tre tipi generali di metodi per l'analisi di grammatiche libere dal contesto:

1. Metodi di parsing universali come ad esempio l'algoritmo di Cocke-Younger-Kasami o l'algoritmo di Earley. Questi algoritmi possono fare il parsing di una qualsiasi grammatica. Tuttavia, proprio per la loro generalità, questi metodi sono troppo inefficienti per essere usati all'interno di un compilatore.

2. Top-Down parsing. Costruiscono il parse tree dall'alto (la radice) verso il basso (le foglie) leggendo l'input da sinistra a destra.

3. Bottom-Up parsing. Costruiscono il parse tree dal basso (le foglie) verso l'alto (la radice) leggendo l'input da sinistra a destra.

I metodi di parsing top-down o bottom-up più efficienti funzionano solo per certe sottoclassi di grammatiche, ma alcune di queste sottoclassi, come ad esempio le grammatiche LL e LR, sono sufficientemente espressive per descrivere la maggior parte dei costrutti dei linguaggi di programmazione. I parser implementati "a mano" spesso sono quelli per grammatiche LL, mentre i parser per la classe più vasta delle grammatiche LR di solito sono costruiti tramite dei tool automatici.

3.2 Grammatiche libere dal contesto

Diamo ora una definizione precisa di grammatica libera dal contesto.

Definizione 3.1 *Una* Grammatica libera dal contesto *è una tupla* $G = \langle \Sigma, V, S, P \rangle$ *dove:*

- Σ *è un insieme finito dei simboli di alfabeto o* Simboli Terminali
- V *è un insieme finito di simboli che rappresentano* Categorie Sintattiche *o simboli* non terminali
- $S \in V$ *è un simbolo di categoria sintattica indicato come* iniziale *o* principale
- P *è un insieme finito di regole, chiamate* Produzioni, *della forma*

$$A \to X_1 X_2 \cdots X_k$$

dove:

  - $A \in V$ *è la* testa *o parte sinistra della produzione*
  - $\forall i \in \{1, \ldots, k\}.\ X_i \in (V \cup \Sigma)$. *La stringa* $X_1 X_2 \cdots X_k \in (V \cup \Sigma)^*$ *si chiama* corpo *o parte destra della produzione.*

Il corpo di una produzione può anche essere vuoto. In questo caso la produzione viene scritta $A \to \epsilon$.

In alcuni testi l'operatore `::=` *è usato a posto della freccia* $\to$ *nella scrittura delle produzioni. Le due notazioni sono equivalenti.*

Il compito principale di una grammatica è quello di generare stringhe di simboli terminali. Il processo di generazione delle stringhe può avvenire tramite costruzione di alberi di derivazione o, equivalentemente, derivazioni.

Prima di tutto fissiamo alcune convenzioni di notazione che ci permetteranno di illustrare più chiaramente i concetti che introdurremo via via nel seguito. Associamo degli insiemi di simboli a certi tipi di oggetti formali che tratteremo spesso. In questo modo eviteremo di dover specificare, ogni volta che useremo un simbolo nuovo, che cosa quel simbolo sta ad indicare.

1. I seguenti simboli indicano simboli **terminali** della grammatica:
   - Lettere minuscole all'inizio dell'alfabeto:

$$a, b, c, \ldots, a', a'', \ldots, a_1, a_2, \ldots$$

- Simboli di operatori: $+$, $-$, $*$, ...
- Simboli di punteggiatura come virgole, punti e virgola, due punti, punti, ...
- Le cifre 0,1,...,9.
- Stringhe in grassetto come **id** o **if**.

2. I seguenti simboli indicano simboli **non terminali** della grammatica:

   - Lettere maiuscole all'inizio dell'alfabeto:

   $$A, B, C, \ldots, A', A'', \ldots, A_1, A_2, \ldots$$

   - La lettera S che in genere indica il simbolo iniziale della grammatica (a meno che non sia specificato diversamente)
   - Stringhe in minuscolo e in corsivo come ad esempio *expr* o *stmt*.
   - Stringhe qualsiasi tra $<$ e $>$ come ad esempio $<$ OP $>$ oppure $<$ AB $>$.

3. Le lettere maiuscole alla fine dell'alfabeto:

   $$X, Y, Z, \ldots, X', X'', \ldots, X_1, X_2, \ldots$$

   rappresentano un simbolo terminale o non terminale, cioè simboli nell'insieme $\Sigma \cup V$.

4. Le lettere minuscole alla fine dell'alfabeto:

   $$u, v, w, x, y, z, \ldots, u', x', \ldots, x_1, v_2, \ldots$$

   rappresentano *stringhe* di soli simboli terminali (anche vuote), cioè elementi di Σ^*.

5. Le lettere minuscole dell'alfabeto greco:

   $$\alpha, \beta, \delta, \gamma, \ldots, \alpha', \alpha'', \ldots, \beta_1, \gamma_2, \ldots$$

   rappresentano stringhe, anche vuote, formate da simboli terminali o non terminali, cioè elementi di $(V \cup \Sigma)^*$. Tramite le convenzioni viste fino ad ora, vediamo che una generica produzione della grammatica può essere indicata con $A \to \alpha$.

6. Se $A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_k$ sono tutte le produzioni con lo stesso simbolo a sinistra A (le chiamiamo A-produzioni), allora possiamo scrivere equivalentemente $A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$. Le $\alpha_i, i = 1, 2, \ldots, k$ sono chiamate *alternative* per A.

7. A meno che non si specifichi diversamente, assumeremo che la parte sinistra della prima produzione data per una grammatica sia il simbolo iniziale.

3.2.1 Alberi di derivazione

Il modo più semplice di generare una stringa da una grammatica è quello di costruire un albero di derivazione.

Definizione 3.2 (Albero di derivazione) *Sia $G = \langle \Sigma, V, S, P \rangle$ una grammatica libera dal contesto. Un albero di derivazione di G è un albero in cui i nodi sono etichettati con i simboli della grammatica $\Sigma \cup V$ e sono rispettate le seguenti proprietà:*

- *La radice dell'albero è etichettata con il simbolo iniziale S*
- *Ogni foglia dell'albero è etichettata con un simbolo terminale*
- *Ogni nodo interno dell'albero è etichettato con un simbolo non terminale A i cui figli, presi da sinistra a destra, sono etichettati con i simboli $X_1 X_2 \cdots X_n$ della parte destra di una qualche produzione $A \to X_1 X_2 \cdots X_n$ in P.*

La stringa w che si ottiene concatenando i simboli associati alle foglie di un albero di derivazione, andando da sinistra verso destra, si chiama stringa associata *all'albero di derivazione. Diciamo anche che un albero è un albero di derivazione per w se w è la sua stringa associata.*

Nel contesto dell'analisi sintattica l'albero di derivazione viene anche chiamato parse tree.

Esempio 3.3 *Consideriamo la seguente grammatica:*

$$E \to E + E \mid E * E \mid (E) \mid -E \mid \mathbf{id}$$

I simboli terminali sono $(,), +, *, -, \mathbf{id}$. L'unico simbolo non terminali è E che è, ovviamente, anche il simbolo iniziale.

L'albero in Figura 3.2 è un albero di derivazione la cui stringa associata è $\mathbf{id} + \mathbf{id} * \mathbf{id}$.

Tramite gli alberi di derivazione e la nozione di stringa associata possiamo definire precisamente cosa si intende per linguaggio generato da una grammatica.

Definizione 3.4 (Linguaggio generato) *Sia $G = \langle \Sigma, V, S, P \rangle$ una grammatica libera dal contesto. Il linguaggio generato da G è indicato con $L(G)$ ed è l'insieme delle stringhe di $w \in \Sigma^*$ tali che esiste un albero di derivazione di G la cui stringa associata è w.*

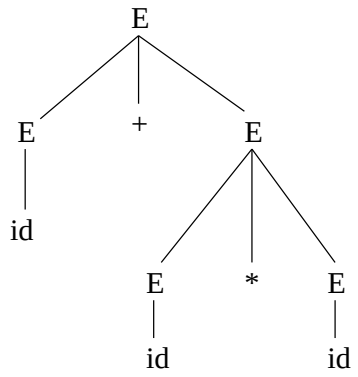


Figura 3.2: Un albero di derivazione per **id** + **id** ∗ **id**.

3.2.2 Derivazioni

Definiamo ora il concetto di *derivazione* di una stringa a partire da un simbolo di categoria sintattica. Questa nozione è un'alternativa a quella degli alberi di derivazione. Anch'essa può essere usata per definire il linguaggio delle stringhe generate da una grammatica.

Sia $G = \langle \Sigma, V, S, P \rangle$ una grammatica libera dal contesto. Sia α una stringa che contiene almeno un simbolo non terminale A, cioè $\alpha = \delta A \gamma$. Possiamo applicare ad α un *passo di derivazione* utilizzando una produzione di P che ha A come testa. Il passo consiste nel riscrivere α sostituendo al simbolo di categoria sintattica A che abbiamo messo in evidenza il corpo della produzione scelta.

Definizione 3.5 (Passo di Derivazione) *Sia $G = \langle \Sigma, V, S, P \rangle$ una grammatica libera dal contesto e sia α una stringa che contiene almeno un simbolo non terminale A. Se $A \to X_1 X_2 \cdots X_k \in P$, allora possiamo riscrivere:*

$$\alpha = \delta A \gamma \Rightarrow_G \delta X_1 X_2 \cdots X_k \gamma$$

Il simbolo $\Rightarrow_G$ rappresenta un passo di derivazione *per la grammatica G. Spesso, se la grammatica che consideriamo è chiara dal contesto, ometteremo il pedice G.*

Definizione 3.6 (Derivazione) *Sia $G = \langle \Sigma, V, S, P \rangle$ una grammatica libera dal contesto. Una derivazione di una stringa $w \in \Sigma^*$ a partire da S è una sequenza:*

$$S = \alpha_0 \Rightarrow_G \alpha_1 \Rightarrow_G \alpha_2 \cdots \Rightarrow_G \alpha_n = w$$

dove, per ogni $i \in \{0, 1, \ldots, n-1\}$, $\alpha_i \Rightarrow_G \alpha_{i+1}$ è un passo di derivazione che riscrive un qualche simbolo non terminale di α_i.

Si noti che se $i < n$ allora $\alpha_i \in (\Sigma \cup V)^*$. Una stringa di questo tipo, generata cioè con un certo numero di passi di derivazione a partire dal simbolo iniziale, viene chiamata forma sentenziale.

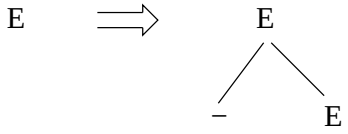
Una derivazione lunga 0 passi è la sequenza composta solo dal simbolo S.

Per indicare una derivazione di zero o più passi da S ad una certa stringa α scriviamo $S \stackrel{*}{\Rightarrow}_G \alpha$, mentre $S \stackrel{+}{\Rightarrow}_G \alpha$ indica una derivazione lunga almeno 1 passo da S a α.

Possiamo definire il linguaggio generato dalla grammatica anche con la nozione di derivazione dicendo che $L(G) = \{w \in \Sigma^* \mid S \stackrel{*}{\Rightarrow}_G w\}$. Questa definizione è equivalente a quella data usando gli alberi di derivazione, nel senso che l'insieme di stringhe definito con le derivazioni è lo stesso di quello definito con gli alberi di derivazione. Per convincersi di questo basta notare che la costruzione di una derivazione induce in maniera naturale la costruzione di un albero di derivazione. Prendiamo ad esempio una derivazione per la stringa $-(\mathbf{id} + \mathbf{id})$ con la grammatica dell'Esempio 3.3:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

Ogni passo di derivazione si riflette, nella costruzione del corrispondente albero di derivazione, nell'aggiunta dei figli al nodo corrispondente al simbolo non terminale che si sta riscrivendo. Tali figli sono i simboli del corpo della produzione che si sta usando per il passo di derivazione. All'inizio il solo simbolo E, al passo zero di derivazione, corrisponde alla radice dell'albero. Al primo passo viene applicata la produzione $E \to -E$ e quindi vengono aggiunti due figli ($-$ ed E) alla radice:



Tutte le altre trasformazioni sono riportate in Figura 3.3

Si noti che, durante la derivazione di una stringa dal simbolo iniziale della grammatica, ad ogni passo occorre fare due scelte. Supponiamo di avere una forma sentenziale β (cioè $S \stackrel{*}{\Rightarrow} \beta$) che non sia una stringa di soli terminali. In generale, per fare un passo di derivazione $\beta \Rightarrow \beta'$, bisogna:

1. Individuare un simbolo non terminale in β che sarà il candidato per la riscrittura: $\beta = \alpha A \gamma$

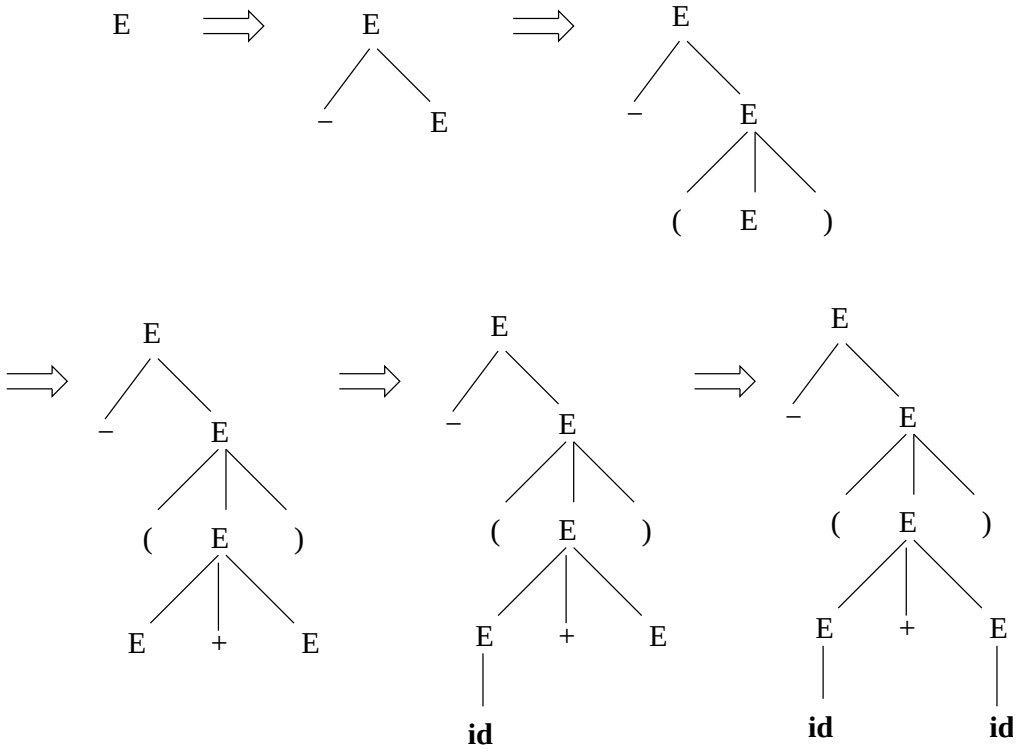


Figura 3.3: Da una derivazione a un albero di derivazione

2. Scegliere una produzione $A \to \delta$, fra le eventuali possibili scelte per A, con la quale effettuare la riscrittura $\beta = \alpha A \gamma \Rightarrow \alpha\delta\gamma$

Per il primo tipo di scelta possiamo fissare una regola che identifichi univocamente ad ogni passo il simbolo non terminale da riscrivere. Questa restrizione vedremo che sarà molto utile nell'ambito del parsing di una stringa rispetto ad una grammatica. Le due regole principali che si usano sono le seguenti:

- **Derivazione Leftmost** Ad ogni passo si riscrive il simbolo non terminale A di una forma sentenziale β che sta più a sinistra: $\beta = xA\alpha$ (ricordiamo che per x si intende una stringa di soli simboli terminali). Le forme sentenziali che si trovano lungo una derivazione leftmost si chiamano *forme sentenziali sinistre*. Per indicare che ad un passo di derivazione si è applicata la regola leftmost utilizziamo la notazione $xA\alpha \Rightarrow_{lm} x\delta\alpha$[1]. In caso di più passi di derivazione leftmost usiamo $\overset{*}{\Rightarrow}_{lm}$ ($\overset{+}{\Rightarrow}_{lm}$) per zero o più passi (per uno o più passi). Si noti che una derivazione è leftmost se e solo se tutti i passi di cui è composta sono leftmost.

- **Derivazione Rightmost** Ad ogni passo riscriviamo il simbolo non terminale più a destra. Le forme sentenziali sono dette forme sentenziali destre e la notazione usata è $\Rightarrow_{rm}$ con le stesse estensioni viste nel caso leftmost: $S \overset{*}{\Rightarrow}_{rm} \alpha Ax \Rightarrow_{rm} \alpha\delta x$.

3.2.3 Ambiguità

Una questione importante che riguarda le grammatiche libere dal contesto nel loro uso come generatori di linguaggi di cui viene effettuato il parsing è l'ambiguità.

Intuitivamente possiamo vedere la scrittura di una grammatica come la definizione di un algoritmo (ricorsivo) per generare le stringhe di un linguaggio. Questo algoritmo usa le produzioni e costruisce un albero di derivazione (o una derivazione) per una certa stringa data. È facile vedere che durante la generazione di una stringa l'algoritmo (cioè la grammatica) impone certe scelte che riflettono la struttura delle produzioni. La questione dell'ambiguità può essere vista intuitivamente nel seguente modo: la grammatica è ambigua se le produzioni permettono di seguire almeno due strade differenti per generare una certa stringa. Si noti che basta che esista una sola stringa

[1] Omettiamo il simbolo G che indica la grammatica che stiamo usando. In questi casi sarà opportuno sincerarsi che la grammatica G in questione sia chiaramente specificata nel contesto.

per cui questo è vero e tutta la grammatica diventa ambigua. Da questo approccio intuitivo si può anche ricavare una giustificazione del fatto che fare il parsing di grammatiche ambigue risulta molto difficoltoso: il parser che cerca di analizzare una stringa che può essere generata in due (o più) modi non sa decidere quale strada seguire, fra le possibili, nella ricostruzione dell'albero. Inoltre la struttura "troppo libera" delle produzioni rende difficoltosa l'analisi anche delle stringhe che hanno un solo albero di derivazione.

Dopo questa premessa informale definiamo formalmente quando una grammatica è ambigua:

Definizione 3.7 (Ambiguità) *Una grammatica libera dal contesto G si dice ambigua se e solo se esiste una stringa $w \in L(G)$ tale che esistono due alberi di derivazione T e T' di G con le seguenti proprietà:*

- *T e T' sono diversi*
- *T e T hanno w come stringa associata*

Di converso, G non è ambigua se per ogni stringa $w \in L(G)$ esiste uno ed un solo albero di derivazione di G che ha w come stringa associata.

La caratterizzazione dell'ambiguità è stata data usando gli alberi di derivazione. In effetti, visto lo stretto rapporto che c'è tra gli alberi di derivazione e le derivazioni, anche queste ultime possono essere usate per definire l'ambiguità. L'osservazione fondamentale è che se si fissa una regola per scegliere il simbolo da riscrivere ad ogni passo di derivazione allora due derivazioni diverse corrispondono ad alberi di derivazione diversi e due derivazioni uguali corrispondono allo stesso albero di derivazione.

Quindi, ad esempio prendendo la regola leftmost (o rightmost), si ha che una grammatica non è ambigua se e solo se per ogni stringa esiste una unica derivazione leftmost (o rightmost).

Esempio 3.8 *Consideriamo la seguente grammatica:*

$$E \to E + E \mid E * E \mid (E) \mid \mathbf{id}$$

Questa grammatica è ambigua poiché per la stringa $\mathbf{id} + \mathbf{id} * \mathbf{id}$ *esistono i due alberi di derivazione mostrati in Figura 3.4. Equivalentemente esistono due derivazioni leftmost. La seguente corrisponde all'albero in Figura 3.4(a):*

$$E \Rightarrow_{\text{lm}} E + E \Rightarrow_{\text{lm}} \mathbf{id} + E \Rightarrow_{\text{lm}} \mathbf{id} + E * E \Rightarrow_{\text{lm}} \mathbf{id} + \mathbf{id} * E \Rightarrow_{\text{lm}} \mathbf{id} + \mathbf{id} * \mathbf{id}$$

La seguente invece corrisponde all'albero in Figura 3.4(b):

$$E \Rightarrow_{\text{lm}} E * E \Rightarrow_{E} + E * E \Rightarrow_{\text{lm}} \mathbf{id} + E * E \Rightarrow_{\text{lm}} \mathbf{id} + \mathbf{id} * E \Rightarrow_{\text{lm}} \mathbf{id} + \mathbf{id} * \mathbf{id}$$

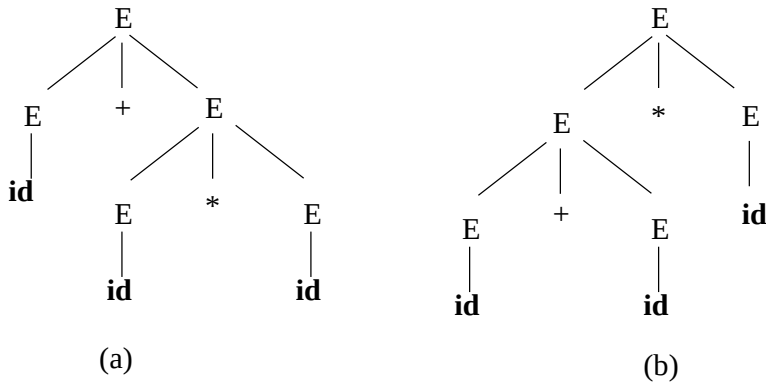


Figura 3.4: Due alberi di derivazione diversi per la stringa **id** + **id** ∗ **id**.

Un indizio che indica spesso che una grammatica è ambigua è il fatto che in almeno una produzione è usata la ricorsione "doppia", cioè il simbolo a sinistra della produzione occorre almeno due volte nella parte destra. È indicativo soprattutto se la grammatica genera un linguaggio con operatori binari come quello dell'esempio precedente.

In generale è molto difficile dimostrare che una grammatica *non* è ambigua poiché bisogna dimostrare l'unicità dell'albero di derivazione per *tutte* le stringhe del linguaggio. Dimostrare invece la non ambiguità è semplice: basta esibire un controesempio, cioè una stringa per cui esistono due alberi di derivazione oppure due derivazioni leftmost (o rightmost). Per capire se una grammatica è ambigua oppure no è bene innanzitutto cercare di capire il linguaggio generato dalla stessa e poi capire *come* vengono generate le stringhe: se c'è un algoritmo che impone delle scelte per ogni tipo di stringa generabile oppure se le produzioni lasciano abbastanza libertà tanto da consentire che una certa stringa o un certo gruppo di stringhe possano essere generate seguendo diverse strade.

Nella pratica, se si vuole scrivere una grammatica non ambigua per un certo linguaggio, è utile progettare le produzioni in modo tale che la generazione di ogni stringa del linguaggio debba seguire una ed una sola strada.

Se una grammatica risulta essere ambigua e deve essere usata per generare un parser è bene che sia disambiguata. Infatti molti metodi per generare parser (tra cui tutti quelli che vedremo noi) falliscono se la grammatica che viene considerata è ambigua.

La disambiguazione di una grammatica consiste nella riscrittura delle sue produzioni (anche introducendo o togliendo simboli non terminali) in modo tale da lasciare inalterato il linguaggio generato e da avere un solo albero di derivazione per tutte le stringhe, anche quelle per le quali esistevano più

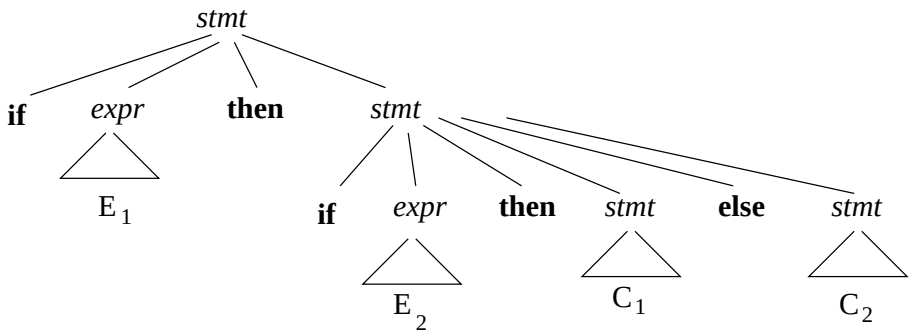


Figura 3.5: Un albero di derivazione per la stringa

alberi di derivazione associati nella grammatica di partenza.

Un costrutto ambiguo

In questa sezione consideriamo un esempio classico di ambiguità e successiva disambiguazione. Il costrutto che produce il problema è uno dei più usati in tutti i linguaggi di programmazione: il condizionale if-then-else.

L'ambiguità nasce dal fatto che un costrutto condizionale può avere un solo ramo oppure può essere completo e avere due rami. Prendiamo il seguente spezzone di grammatica del Pascal:

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
 & | & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\
 & | & \textbf{altri\_comandi}
\end{array}
$$

Consideriamo il comando **if** E_1 **then if** E_2 **then** C_1 **else** C_2 dove supponiamo che E_1, E_2, C_1 e C_2 corrispondono a costrutti corretti (espressioni e comandi). Possiamo individuare due alberi di derivazione per il comando dato. Ciò è dovuto al fatto che la grammatica permette di specificare comandi condizionali con e senza **else** senza nessun vincolo.

Nelle Figure 3.5 e 3.6 sono mostrati due alberi di derivazione diversi per la stringa data che sono dimostrazione del fatto che la grammatica che abbiamo considerato è ambigua.

Siamo interessati a disambiguare la grammatica. In Pascal e anche in altri linguaggi di programmazione simili si fissa una regola che permette di interpretare le stringhe come quella che abbiamo considerato sopra in una maniera univoca. La regola che si segue dice che ogni **else** deve essere associato al **then** non associato più vicino. Quindi l'albero di derivazione "giusto" è quello di Figura 3.5.

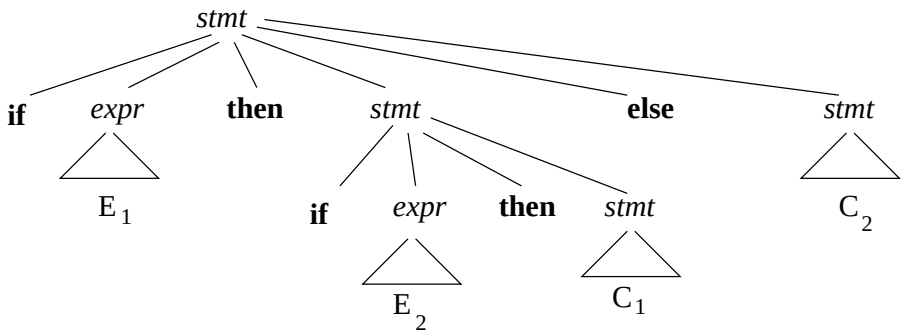


Figura 3.6: Un altro albero di derivazione per la stessa stringa

Questa regola di associazione può essere incorporata direttamente nella grammatica disambiguandola. L'idea è quella di dividere i comandi in due tipologie:

1. i comandi "matched", rappresentati dalla categoria sintattica *matched_stmt*, che sono i comandi in cui tutti i **then** hanno un **else** associato oppure non sono comandi condizionali (generati dalla categoria sintattica *non_cond*)

2. i comandi "unmatched", rappresentati dalla categoria sintattica *unmatched_stmt*, che sono tutti i comandi condizionali che hanno solo il ramo **then** seguito da qualsiasi comando oppure che hanno anche un ramo **else**, ma in questo caso il comando fra il **then** e l'**else** è un comando "matched" e il comando dopo l'**else** è "unmatched".

In questo modo ogni comando che appare tra un **then** ed un **else** non può finire con un **then** non associato seguito da un altro comando qualsiasi. Ecco la nuova grammatica:

stmt	$\rightarrow$	*matched_stmt* | *unmatched_stmt*
matched_stmt	$\rightarrow$	**if** *expr* **then** *matched_stmt* **else** *matched_stmt*
	|	*non_cond*
unmatched_stmt	$\rightarrow$	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

In Figura 3.7 è disegnato l'unico albero di derivazione per la stringa che abbiamo considerato sopra. Questa volta non si può costruire un albero simile a quello di Figura 3.6 perché fra il primo **then** e l'**else** non possiamo mettere un comando "unmatched" come un condizionale senza l'**else**.

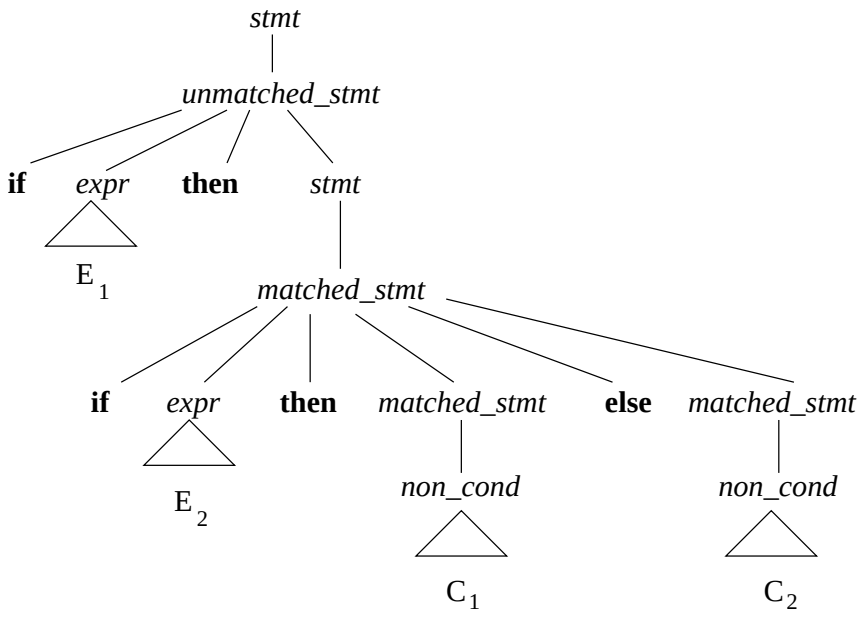


Figura 3.7: L'unico albero di derivazione per la stringa con la nuova grammatica.

3.3 Associatività e precedenza degli operatori

La struttura di un albero di derivazione per una certa stringa è molto importante nel contesto del parsing e dei compilatori in generale. Infatti, se una frase di un linguaggio di programmazione viene strutturata in maniera corretta rispetto alla sua semantica, la traduzione della stessa risulta estremamente semplice.

Per capire meglio questo aspetto facciamo l'esempio classico delle espressioni aritmetiche. Possiamo pensare di usare l'albero di derivazione di una certa espressione aritmetica per calcolarne il valore. Si consideri ad esempio l'albero in Figura 3.4(a) per la stringa **id**+**id**∗**id**. L'osservazione fondamentale è che ogni nodo interno etichettato E corrisponde ad una sottoespressione il cui valore può essere calcolato in base ai valori associati ai nodi figli. Ad esempio se il nodo ha un unico figlio etichettato **id** allora il valore ad esso associato è il valore associato all'identificatore in memoria (stiamo parlando della semantica di esecuzione del programma). Se il nodo ha tre figli etichettati con $E + E$ rispettivamente allora il suo valore sarà la somma del valore associato al primo figlio E con il valore associato al secondo figlio E (questi valori sono calcolati ricorsivamente). In questo modo il valore associato alla

radice è il valore dell'espressione aritmetica.

A questo punto è chiaro perché la struttura dell'albero si rivela fondamentale: deve rispecchiare le regole (semantiche) di valutazione delle espressioni aritmetiche affinché il valore associato alla radice sia quello giusto. Noi conosciamo queste regole: sono regole di associatività e di precedenza. Ad esempio, l'albero di Figura 3.4(b) non può essere usato per valutare l'espressione perché la sua struttura non rispetta la precedenza dell'operatore $*$ di moltiplicazione rispetto all'operatore $+$ di addizione.

In questa sezione vediamo come esprimere l'associatività e la precedenza fra operatori binari (cioè che prendono due argomenti) con le produzioni della grammatica. L'esempio che useremo via via sarà quello delle espressioni aritmetiche con l'addizione, la sottrazione, la moltiplicazione, la divisione e le parentesi tonde fra numeri formati da una sola cifra (per semplificare la trattazione e non introdurre dettagli inutili).

Innanzitutto chiariamo bene cosa intendiamo per associatività e precedenza.

- Associatività. Prendiamo ad esempio l'operatore $+$. In una espressione come $3 + 5 + 7$ non è chiaro a quale $+$ deve essere associato il numero 5 poiché ne ha uno a destra e uno a sinistra ($+$ è un operatore binario e quindi per essere applicato ha bisogno di due operandi: uno alla sua destra e uno alla sua sinistra in notazione infissa). La regola di associatività specifica proprio questo punto: se si stabilisce che il $+$ associa a sinistra allora il 5 viene associato al $+$ alla sua sinistra e la struttura sottintesa dell'espressione di cui sopra è la stessa di $(3+5)+7$ (dove la struttura è stata esplicitata con l'uso delle parentesi). Nel caso di associatività a destra la struttura sarebbe stata $3 + (5 + 7)$.

- Precedenza. Prendiamo due operatori classici con precedenza diversa: $+$ e $*$. In una espressione come $3 + 5 * 7$ ancora una volta non è chiaro a quale operatore deve essere associato il 5 centrale. La regola di precedenza risolve questo punto ponendo gli operatori su livelli diversi di precedenza. Se, come è di solito, il $*$ ha precedenza maggiore del $+$ allora il 5 viene associato al $*$ e quindi la struttura esplicitata dell'espressione è $3 + (5 * 7)$.

Per scrivere una grammatica le cui produzioni riflettano le scelte di precedenza e associatività bisogna innanzitutto definire i vari livelli di precedenza. In ogni livello di precedenza bisogna inserire uno o più operatori che hanno la stessa precedenza (ad esempio il $+$ ed il $-$ nelle espressioni aritmetiche) e per ogni livello si indica se l'associatività deve essere a destra o a sinistra. Per ogni livello definito si crea un simbolo non terminale della grammatica.

Illustriamo il procedimento di costruzione delle produzioni con un esempio. Prendiamo i seguenti livelli di precedenza fra gli operatori aritmetici:

1. Fattori (simbolo F), cioè gli operandi. Hanno il maggiore livello di precedenza per definizione.

2. Termini (simbolo T), cioè applicazione di $*$ o $/$. Hanno la stessa precedenza, inferiore a quella dei fattori. L'associatività è a sinistra (come è di solito nei linguaggi)

3. Espressioni (simbolo E), cioè applicazione di $+$ e $-$. Hanno la precedenza più bassa di tutti. L'associatività è a sinistra.

Per il livello più alto si scrivono semplicemente le produzioni:

$$F \to 0 \mid 1 \mid \cdots \mid 9 \mid (E)$$

Si noti che una espressione tra parentesi è da considerarsi come un operando poiché le parentesi forzano l'interpretazione di una espressione permettendo di eludere le regole di precedenza e associatività se occorre.

Nel livello successivo l'associatività a sinistra è espressa mediante la ricorsione a sinistra nelle produzioni. Una produzione ricorsiva a sinistra provoca infatti l'addossamento a sinistra degli alberi di derivazione che la usano e questa struttura corrisponde proprio alla regola semantica di calcolo con l'associatività a sinistra. Le produzioni sono:

$$T \to T * F \mid T/F \mid F$$

Si noti che un fattore può essere sempre visto come un termine (produzione $T \to F$).

Per il successivo e ultimo livello si ha:

$$E \to E + T \mid E - T \mid T$$

In generale ogni elemento di un livello può essere visto come elemento del livello successivo (in questo caso particolare si guardi la produzione $E \to T$).

Attraverso l'imposizione delle regole di precedenza e di associatività abbiamo ottenuto una grammatica non ambigua per le espressioni aritmetiche. In Figura 3.8 è mostrato un albero di derivazione per l'espressione $3+4+5*7$.

3.4 Top-Down Parsing

In questa sezione introdurremo un metodo per generare parser top-down di una grammatica data. La prima parte introduce delle trasformazioni che

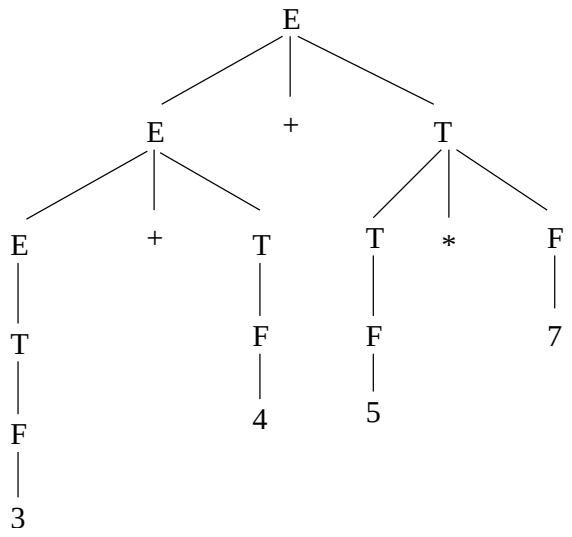


Figura 3.8: Un albero di derivazione per l'espressione $3 + 4 + 5 * 7$.

permettono di modificare una grammatica in modo da renderla più adatta ad un parsing di questo tipo. In seguito introdurremo più precisamente il concetto di parser top-down e daremo un esempio di parser ricorsivo predittivo. L'ultima parte invece tratta la costruzione di parser predittivi non ricorsivi mediante la costruzione di tabelle di parsing appropriate. La sezione si conclude con la definizione delle grammatiche $LL(k)$ e delle loro proprietà.

3.4.1 Eliminazione della ricorsione a sinistra

Una grammatica si dice *ricorsiva a sinistra* se esiste un simbolo non terminale A tale che $A \stackrel{+}{\Rightarrow} A\alpha$ per una qualche stringa α. I metodi di parsing top-down non possono trattare grammatiche ricorsive a sinistra e quindi spesso c'è bisogno di una trasformazione che la elimini.

Si consideri un esempio tipico di produzioni che comportano ricorsione a sinistra:

$$A \to A\alpha \mid \beta$$

Un albero di derivazione che usa queste regole sarà tutto addossato a sinistra e genererà una stringa della forma $\beta\alpha\alpha\cdots\alpha$. Si noti il modo in cui viene generata questa stringa: al primo passo viene generata l'ultima (a destra) α, al secondo passo la penultima e così via fino all'ultimo passo in cui viene generata la β iniziale. Questo procedimento è raffigurato in Figura 3.9.

Un modo alternativo per ottenere la stessa stringa è quello di generare

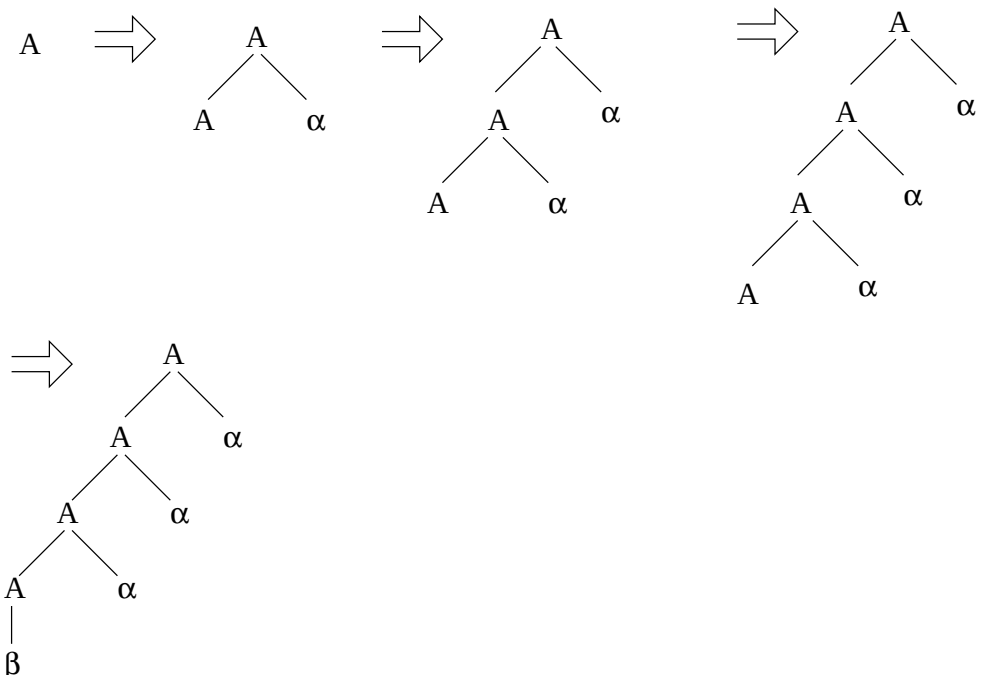


Figura 3.9: Generazione di una stringa con produzioni ricorsive a sinistra.

dapprima la β e poi via via tutte le α procedendo verso destra. Le produzioni che descrivono questo procedimento sono le seguenti:

$$\begin{array}{lcl} A & \rightarrow & \beta A' \\ A' & \rightarrow & \alpha A' \mid \epsilon \end{array}$$

La grammatica scritta in questo modo non è più ricorsiva a sinistra.

Esempio 3.9 *Consideriamo la grammatica per le espressioni aritmetiche che abbiamo scritto in Sezione 3.3 usando le regole di precedenza e associatività (qui come operandi usiamo gli identificatori invece delle cifre):*

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

Seguendo l'osservazione fatta sopra possiamo eliminare la ricorsione a sinistra immediata di E e T ed ottenere la seguente grammatica equivalente:

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \mid \epsilon \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

Il procedimento descritto elimina la ricorsione a sinistra *immediata*, cioè quella che appare direttamente nelle produzioni della grammatica. Nel caso generale il procedimento per eliminarla è il seguente.

Per ogni simbolo A della grammatica si raggruppano le produzioni secondo il seguente schema:

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \cdots \mid \beta_n$$

dove nessun β_i inizia con A e ogni $\alpha_i \neq \epsilon$. Queste produzioni vengono rimpiazzate con le seguenti:

$$\begin{array}{lcl} A & \to & \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' & \to & \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{array}$$

Se non ci sono produzioni di A ricorsive a sinistra allora la definizione di A rimane inalterata.

La grammatica che si ottiene con questa trasformazione è equivalente a quella di partenza e non presenta produzioni con la ricorsione a sinistra immediata.

Passiamo ora ad esaminare la ricorsione a sinistra più in generale e quindi anche quella non immediata (cioè che non si evince dalle produzioni). Si consideri ad esempio la seguente grammatica:

$$\begin{array}{lcl} S & \to & Aa \mid b \\ A & \to & Ac \mid Sd \mid \epsilon \end{array} \tag{3.1}$$

Si ha che A è ricorsivo a sinistra poiché c'è una ricorsione a sinistra immediata nelle sue produzioni. Inoltre anche S è ricorsivo a sinistra poiché si ha la seguente derivazione:

$$S \Rightarrow Aa \Rightarrow Sda$$

Tuttavia S non ha produzioni con la ricorsione a sinistra immediata.

Per eliminare sistematicamente la ricorsione a sinistra, sia essa immediata o no, si può usare l'algoritmo in Figura 3.10. Si noti che questo algoritmo è garantito funzionare sempre per grammatiche prive di cicli (cioè situazioni del tipo $A \stackrel{+}{\Rightarrow} A$) e di ϵ-produzioni (cioè produzioni del tipo $A \to \epsilon$).

Esempio 3.10 *Applichiamo l'algoritmo in Figura 3.10 alla grammatica 3.1. Tecnicamente non è garantito che l'algoritmo funzioni poiché la grammatica contiene ϵ-transizioni, ma in questo caso particolare il risultato è corretto.*

Fissiamo il seguente ordinamento: S, A. Al primo passo ($i = 1$) non succede niente in quanto S non ha produzioni con ricorsione a sinistra immediata.

Input: Una grammatica G senza cicli ed ϵ-transizioni.
Output: Una grammatica equivalente a G non ricorsiva a sinistra.
Metodo: Applica i seguenti passi a G. Si noti che la grammatica risultante potrebbe contenere ϵ-transizioni.

1. Metti i simboli non terminali in un ordine qualsiasi: $A_1, A_2, \ldots, A_n$.

2. **for** $i := 1$ **to** n **do**
   begin
   for $j := 1$ **to** $i - 1$ **do**
   Rimpiazza ogni produzione della forma $A_i \rightarrow A_j\gamma$
   con le produzioni $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$
   se $A_j \rightarrow \delta_1 \mid \cdots \mid \delta_k$ sono le produzioni correnti per A_j;
   Elimina la ricorsione a sinistra immediata da A_i
   end

Figura 3.10: Algoritmo per eliminare la ricorsione a sinistra.

Al secondo passo ($i = 2, j = 1$) bisogna sostituire S nelle produzioni di A con tutte le possibili parti destre (di S):

$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

poi va tolta la ricorsione a sinistra immediata ottenendo la seguente grammatica finale:

$$\begin{array}{lcl} S & \rightarrow & Aa \mid b \\ A & \rightarrow & bdA' \mid A' \\ A' & \rightarrow & cA' \mid adA' \mid \epsilon \end{array}$$

Questa grammatica non presenta ricorsione a sinistra non immediata (né immediata).

3.4.2 Fattorizzazione a sinistra

Questa trasformazione è utile per ottenere grammatiche per le quali è possibile definire parser predittivi. L'idea di base è che quando non è chiaro, guardando l'input, quale, fra due produzioni possibili, usare per espandere un non terminale A, possiamo riscrivere le produzioni per A in modo da rimandare la scelta a quando avremo visto abbastanza input per decidere.

Supponiamo, ad esempio, di avere le seguenti produzioni:

$$\begin{array}{rcl} \mathit{stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \textbf{ else } \mathit{stmt} \\ & | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \end{array}$$

In fase di parsing di una stringa, se vediamo il token **if** non possiamo immediatamente decidere con sicurezza quale delle due produzioni usare per espandere *stmt* correttamente in modo da generare la stringa stessa.

In generale, se $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ sono due produzioni per A e l'input corrente inizia con una stringa non vuota derivabile da α, allora non sappiamo se espandere A con $\alpha\beta_1$ oppure con $\alpha\beta_2$. Però quello che possiamo fare è espandere A in $\alpha A'$ e rimandare la decisione a quando abbiamo visto una stringa generabile da α. A questo punto, infatti, dobbiamo espandere A' in β_1 o β_2, ma ci sono maggiori possibilità che l'input dia maggiori informazioni se β_1 e β_2 generano stringhe che iniziano in maniera differente.

Vediamo un algoritmo generale per fattorizzare a sinistra le produzioni di una grammatica e preservare il linguaggio generato:
Input: Una grammatica G.
Output: Una grammatica equivalente fattorizzata a sinistra.

Metodo: Per ogni non terminale A trova il prefisso più lungo α delle sue alternative (parti destre delle produzioni di A). Se $\alpha \neq \epsilon$ (cioè α è un prefisso non banale comune) allora rimpiazza tutte le produzioni $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma_1 \mid \cdots \mid \gamma_k$ (dove nessun γ_i inizia con α) con:

$$\begin{array}{rcl} A & \rightarrow & \alpha A' \mid \gamma_1 \mid \cdots \mid \gamma_k \\ A' & \rightarrow & \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{array}$$

dove A' è un simbolo non terminale nuovo.

Ripeti questo procedimento fino a quando ci sono almeno due alternative che hanno un prefisso non banale (cioè diverso da ϵ) comune.

Esempio 3.11 *La seguente grammatica è una astrazione di quella dell'if-then-else dove "i" sta per* **if**, *"t" sta per* **then** *ed "e" sta per* **else**:

$$\begin{array}{rcl} S & \rightarrow & iEtS \mid iEtSeS \mid a \\ E & \rightarrow & b \end{array}$$

Un prefisso comune alle due produzioni per S è $iEtS$. Applichiamo la trasformazione ed otteniamo la seguente grammatica fattorizzata a sinistra (si può fare solo un passo):

$$
\begin{array}{lcl}
S & \rightarrow & iEtSS' \mid a \\
S' & \rightarrow & eS \mid \epsilon \\
E & \rightarrow & b
\end{array}
$$

3.4.3 Parser top-down

In questa sezione introduciamo le idee di base per il parsing top-down e mostriamo come costruire un tipo efficiente di parser top-down che non usa backtracking: il parser predittivo. Definiamo inoltre la classe di grammatiche $LL(1)$ cioè le grammatiche per le quali è possibile costruire un parser predittivo.

Il parsing top-down può essere visto come un tentativo di trovare una derivazione leftmost per una certa stringa di input utilizzando le produzioni di una grammatica data. Equivalentemente il processo può essere visto come il tentativo di costruire un albero di derivazione per la stringa di input a partire dalla radice dell'albero espandendo i nodi dell'albero da sinistra a destra.

Illustriamo questo processo con un esempio. Consideriamo la seguente grammatica:

$$
\begin{array}{lcl}
\textit{type} & \rightarrow & \textit{simple} \\
 & \mid & \uparrow \textbf{id} \\
 & \mid & \textbf{array} \; [\; \textit{simple} \;] \; \textbf{of} \; \textit{type} \\
\textit{simple} & \rightarrow & \textbf{integer} \\
 & \mid & \textbf{char} \\
 & \mid & \textbf{num dotdot num}
\end{array}
$$

Il linguaggio generato rappresenta un sottoinsieme delle stringhe che definiscono i tipi in Pascal. Vediamo come un parser top-down si accinge ad analizzare la stringa **array [num dotdot num] of integer**, cioè a trovare un albero di derivazione per essa. Un parser top-down inizia a costruire l'albero dalla radice. Pertanto, all'inizio dell'analisi, il parser si trova in una configurazione in cui l'albero (parziale) che ha costruito è formato solo dalla radice etichettata con il simbolo iniziale della grammatica (in questo caso *type*). Questa configurazione è rappresentata nella parte 1) di Figura 3.11.

In una generica configurazione il parser sceglie il nodo etichettato con un non terminale che sia senza figli e che si trovi più a sinistra nell'albero parziale. Nel nostro esempio, all'inizio, la scelta ricade sulla radice.

Una volta selezionato un nodo il parser si deve basare sulla stringa di input per decidere come espanderlo. Per espansione si intende la selezio-

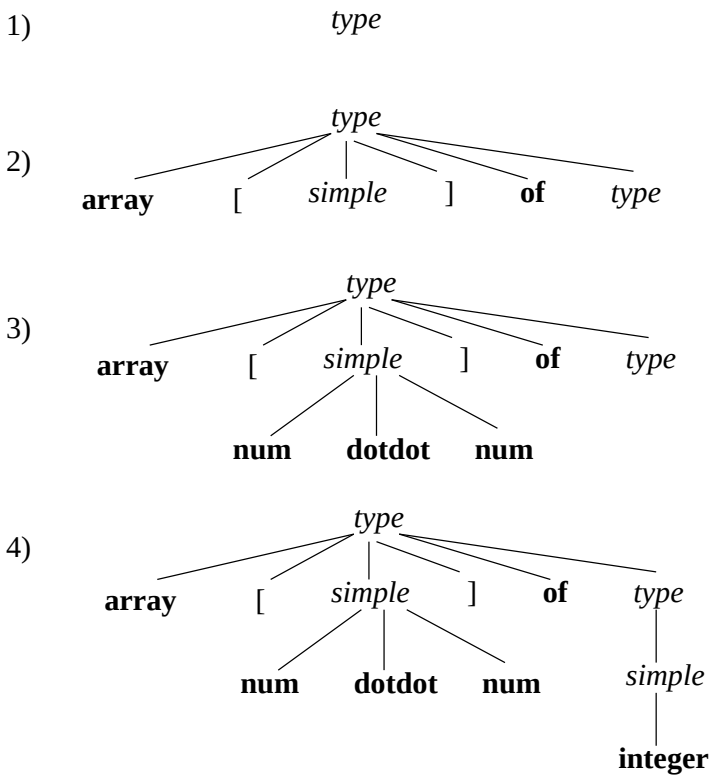


Figura 3.11: Passi nella costruzione top-down del parse tree.

ne di una produzione la cui testa sia il simbolo non terminale che etichetta il nodo scelto e l'aggiunta all'albero di tanti nodi quanti sono i simboli a destra della produzione scelta. Ognuno di questi, nell'ordine, viene aggiunto come figlio del nodo selezionato. Nel nostro esempio il nodo selezionato è etichettato con *type* e per esso è stata scelta la produzione *type* $\rightarrow$ **array** [*simple*] **of** *type*. L'albero parziale risultante dopo l'espansione è quello della parte 2) di Figura 3.11.

Il processo continua fino a quando non si ottiene un albero di derivazione per la stringa data oppure si fallisce.

Nell'esempio, a questo punto, il nodo da selezionare è quello etichettato *simple*. Viene scelta la produzione *simple* $\rightarrow$ **num dotdot num** e, dopo l'espansione, si ha l'albero parziale disegnato nella parte 3) di Figura 3.11. Continuando si arriva ad un albero di derivazione per la stringa data (parte 4) di Figura 3.11).

Si noti che il problema principale, per un parser top-down, è quello di operare una scelta corretta della produzione con cui espandere il nodo selezionato. Se un albero di derivazione esiste per la stringa e la grammatica

non è ambigua, allora ad ogni passo esiste una ed una sola scelta corretta che porta alla costruzione dell'albero di derivazione. Ci si può convincere di questo fatto osservando che la scelta del nodo etichettato con il non terminale più a sinistra corrisponde alla costruzione dell'albero di derivazione seguendo i passi di una derivazione leftmost della stringa stessa.

Operare la scelta corretta ad ogni passo non è sempre possibile. Un parser top-down per una grammatica generica potrebbe dover ritornare sulle sue scelte se si accorge ad un certo punto che la strada che ha intrapreso non porta alla generazione della stringa data. Un parser di questo tipo si chiama "parser con backtracking" poiché l'operazione di ritornare sulle scelte fatte e intraprendere una nuova strada operando una scelta diversa (ove possibile) è in genere denominata "backtracking".

È facile convincersi che un parser che fa backtracking può essere molto inefficiente (nel caso pessimo deve operare tutte le scelte possibili per ogni nodo non terminale). Tuttavia in certi casi può essere conveniente utilizzarne uno.

Un parser molto efficiente, d'altra parte, è uno che ad ogni passo riesce ad "indovinare" la produzione giusta che porterà tutto il processo alla costruzione dell'albero di derivazione per la stringa di input. Un parser di questo tipo si chiama *predittivo*.

Prima di passare ad occuparci più a fondo dei parser predittivi manteniamo ancora un tono generale e guardiamo con maggiore dettaglio il processo di analisi del parser top-down.

La prima cosa da chiarire è che il parser non guarda ad ogni passo tutta la stringa di input per operare la sua scelta. Ciò infatti peserebbe troppo sull'efficienza del processo. Il parser dovrebbe guardare il minimo numero di caratteri della stringa di input che gli servono per poter effettuare una scelta corretta. Vedremo che, in genere, questo numero è 1. I simboli che vengono guardati dal parser sono presi da sinistra a destra dalla stringa di input e si chiamano simboli di *lookahead*.

In Figura 3.12 è raffigurato il processo di costruzione dell'albero di derivazione che abbiamo visto sopra precisando la porzione di input che viene utilizzata per effettuare la scelta. Nella parte a) al di sopra della linea orizzontale è rappresentato il parse tree parziale iniziale. La freccia indica il nodo che il parser sta prendendo in considerazione. Sotto la linea orizzontale è riportata la stringa di input e la freccia qui indica il simbolo di lookahead (in questo esempio ne viene usato uno). Il parser fa la sua scelta esclusivamente in base a questi due simboli: quello che etichetta il nodo corrente e quello di lookahead.

Dopo la prima espansione la situazione è quella rappresentata nella parte b) di Figura 3.12. Il simbolo di lookahead è sempre lo stesso, ma il nodo

corrente è quello più a sinistra. Si noti che in questa trattazione più dettagliata del processo (che poi è anche l'algoritmo di parsing vero e proprio) il nodo corrente è sempre quello più a sinistra e non quello più a sinistra etichettato con un simbolo non terminale. In ogni caso, ovviamente, solo i nodi di questo ultimo tipo saranno oggetto di espansione (e con le modalità viste sopra). In questa situazione si ha che i due simboli che il parser prende in considerazione per le sue decisioni coincidono e sono lo stesso simbolo terminale. In questo caso il parser riconosce un *match* fra l'albero e la stringa di input. L'azione è quella di portare avanti entrambi i puntatori: nell'albero al prossimo nodo e nell'input al prossimo simbolo. La nuova configurazione è rappresentata nella parte c) di Figura 3.12.

Il processo continua trovando un match fra le parentesi quadre aperte fino ad arrivare ad avere **num** come simbolo di lookahead e *type* come simbolo non terminale che etichetta il nodo corrente. In questo caso viene effettuata l'espansione che abbiamo visto sopra decidendo di espandere con la produzione *simple* → **num dotdot num** in base al simbolo corrente di lookahead **num**.

Continuando con i match e le espansioni si arriverà (se il parser è corretto e la stringa fa parte del linguaggio della grammatica) ad avere tutto l'albero di derivazione dopo aver letto tutto l'input.

Parser predittivi

Una grammatica per cui può essere scritto un parser predittivo è fatta in modo tale che, dato un qualsiasi simbolo di lookahead a e un simbolo non terminale da espandere A, una sola tra le alternative α_i di $A \to \alpha_1 \mid \cdots \mid \alpha_k$ può generare una stringa che inizia per a. Se questo è vero, ad ogni passo il parser sa esattamente quale produzione applicare per generare il resto dell'input.

Un esempio molto semplice per capire questo punto è il seguente. In un linguaggio di programmazione tipo il Pascal, quando il parser si aspetta di trovare uno statement, è sufficiente guardare un solo simbolo di input per decidere quale tra le seguenti produzioni applicare:

$$
\begin{array}{lcl}
\textit{stmt} & \to & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\
 & | & \textbf{while } \textit{expr} \textbf{ do } \textit{stmt} \\
 & | & \textbf{begin } \textit{stmt\_list} \textbf{ end}
\end{array}
$$

Evidentemente, se il primo simbolo dell'input è **if** allora il parser applicherà la prima produzione. Se è **while** la seconda e se è **begin** la terza.

Un parser predittivo può essere realizzato in diversi modi. Un modo semplice e naturale è quello di utilizzare delle procedure ricorsive. Vedremo

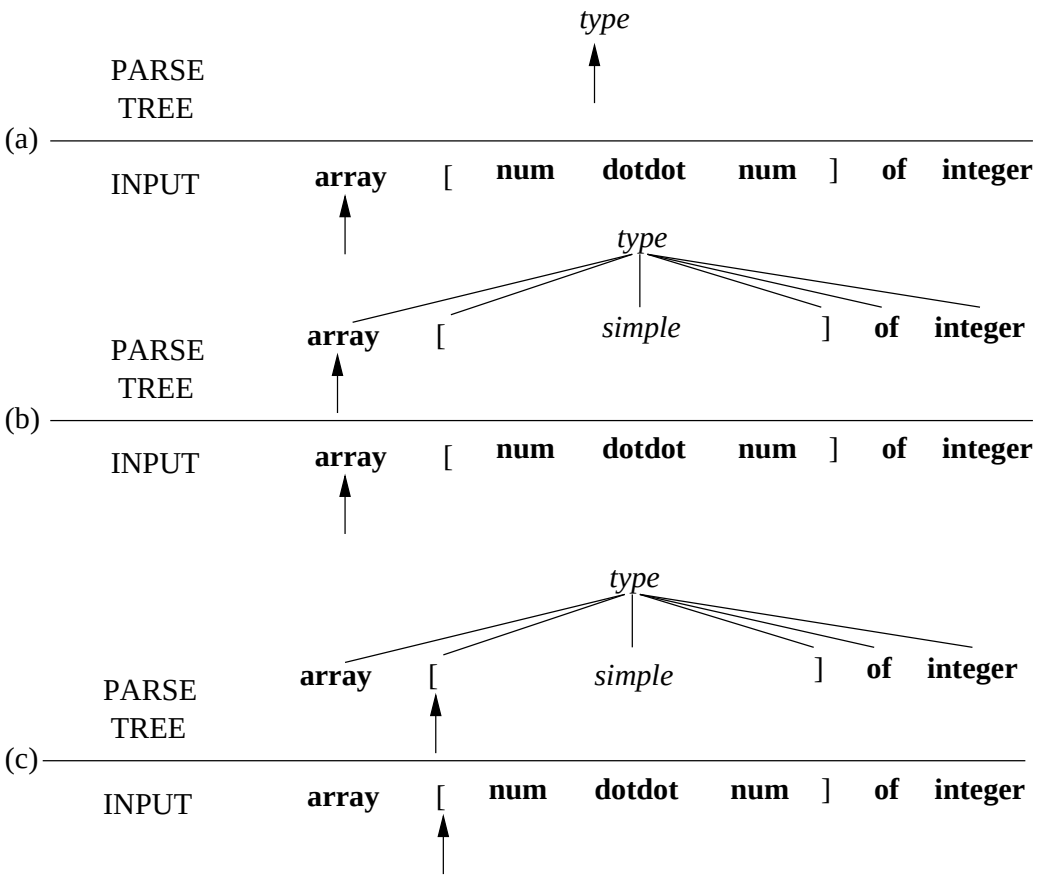


Figura 3.12: Parsing top-down durante la scansione dell'input da sinistra a destra.

poi però che un parser che gestisce esplicitamente uno stack ed è iterativo è più efficiente e maggiormente utilizzato.

Per costruire un parser predittivo utilizzando delle procedure ricorsive basta scrivere una procedura che realizza il match fra simboli terminali (facendo scorrere anche il lookahead) e una procedura per ogni simbolo non terminale. Ognuna di queste ultime è chiamata a riconoscere, in base al simbolo di lookahead, quale produzione applicare per il simbolo non terminale che gestisce e a realizzare le azioni conseguenti (espansione e match).

Le procedure che realizzano un parser predittivo per la grammatica che genera un sottoinsieme dei tipi del Pascal vista sopra sono le seguenti:

```
procedure match(t : token);
begin
        if lookahead = t then
          lookahead := nexttoken
        else error
end;
procedure type;
begin
        if lookahead is in { integer, char, num } then
          simple();
        else if lookahead = '↑' then begin
          match('↑'); match(id)
        end
        else if lookahead = array then begin
          match(array); match('['); simple(); match(']'); match(of); type()
        else error
end;
procedure simple;
begin
        if lookahead = integer then
          match(integer)
        else if lookahead = char then
          match(char)
        else if lookahead = num then begin
          match(num); match(dotdot); match(num)
        end
        else error
end;
```

Prendiamo ad esempio la procedura *type*. Se guardiamo la grammatica ci accorgiamo che se dobbiamo generare una stringa da *type* questa deve ne-

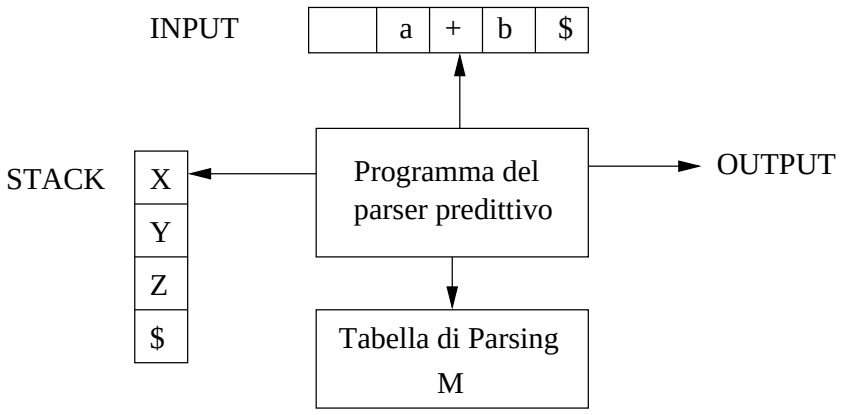


Figura 3.13: Struttura di un parser predittivo non ricorsivo.

cessariamente iniziare con uno dei simboli in { **integer, char, num, array,** ↑}. Inoltre abbiamo un simbolo iniziale diverso in ogni produzione e quindi il parser può essere predittivo. Se ad esempio il simbolo di lookahead è in { **integer, char, num**} allora la produzione da applicare sarà *type* → *simple*. L'espansione è realizzata semplicemente chiamando la procedura *simple* che si occuperà di espandere e fare il match della sottostringa di input che le compete.

Nel caso che il simbolo di lookahead sia **array** allora la produzione da applicare sarà *type* → **array** [*simple*] **of** *type* e le azioni da intraprendere sono dapprima il match dei primi due simboli terminali **array** e [, poi la chiamata della procedura *simple* che si occuperà di fare il match con un tipo semplice, in seguito si deve fare il match con] e **of** e alla fine si richiama ricorsivamente la procedura *type* per andare a riconoscere un altro tipo nella stringa di input rimasta dopo questa espansione.

Per le altre parti delle procedure si hanno considerazioni simili.

Parser predittivi non ricorsivi

È possibile costruire un parser predittivo non ricorsivo gestendo esplicitamente uno stack invece di utilizzarlo implicitamente tramite le chiamate ricorsive.

Abbiamo visto che il problema centrale durante il parser predittivo è quello di determinare quale produzione applicare per espandere un non terminale. In Figura 3.13 è mostrata la struttura di un parser non ricorsivo che decide le sue mosse consultando una tabella di parsing M.

Un parser predittivo guidato da una tabella ha un buffer di input, uno stack, una tabella di parsing e uno stream di output. Il buffer di input

contiene la stringa che deve essere analizzata seguita dal carattere speciale \$.

Il carattere \$ è usato come marcatore destro della fine della stringa. Lo stack contiene una sequenza di simboli della grammatica (terminali o no) con il carattere \$ nella posizione più bassa[2]. La tabella di parsing è un array a due dimensioni in cui ogni entrata $M[A, a]$ (A simbolo non terminale e a simbolo terminale o \$) indica l'azione da eseguire.

Il parser è controllato da un programma che, a grandi linee, si comporta come segue. Il programma considera il simbolo X, il simbolo in testa allo stack, e a, il simbolo corrente di input (o simbolo di lookahead). Questi due simboli determinano l'azione del parser. Ci sono tre possibilità:

1. Se $X = a = \$$ allora il parser si ferma e annuncia la conclusione del parsing con successo.

2. Se $X = a \neq \$$ allora il parser toglie il simbolo a dalla testa della pila (esegue l'operazione *pop* tipica delle pile) e fa avanzare di un simbolo il lookahead (cosicché il nuovo simbolo di lookahead è quello che seguiva a nell'input).

3. Se X è un non terminale allora il parser consulta la tabella all'entrata $M[X, a]$. Il contenuto può essere una X-produzione della grammatica oppure una segnalazione di errore. Se ad esempio $M[X, a] = X \rightarrow UVW$ allora il parser mette nello stack, a posto di X, i simboli WVU (si noti che l'inserimento va fatto a partire dalla fine della parte destra della produzione in modo da avere il primo simbolo, U, in testa). Come output il parser stampa semplicemente la produzione usata (in questo modo abbiamo la sequenza di produzioni da usare nella derivazione leftmost, per la stringa di input, che si sta costruendo). Se $M[X, a] =$ **error** allora il parser chiama una routine di recovery dall'errore.

In Figura 3.14 è formalizzato l'algoritmo eseguito da un generico parser predittivo basato su tabella.

Esempio 3.12 *Si consideri la seguente grammatica:*

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \mid \epsilon \\ F & \rightarrow & (E) \mid \textbf{id} \end{array}$$

[2]Stack significa "pila", cioè la struttura dati che gestisce l'inserimento e il prelevamento di dati in una sequenza con la politica Last-In First-Out.

assegna ad *ip* il puntatore al primo carattere di w\$;
repeat
    sia $X = top$(stack) e sia a il simbolo puntato da *ip*;
    if X is terminale oppure \$ **then**
        if $X = a$ **then**
            pop(stack) e fai avanzare *ip* di un simbolo;
        else *error*
    else /* X non terminale */
        if $M[X, a] = X \to Y_1 Y_2 \cdots Y_k$ **then begin**
            pop(stack);
            push(Y_k, stack);
            push(Y_{k-1}, stack);
            ⋮
            push(Y_1, stack);
            stampa in output la produzione $X \to Y_1 Y_2 \cdots Y_k$
        end
        else *error*
until X = \$ /* Stack vuoto */

Figura 3.14: Programma del parser predittivo.

NON TERM.	**id**	SIMBOLO +	DI *	INPUT (	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**			$F \to (E)$		

Figura 3.15: Tabella di parsing M per la grammatica.

Una tabella di un parser predittivo per questa grammatica è mostrata in Figura 3.15. Le entrate lasciate in bianco si sottointendono contenenti il valore **error**. *Le entrate non bianche indicano una produzione con la quale espandere il non terminale che si trova in testa allo stack. Si noti che ancora non abbiamo visto come costruire questa tabella. Per ora concentriamoci sull'algoritmo del parser utilizzando questa tabella già pronta.*

Con la stringa di input **id+id*id** *il parser predittivo esegue la sequenza di mosse indicate in Figura 3.16. Il simbolo di lookahead ad ogni passo è il primo simbolo della stringa che si trova nella colonna INPUT. Lo stack è rappresentato linearmente e il simbolo di testa ad ogni passo è quello più a destra della stringa riportata nella colonna STACK. Ad ogni passo di espansione nella colonna OUTPUT è riportata la produzione usata.*

Osservando con attenzione le mosse del parser si vede che la sequenza di produzioni nella colonna OUTPUT corrisponde ai passi di una derivazione leftmost della stringa di input.

FIRST e FOLLOW

La costruzione della tabella di un parser predittivo (ma anche dei parser bottom-up che vedremo più avanti) è assistita da due utili funzioni associate alla grammatica G. Queste funzioni, FIRST e FOLLOW, ci permettono di inserire le entrate nella tabella di parsing, quando possibile.

Se α è una stringa qualsiasi di elementi della grammatica allora FIRST(α) è l'insieme dei simboli terminali con cui iniziano le stringhe derivate da α. Se $\alpha \stackrel{*}{\Rightarrow} \epsilon$, allora anche $\epsilon \in$ FIRST(α).

Per un non terminale A l'insieme FOLLOW(A) contiene i simboli terminali a che possono apparire immediatamente alla destra di A in una forma sentenziale, cioè l'insieme degli a tali che esiste una derivazione $S \stackrel{*}{\Rightarrow} \alpha A a \beta$ per qualche α e β. Si noti che, ad un certo punto della derivazione, potrebbero esserci dei simboli non terminali tra A e a che poi vengono riscritti in

STACK	INPUT	OUTPUT
$\$E$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
$\$E'T$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$E \to E'T$
$\$E'T'F$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$T \to FT'$
$\$E'T'\mathbf{id}$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	$F \to \mathbf{id}$
$\$E'T'$	$+\mathbf{id} * \mathbf{id}\$$	
$\$E'$	$+\mathbf{id} * \mathbf{id}\$$	$T' \to \epsilon$
$\$E'T+$	$+\mathbf{id} * \mathbf{id}\$$	$E' \to +TE'$
$\$E'T$	$\mathbf{id} * \mathbf{id}\$$	
$\$E'T'F$	$\mathbf{id} * \mathbf{id}\$$	$T \to FT'$
$\$E'T'\mathbf{id}$	$\mathbf{id} * \mathbf{id}\$$	$F \to \mathbf{id}$
$\$E'T'$	$*\mathbf{id}\$$	
$\$E'T'F*$	$*\mathbf{id}\$$	$T' \to *FT'$
$\$E'T'F$	$\mathbf{id}\$$	
$\$E'T'\mathbf{id}$	$\mathbf{id}\$$	$F \to \mathbf{id}$
$\$E'T'$	$\$$	
$\$E'$	$\$$	$T' \to \epsilon$
$\$$	$\$$	$E' \to \epsilon$

Figura 3.16: Mosse del parser sulla stringa $\mathbf{id} + \mathbf{id} * \mathbf{id}$.

ϵ. Inoltre, se A può essere il simbolo più a destra in una forma sentenziale allora anche il simbolo speciale \$ appartiene a FOLLOW(A).

Per calcolare FIRST(X) per tutti i simboli di grammatica X basta applicare le seguenti regole fino a quando nessun nuovo terminale o ϵ può essere aggiunto ad uno qualsiasi degli insiemi FIRST (iterazione di punto fisso):

1. Se X è terminale allora FIRST(X) è $\{X\}$.

2. Se $X \to \epsilon$ è una produzione della grammatica allora aggiungi ϵ a FIRST(X).

3. Se X è un non terminale e $X \to Y_1 Y_2 \cdots Y_k$ è una produzione della grammatica allora poni a in FIRST(X) se, per qualche i, $a \in$ FIRST(Y_i) e per ogni $j = 1, 2, \ldots, i-1$ si ha $\epsilon \in$ FIRST(Y_j) (cioè $Y_1 Y_2 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$). Se ϵ è in FIRST(Y_j) per ogni $j = 1, 2, \ldots, k$ allora aggiungi ϵ a FIRST(X).

   Ad esempio, ogni cosa che sta in FIRST(Y_1) sta sicuramente anche in FIRST(X). Se poi $\epsilon \in$ FIRST(Y_1) allora bisogna aggiungere a FIRST(X) anche tutti gli elementi di FIRST(Y_2) e così via.

Utilizzando questa procedura come base specifichiamo come si calcola l'insieme FIRST per una qualsiasi stringa $X_1X_2\cdots X_n$ di simboli della grammatica. Dapprima si aggiungono a FIRST$(X_1X_2\cdots X_n)$ tutti i simboli di FIRST(X_1) tranne ϵ. Se $\epsilon \in$ FIRST(X_1) allora si aggiungono anche i simboli non ϵ di FIRST(X_2). Si continua allo stesso modo se ϵ fa parte di FIRST(X_2) e così via. Se per ogni $i = 1, 2, \ldots, n$ si ha che $\epsilon \in$ FIRST(X_i) allora si aggiunge anche ϵ a FIRST$(X_1X_2\cdots X_n)$.

Per calcolare FOLLOW(A) per tutti i non terminali A si applicano le seguenti regole fino a che nessun nuovo simbolo può essere aggiunto ad un qualsiasi insieme FOLLOW (punto fisso):

1. Inserisci \$ in FOLLOW(S) dove S è il simbolo iniziale e \$ è il marcatore di fine input.

2. Se c'è una produzione $A \to \alpha B\beta$ allora, ogni cosa in FIRST(β), eccetto ϵ, va inserito in FOLLOW(B)

3. Se c'è una produzione $A \to \alpha B$ oppure una produzione $A \to \alpha B\beta$ con $\epsilon \in$FIRST(β) (cioè $\beta \stackrel{*}{\Rightarrow} \epsilon$) allora ogni simbolo in FOLLOW(A) è anche in FOLLOW(B).

Esempio 3.13 *Consideriamo ancora una volta la grammatica per le espressioni aritmetiche modificata per eliminare la ricorsione a sinistra:*

$$\begin{array}{lcl} E & \to & TE' \\ E' & \to & +TE' \mid \epsilon \\ T & \to & FT' \\ T' & \to & *FT' \mid \epsilon \\ F & \to & (E) \mid \mathbf{id} \end{array}$$

*Le produzioni $F \to (E)$ e $F \to$ **id** ci dicono che $FIRST(F) = \{(, \mathbf{id}\}$.*
Calcoliamo $FIRST(T')$. Per prima cosa notiamo che la produzione $T' \to \epsilon$ comporta l'aggiunta di ϵ all'insieme. L'altra produzione per T' aggiunge il simbolo $*$. Quindi $FIRST(T') = \{*, \epsilon\}$
L'insieme $FIRST(T)$ è lo stesso di $FIRST(F)$ dato che l'unica produzione per T è $T \to FT'$ e $FIRST(F)$ non contiene ϵ.
Per $FIRST(E')$ valgono le stesse considerazioni fatte per T'. Si ha $FIRST(E') = \{+, \epsilon\}$.
Infine, $FIRST(E)$ è uguale a $FIRST(T)$ cioè $FIRST(F)$.
Calcoliamo ora gli insiemi FOLLOW. Innanzitutto aggiungiamo a $FOLLOW(E)$ il simbolo speciale \$ applicando la regola 1. Per determinare tutto l'insieme $FOLLOW(E)$ guardiamo le produzioni in cui E compare a destra e applichiamo le regole 2. e 3. Si ha che E appare a destra solo nella produzione

$F \to (E)$ e, applicando la regola 2., aggiungiamo il simbolo) a FOLLOW(E) che diventa così l'insieme $\{), \$\}$
Il simbolo E' appare a destra in due produzioni ($E \to TE'$ e $E' \to +TE'$) entrambe le volte come ultimo simbolo. La regola 3. ci dice di aggiungere a FOLLOW(E') sia FOLLOW(E) che FOLLOW(E'). Nel primo caso aggiungiamo) e \$ mentre il secondo caso non aggiunge niente di nuovo all'insieme. Il risultato è che FOLLOW(E') = FOLLOW(E) = $\{), \$\}$.
Il simbolo T appare nelle due produzioni $E' \to +TE'$ e $E \to TE'$. In entrambi i casi la regola 2. ci dice di aggiungere i simboli non ϵ di FIRST(E') (cioè solo il +) a FOLLOW(T). Poi, siccome $\epsilon \in$ FIRST(E'), la regola 3. ci dice di aggiungere a FOLLOW(T) sia FOLLOW(E) che FOLLOW(E'). Pertanto il risultato finale è che FOLLOW(T) = $\{+,), \$\}$.
Per T' valgono le stesse considerazioni di E' e abbiamo FOLLOW(T') = $\{+,), \$\}$.
Infine, per F valgono le stesse considerazioni fatte per T. Otteniamo FOLLOW(F) = FIRST(T')$\cup$ FOLLOW(T)$\cup$ FOLLOW(T') = $\{+, *,), \$\}$.

Costruzione di tabelle per un parser predittivo

L'algoritmo che segue può essere usato per costruire la tabella di parsing. L'idea dell'algoritmo è la seguente. Supponiamo che $A \to \alpha$ sia una produzione tale che $a \in \text{FIRST}(\alpha)$. Allora il parser espanderà A con questa produzione ogni volta che il simbolo di lookahead sarà a. L'unica complicazione si verifica quando $\alpha \stackrel{*}{\Rightarrow} \epsilon$ o $\alpha = \epsilon$. In questi casi dobbiamo sempre espandere A con α, ma solo se il simbolo di lookahead appartiene a FOLLOW(A).

Input: Una grammatica G
Output: Tabella di parsing M

Metodo:

1. Per ogni produzione $A \to \alpha$ di G esegui i passi 2 e 3.

2. Per ogni terminale $a \in \text{FIRST}(\alpha)$ poni $M[A, a] := A \to \alpha$.

3. Se $\epsilon \in \text{FIRST}(\alpha)$ allora poni $M[A, b] := A \to \alpha$ per ogni terminale b in FOLLOW(A). Se $\epsilon \in \text{FIRST}(\alpha)$ e $\$ \in \text{FOLLOW}(A)$ allora poni $M[A, \$] := A \to \alpha$.

4. Poni ogni altra entrata indefinita a *error*.

NON TERM.	a	SIMBOLO b	DI e	INPUT i	t	$
S	$S \to a$			$S \to iCtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

Figura 3.17: Tabella di parsing per la grammatica.

Esempio 3.14 *Applicando l'algoritmo appena visto alla solita grammatica per le espressioni aritmetiche otteniamo la tabella di parsing di Figura 3.15.*

Grammatiche $LL(1)$

L'algoritmo di costruzione della tabella può essere applicato ad una qualsiasi grammatica G. Tuttavia, per alcune grammatiche, M può avere alcune entrate che sono multidefinite (cioè c'è più di un valore per la stessa casella $M[A, a]$). Ad esempio, se G è ricorsiva a sinistra oppure è ambigua allora M avrà almeno una entrata multidefinita.

Esempio 3.15 *Consideriamo ancora la grammatica che astrae il comando if-then-else:*

$$\begin{array}{lcl} S & \to & iEtSS' \mid a \\ S' & \to & eS \mid \epsilon \\ E & \to & b \end{array}$$

*La tabella di parsing per questa grammatica è mostrata in Figura 3.17. L'entrata per $M[S', e]$ contiene sia $S' \to eS$ che $S' \to \epsilon$ poiché $FOLLOW(S') = \{e, \$\}$. La grammatica è ambigua e l'ambiguità si manifesta nella scelta di quale produzione utilizzare quando viene incontrato il simbolo e (***else***). Possiamo risolvere questa ambiguità se preferiamo $S' \to eS$. Questa scelta corrisponde ad associare ogni* **else** *con il* **then** *precedente più vicino. Si noti che la scelta $S' \to \epsilon$ impedirebbe sempre ad e di essere posto nello stack o rimosso dall'input e quindi è sicuramente sbagliata.*

Definizione 3.16 (Grammatica $LL(1)$) *Una grammatica si dice $LL(1)$ se esiste una tabella per il parsing predittivo che non ha entrate multidefinite.*

La prima "L" di $LL(1)$ indica che l'input è scandito da sinistra a destra ("L" come Left, in inglese). La seconda L indica che il parsing produce una derivazione leftmost e 1 è il numero di simboli di lookahead usati.

È possibile dimostrare che l'algoritmo visto produce, per ogni grammatica G che sia $LL(1)$, una tabella di parsing che analizza tutte e sole le parole di $L(G)$.

Vediamo adesso un'altra caratterizzazione delle grammatiche $LL(1)$ e la definizione di grammatica $LL(k)$.

Definizione 3.17 (Guide) *Data una grammatica G e una sua produzione $A \to \delta$, la GUIDA della stessa è il seguente insieme:*

$$\text{GUIDA}(A \to \delta) = \{a \mid a \in \text{FIRST}(\delta) \vee (\delta \stackrel{*}{\Rightarrow} \epsilon \wedge a \in \text{FOLLOW}(A))\}$$

Proposizione 3.1 *Una grammatica G è $LL(1)$ se per ogni coppia di produzioni $A \to \delta_i$, $A \to \delta_j$ si ha che*

$$\text{GUIDA}(A \to \delta_i) \cap \text{GUIDA}(A \to \delta_j) = \{\}$$

Questo risultato segue direttamente dall'algoritmo che abbiamo visto per la costruzione della tabella di parsing.

Affrontiamo ora la questione che riguarda cosa si dovrebbe fare quando la tabella di parsing che si è costruita contiene entrate multidefinite. Quello che si può fare sempre è cercare di cambiare la grammatica con trasformazioni che conservano il linguaggio generato e che portino ad una grammatica per cui la tabella non è più multidefinita.

La cosa tipica che si fa è eliminare la ricorsione a sinistra e fattorizzare a sinistra il più possibile (vedi Sezioni 3.4.1 e 3.4.2).

Sfortunatamente non per tutte le grammatiche l'applicazione di queste trasformazioni porta ad una grammatica $LL(1)$. Un esempio è la grammatica per l'if-then-else ripresa nell'Esempio 3.15: per il linguaggio generato da questa grammatica non esiste nessuna grammatica $LL(1)$ che lo genera. In questi casi si dice che il linguaggio non è $LL(1)$ per indicare che non esiste una grammatica $LL(1)$ per lo stesso. Analogamente si dice che un linguaggio è $LL(1)$ se esiste una grammatica $LL(1)$ che lo genera.

In generale non ci sono regole universali con le quali le tabelle multidefinite possano essere poste ad un singolo valore senza modificare il linguaggio generato dalla grammatica di partenza.

La più grande difficoltà nell'usare il parsing predittivo è nello scrivere una grammatica per il linguaggio sorgente tale che una tabella di parsing possa essere costruita utilizzando l'algoritmo visto. Si noti inoltre che qualora si decida di utilizzare l'eliminazione della ricorsione a sinistra e la fattorizzazione a sinistra si ottengono grammatiche difficili da leggere e da usare a scopi traduttivi.

Una scappatoia usata spesso a questi problemi è quella di usare parser predittivi per i costrutti di controllo dei linguaggi di programmazione (per i quali è semplice definire una grammatica $LL(1)$) e utilizzare altre tecniche per i restanti costrutti. In ogni caso, se è disponibile un analizzatore LR (che vedremo nelle sezioni successive), è meglio usare direttamente questo.

Vediamo ora una definizione delle grammatiche $LL(k)$

Definizione 3.18 (Grammatiche $LL(k)$) *Una grammatica G è $LL(k)$ ($k \geq 1$) se l'esistenza di due derivazioni*

$$S \overset{*}{\Rightarrow}_{\text{lm}} xA\alpha \Rightarrow_{\text{lm}} x\beta\alpha \overset{*}{\Rightarrow}_{\text{lm}} xy$$

$$S \overset{*}{\Rightarrow}_{\text{lm}} xA\alpha \Rightarrow_{\text{lm}} x\gamma\alpha \overset{*}{\Rightarrow}_{\text{lm}} xz$$

con $y/k = z/k$ (cioè i primi k simboli di y sono uguali ai primi k simboli di z)
implica $\beta = \gamma$.

In altre parole, se non ci sono due produzioni diverse che derivano stringhe di terminali i cui primi k simboli sono uguali.

Questa definizione può essere usata per dimostrare abbastanza facilmente che una grammatica non è $LL(k)$ per un certo k. Basta trovare due derivazioni della forma indicata dalla definizione e far vedere che si sono usate, per A, due produzioni diverse (cioè $\beta \neq \gamma$).

Si ha che la classe dei linguaggi generati dalle grammatiche $LL(k)$ è un sottoinsieme proprio della classe dei linguaggi generati da dalle grammatiche $LL(k')$ con $k' > k$.

Quindi, in pratica, se si dimostra che una grammatica è $LL(1)$ allora si ha che la stessa grammatica è anche $LL(k)$ per ogni $k \geq 1$.

3.5 Bottom-Up Parsing

In questa sezione introduciamo una tecnica di analisi sintattica bottom-up che prende il nome di *shift-reduce parsing* o analisi impila-riduci. Un metodo generale per questo tipo di parsing è il cosiddetto LR parsing che è usato in molti generatori automatici di analizzatori sintattici.

L'analisi appila-riduci si occupa di costruire il parse tree della stringa di token che gli viene data in input. La costruzione parte dalle foglie dell'albero e continua con l'accorpamento di sottoalberi fino ad arrivare alla radice del parse tree cercato.