Dispense del corso di Linguaggi di Programmazione e Compilatori

Luca Tesei

A. A. 2003/2004

Indice

Introduzione

Queste note intendono coprire gli argomenti trattati nel corso di Linguaggi di Programmazione e Compilatori del Corso di Laurea in Informatica dell'Università di Camerino.

Il corso tratta, in generale, della struttura dei linguaggi di programmazione di alto livello e delle tecniche sviluppate per implementarli. Per implementazione si intende principalmente lo sviluppo di un compilatore o di un interprete per un linguaggio in modo da rendere effettivamente eseguibile il programma compilato o interpretato su una certa macchina.

La prima parte del corso introdurrà i diversi paradigmi di programmazione e i concetti di "macchina astratta", "implementazione compilativa" e "implementazione interpretativa"[1].

In seguito ci concentreremo soprattutto su alcune fasi di progetto e scrittura di un compilatore. Nel Capitolo 1 introdurremo il contesto di un compilatore e un modello generale di compilatore in cui le operazioni sono divise in fasi distinte.

È importante far notare subito che i principi e le tecniche impiegate in questo campo spaziano su diverse discipline informatiche e costituiscono un nucleo che sarà usato molte volte durante la carriera di un informatico. Questo significa che, anche se non è molto probabile trovarsi a dover progettare e/o implementare un compilatore vero e proprio, gli algoritmi e i concetti che si studieranno in questo corso si riveleranno molto utili in diverse applicazioni.

Le discipline che la scrittura di un compilatore tocca sono le seguenti:

- Linguaggi di Programmazione: sono gli oggetti che trattiamo. È necessario conoscerli e sapere come funzionano anche in maniera formale. In particolare è necessario conoscere i diversi formalismi per specificare la sintassi e la semantica degli stessi.

- Architettura degli elaboratori/Sistemi operativi: il risultato finale di una compilazione è un programma in un linguaggio che può essere ca-

[1] Questa parte coperta dai lucidi powerpoint che si possono scaricare alla pagina `http://www.di.unipi.it/tesei/unicam/LPC20032004.html`

ricato ed eseguito su una certa macchina fisica. La generazione del codice, la sua struttura e l'ottimizzazione dello stesso dipendono da come funziona la macchina fisica. Pertanto, bisogna conoscere in dettaglio sia l'architettura sia il sistema operativo di quest'ultima.

- Teoria dei linguaggi formali: per poter sviluppare tecniche algoritmiche standard ci si basa su una specifica *formale* dei linguaggi di programmazione. In questo modo si ha sia una rappresentazione precisa e matematica degli stessi sia una base su cui sviluppare e provare la correttezza degli algoritmi di traduzione. In particolare, si usano gli automi a stati finiti e le grammatiche libere dal contesto per la specifica di varie componenti dei linguaggi di programmazione.

- Algoritmi: l'analisi lessicale, il *parsing*, la traduzione, la verifica di proprietà statiche e l'ottimizzazione sono implementate da algoritmi studiati fin dagli anni '60. A partire da questi algoritmi sono stati sviluppati anche dei tool che si occupano di generare automaticamente il codice di alcune parti di un compilatore date le specifiche formali.

- Ingegneria del software: il progetto di un compilatore è un problema complesso e richiede l'uso di tecniche di ingegneria del software. Vedremo, ad esempio, che il processo di compilazione sarà logicamente diviso in fasi che poi, nell'implementazione vera e propria, possono essere accorpate.

Nel Capitolo 2 studieremo le tecniche per la prima e più semplice fase di compilazione, cioè l'analisi lessicale. Questa fase consiste nel leggere il codice sorgente di un programma dall'inizio alla fine e produrre, da questo stream di caratteri, uno stream di *token*. Un token è una qualsiasi unità di base del linguaggio che stiamo considerando. Esempi di token sono gli identificatori, le parole riservate, gli operatori aritmetici eccetera.

Verranno introdotti formalmente i linguaggi regolari e la teoria degli automi a stati finiti come loro riconoscitori. Introdurremo le espressioni regolari come formalismo algebrico per la specifica dei linguaggi regolari ed il teorema fondamentale di Kleene. Dal punto di vista algoritmico studieremo l'algoritmo di costruzione dei sottoinsiemi per eliminare il non determinismo e un algoritmo di minimizzazione per minimizzare il numero di stati di un automa. Vedremo come può essere realizzato, utilizzando gli algoritmi studiati, un generatore automatico di analizzatori lessicali dato un file di descrizione dei token (come esempio vedremo il tool Lex).

Nel Capitolo 3 introdurremo formalmente le grammatiche libere dal contesto e studieremo come realizzare un analizzatore che, preso in ingresso uno

stream di token, restituisce un albero di derivazione astratto (*parse tree*) che rappresenta la struttura, secondo la grammatica del linguaggio, dello stream di token in ingresso. Vedremo che la realizzazione di un analizzatore sintattico o *parser* è più complicata di quella di un analizzatore lessicale.

Resta da parlare della gestione degli errori che, in un compilatore, è una componente difficile ma essenziale. In ogni fase della compilazione può verificarsi un errore. Nel trattare ogni fase cercheremo di dare dei cenni su come gestire la situazione, sul *recovering* e sulla continuazione della compilazione per individuare eventuali altri errori.

Capitolo 1

Contesto e fasi di un compilatore

Come abbiamo visto nella parte dedicata alla definizione di compilazione e traduzione, una implementazione compilativa di un linguaggio di programmazione è realizzata tramite un programma (il compilatore) che, preso un altro programma scritto nel linguaggio *sorgente*, restituisce un programma *funzionalmente equivalente* a quello dato, ma scritto in un linguaggio *target*, che è il linguaggio della macchina intermedia scelta per l'implementazione. Il compilatore in sé può essere scritto in qualsiasi linguaggio e girare su qualsiasi macchina astratta.

La conoscenza di tecniche efficaci per la scrittura di un programma compilatore è ad oggi molto vasta e strutturata. Non era così per i primi programmatori che alla fine degli anni '50 si accingevano a scriverne uno. Si pensi ad esempio che la scrittura del primo compilatore Fortran richiese 18 anni di lavoro da parte di uno staff. Sin da allora sono state individuate molte tecniche sistematiche per la scrittura del codice delle fasi più importanti di un compilatore, oltre ad essere sorti linguaggi di programmazione adatti (ad esempio il linguaggio C) per l'implementazione, ambienti di programmazione e tool software (generatori automatici di codice a partire dalle specifiche di varie parti del linguaggio sorgente).

Il processo di compilazione è concettualmente diviso in due parti: l'analisi e la sintesi. La parte di analisi si occupa di dare una struttura al codice sorgente e di creare una sua rappresentazione intermedia. La parte di sintesi genera il codice nel linguaggio target a partire dalla rappresentazione intermedia.

Durante la fase di analisi le operazioni indicate nel programma sorgente vengono riconosciute e raggruppate in una struttura ad albero. Spesso viene usato il cosiddetto *albero sintattico* (syntax tree), cioè un albero in cui

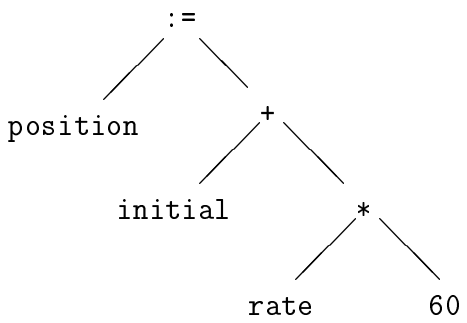


Figura 1.1: Albero sintattico per `position := initial + rate * 60`

ogni nodo interno rappresenta un operatore i cui operandi sono reperibili nei figli. Ad esempio, in Figura 1.1 è raffigurato un albero sintattico per l'assegnamento Pascal `position := initial + rate * 60`

1.1 Contesto di un compilatore

Per poter generare un programma target eseguibile sulla macchina ospite, altri tipi di programmi vengono in aiuto del compilatore. Ciò per consentirgli di concentrarsi solo sulla traduzione.

In molti casi succede che il programma sorgente sia diviso in moduli che si trovano su file diversi. Il compito di recuperare tutti i file e di creare un file unico sorgente è spesso affidato ad un *preprocessore*. Esso, in genere, si occupa anche di espandere le macro che possono essere presenti nel programma sorgente originale. Un esempio è il preprocessore per il compilatore del linguaggio C. Esso si occupa di rimpiazzare le direttive `#include` con il testo contenuto nei file specificati. Inoltre rimpiazza nel codice le definizioni di macro definite dalle direttive `#define`.

L'output della traduzione spesso necessita di altre trasformazioni prima di poter essere effettivamente eseguito sulla macchina ospite. Ad esempio non è raro trovare dei compilatori che generano come codice target un programma scritto nel codice assembly della macchina ospite. Questo codice deve essere assemblato tramite un *assemblatore* per diventare codice macchina rilocabile.

Inoltre abbiamo visto che la macchina intermedia utilizzata nell'approccio compilativo viene simulata sulla macchina ospite e spesso questa simulazione viene implementata tramite un supporto a tempo di esecuzione. Riprendendo l'esempio del linguaggio C, nei sistemi unix il supporto a tempo di esecuzione per il C è formato da librerie di funzioni rilocabili (che esistono in unica copia in directory apposite del file system) che vengono caricate solo quando il programma va in esecuzione. Il *caricatore* si preoccupa di reperire questo

codice rilocabile condiviso e di rilocare tutto il codice così ottenuto. Il risultato è un codice macchina assoluto che può essere eseguito sulla macchina ospite. In Figura 1.2 è raffigurato questo esempio. La rilocazione del codice consiste nel convertire gli indirizzi relativi all'interno dello spazio logico del programma in indirizzi assoluti (a seconda dell'effettiva zona della memoria in cui il programma viene caricato). In molti casi, a seconda della macchina hardware e del sistema operativo, la rilocazione viene fatta a livello hardware o firmware.

1.2 Analisi

In questa sezione introdurremo in maniera non dettagliata le fasi che compongono la parte di analisi. Per ognuna di queste fasi, approfondiremo le tecniche e gli algoritmi che le sono propri nei capitoli successivi.

L'analisi viene divisa in tre fasi:

1. Analisi lineare: lo stream di caratteri che costituiscono il programma sorgente è letto da sinistra a destra e raggruppato in *token* che sono sequenze di caratteri che hanno un significato comune.

2. Analisi gerarchica: i token vengono raggruppati gerarchicamente in strutture annidate (alberi) che specificano la struttura delle frasi del linguaggio.

3. Analisi semantica: attraverso vari controlli si cerca di appurare se le varie parti del programma sono consistenti una con l'altra, a seconda del loro significato.

1.2.1 Analisi lessicale

In un compilatore, l'analisi lineare è chiamata analisi lessicale o *scanning*. Durante questa analisi una linea di caratteri come la seguente

```
position := initial + rate * 60
```

viene, in genere, raggruppata nei seguenti token:

1. L'identificatore `position`

2. Il simbolo di assegnamento: `:=`

3. L'identificatore `initial`

4. Il segno dell'addizione

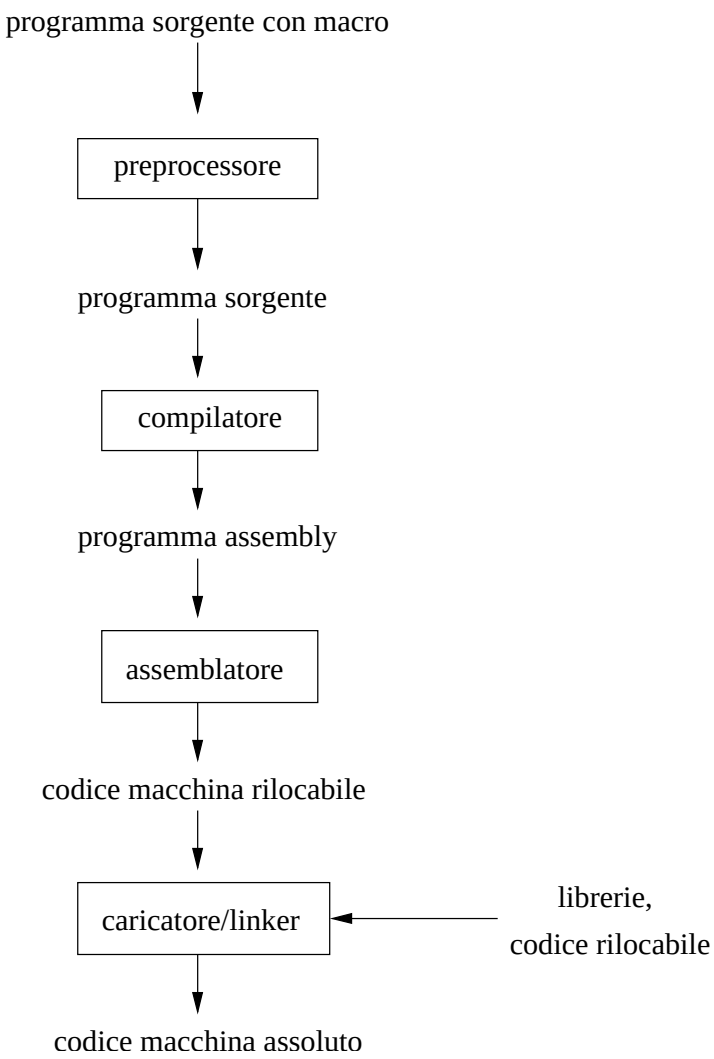


Figura 1.2: Esempio di un contesto di un compilatore.

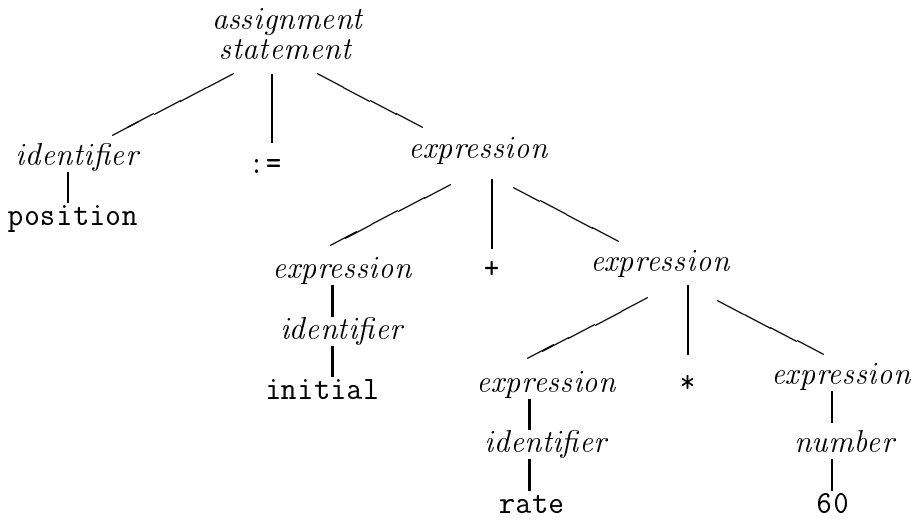


Figura 1.3: Parse tree per `position := initial + rate * 60`

5. L'identificatore `rate`
6. Il segno della moltiplicazione
7. Il numero `60`

I caratteri di spazio (ma anche tabulazioni, commenti e caratteri che indicano la fine di una linea) che separano i token vengono, normalmente, eliminati durante l'analisi lessicale.

1.2.2 Analisi sintattica

L'analisi gerarchica viene chiamata *parsing* o analisi sintattica. Il suo compito è di raggruppare i token in frasi grammaticali che poi saranno usate per sintetizzare l'output. Generalmente le frasi grammaticali del programma sorgente sono rappresentate con un albero di derivazione (o *parse tree*) di una grammatica libera dal contesto. Vedremo questo formalismo più in dettaglio nel Capitolo 3. In Figura 1.3 è raffigurato un parse tree per i token dell'esempio della sezione precedente.

La divisione fra analisi lessicale e sintattica è una scelta del progettista del compilatore e non ci sono direttive prestabilite. In genere la scelta ricade sulla soluzione che semplifica di più l'analisi sintattica. Un fattore determinante per la divisione è l'intrinseca ricorsività di un costrutto. I costrutti lessicali non richiedono la ricorsione, mentre i costrutti sintattici spesso si. E infatti le grammatiche libere dal contesto sono una formalizzazione di regole ricorsive che possono essere usate per guidare l'analisi sintattica.

Facciamo un esempio: la ricorsione non è necessaria per riconoscere gli identificatori che, tipicamente, sono stringhe che cominciano con una lettera e continuano con cifre o lettere. Per riconoscere un identificatore, quindi, basta vedere una lettera e continuare a leggere l'input fino a che non si incontra un carattere che non è né una cifra né una lettera. Quando questo accade basta raggruppare tutti i caratteri letti, meno l'ultimo, e considerarli come un token identificatore. Questi caratteri raggruppati vengono poi inseriti nella *tabella dei simboli* (che è una struttura dati molto importante usata in tutte le fasi della compilazione) e rimossi dall'input in modo che si possa continuare la ricerca del token successivo.

D'altra parte, questo tipo di scansione lineare non è abbastanza potente per poter analizzare espressioni o *statement*. Ad esempio non si possono riconoscere stringhe in cui le parentesi aperte e chiuse sono in egual numero, oppure non si possono associare nella maniera giusta i `begin` e gli `end` dei costrutti senza dare una struttura gerarchica o annidata all'input.

Il parse tree in Figura 1.3 descrive la struttura sintattica dell'input. Una rappresentazione più concisa e per questo più usata come rappresentazione interna è quella dell'albero sintattico, di cui abbiamo un esempio in Figura 1.1 (questo albero è il corrispondente più conciso dell'albero di derivazione di Figura 1.3). Approfondiremo l'uso di queste strutture dati nel Capitolo 3.

1.2.3 Analisi semantica

La fase di analisi semantica controlla se ci sono errori semantici nel programma sorgente e acquisisce l'informazione sui tipi che verrà usata nella fase successiva di generazione del codice. La rappresentazione gerarchica generata dall'analisi sintattica viene usata per identificare gli operatori e gli operandi delle espressioni e degli statement.

Una componente importante di questa fase è il *type checking* in cui il compilatore controlla che ogni operatore viene applicato ad operandi consentiti dalla specifica del linguaggio. Un esempio: la definizione di molti linguaggi di programmazione richiede che sia riportato un errore se un numero reale è utilizzato per indicizzare un array.

Tuttavia alcuni "abusi" sull'uso degli operandi potrebbero essere permessi, come ad esempio l'applicazione di un operatore aritmetico binario ad un numero intero e ad un reale. In questo caso il compilatore può dover convertire l'intero in reale per ottenere un risultato coerente. Supponiamo, infatti, che il type checking ha rivelato che l'identificatore `rate` nella nostra stringa di esempio abbia tipo `real` e che il numero `60` sia considerato di per se stesso come un intero. Generalmente in questi casi il compilatore deve convertire il numero intero in numero reale poiché la rappresentazione interna degli interi

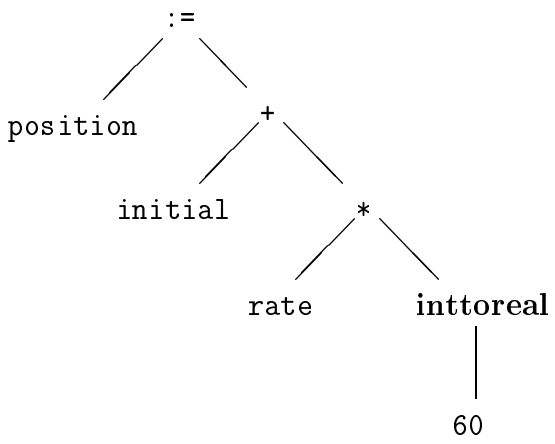


Figura 1.4: Albero sintattico dopo l'analisi semantica.

e dei reali in genere è diversa. Questa operazione viene segnalata inserendo un ulteriore nodo interno nell'albero sintattico e sarà effettuata nella fase di traduzione (l'albero sintattico diventa come in Figura 1.4).

1.3 Le fasi di un compilatore

Concettualmente un compilatore opera in diverse fasi ognuna delle quali trasforma il programma sorgente da una rappresentazione ad un'altra. Una tipica decomposizione in fasi è mostrata in Figura 1.5. Nella reale implementazione, poi, le fasi vengono spesso accorpate e le rappresentazioni intermedie che ci sono tra le fasi raggruppate non vengono costruite esplicitamente.

Dalla figura si notano due componenti, il gestore della tabella dei simboli e il gestore degli errori, che interagiscono con tutte le fasi. Le chiameremo informalmente "fasi", ma non lo sono propriamente.

1.3.1 Gestione della tabella dei simboli

Una funzione essenziale di un compilatore è quella di memorizzare gli identificatori che vengono usati nel programma sorgente insieme con i relativi valori di diversi attributi. Questi attributi possono essere, ad esempio, lo spazio in byte allocato per l'identificatore, il suo tipo, il suo *scope* (la parte di programma sorgente in cui l'identificatore ha significato). Ancora, se l'identificatore è il nome di una procedura, altri tipi di attributi sono il numero e i tipi di parametri che essa richiede, la modalità di passaggio di questi parametri (per valore, per riferimento, per nome), il tipo del valore ritornato, se c'è.

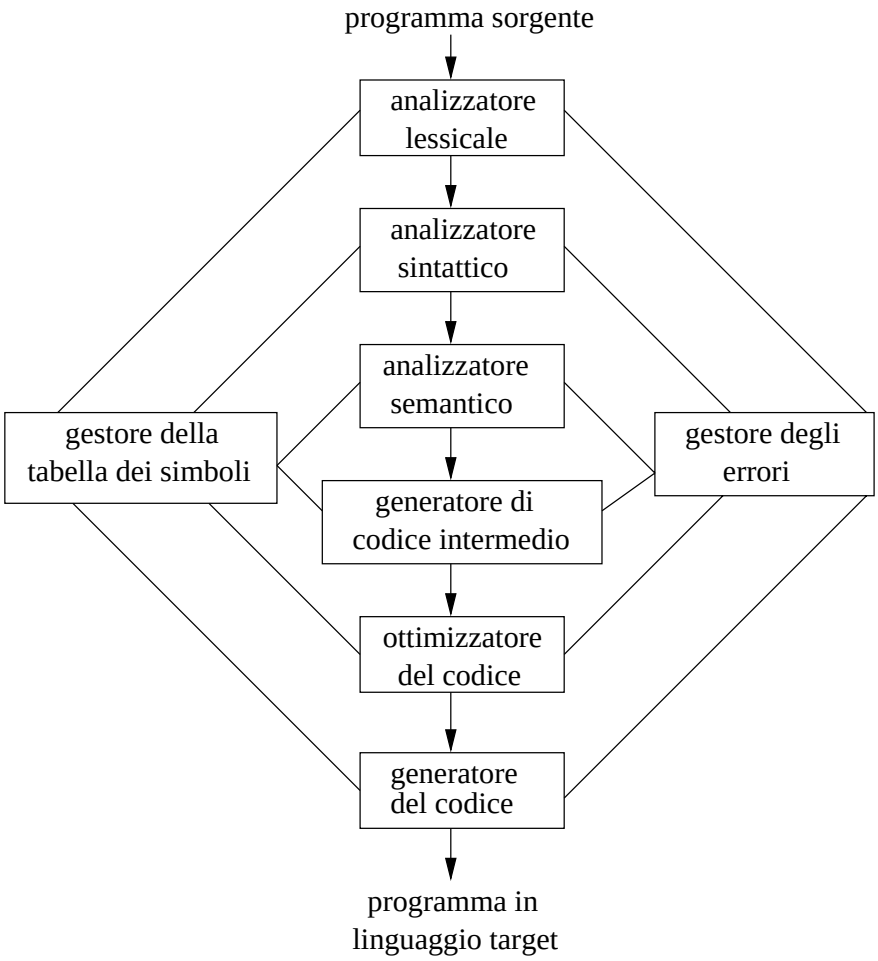


Figura 1.5: Fasi di un compilatore.

La tabella dei simboli è una struttura dati che contiene un record per ogni identificatore e in cui i vari campi del record sono gli attributi. Questa struttura dati deve consentire di trovare efficientemente il record di un identificatore che stiamo cercando e di leggere/scrivere nuovi valori in maniera efficiente.

Quando un nuovo identificatore viene trovato nel codice sorgente durante la fase di analisi lessicale esso viene inserito nella tabella dei simboli dall'analizzatore. Tuttavia i valori degli attributi non possono essere trovati tutti durante questa fase. Ad esempio, nella dichiarazione Pascal:

```
var position, initial, rate :  real;
```

il tipo `real` non è ancora conosciuto quando gli identificatori `position`, `initial`, `rate` sono letti dall'analizzatore lessicale.

Le fasi rimanenti inseriscono informazioni nella tabella dei simboli e/o le usano in vari modi. Ad esempio, durante le fasi di analisi semantica e di generazione del codice intermedio c'è bisogno di sapere il tipo degli identificatori per controllare che siano usati propriamente e per generare le corrette operazioni su di essi. La fase di generazione del codice inoltre aggiunge alla tabella dei simboli anche informazioni dettagliate sullo spazio di memoria allocato per ogni identificatore.

1.3.2 Rilevazione e presentazione degli errori

Una parte importante del processo di compilazione, oltre alla traduzione vera e propria, è il riconoscimento e la segnalazione di errori nel programma sorgente. Ogni fase può incontrare errori e, dopo averne rilevato uno, essa deve in qualche modo trattarlo per fare in modo che la compilazione continui e altri eventuali errori possano essere rilevati. Questo perchè un compilatore che si fermi una volta che ha trovato il primo errore non è così di aiuto come potrebbe essere.

Le fasi di analisi sintattica e semantica sono quelle che devono fronteggiare la frazione maggiore di errori che un compilatore è in grado di rilevare. L'analizzatore lessicale può rilevare un errore quando i caratteri rimanenti dell'input non formano nessun token del linguaggio. Quando lo stream di token vìola le regole strutturali del linguaggio (la sintassi), il parser rileva un errore. Durante l'analisi semantica possono essere riconosciuti i costrutti che, pur essendo sintatticamente corretti, implicano operazioni non consentite fra gli operandi in gioco. Si pensi ad esempio all'addizione fra una variabile di un tipo primitivo ed un array[1]. Quando parleremo dettagliatamente di ogni

[1]Nulla toglie ovviamente che il linguaggio possa prevedere questo tipo di operazione, ma in ogni caso essa non deve essere trattata come una normale addizione, bensì dovrà essere

fase faremo anche degli accenni a come poter gestire gli errori che tipicamente possono sorgere.

1.3.3 Le fasi di analisi

Abbiamo già parlato delle prime tre fasi. In questa sezione però precisiamo alcune cose sulla rappresentazione interna che esse generano.

La parte di analisi lessicale, abbiamo visto, raggruppa i caratteri del programma sorgente in token, che sono sequenze di caratteri che hanno un significato comune (una categoria lessicale), come ad esempio gli identificatori, le parole chiave (`if, then, while`, ecc.) i segni di punteggiatura, gli operatori multicarattere (ad esempio `:=`). La sequenza di caratteri che forma un certo token è chiamata *lexeme* del token in questione. Oltre all'informazione sul token stesso, alcune categorie lessicali richiedono che un'altra informazione sia ad esso associata: il "valore lessicale".

Ad esempio, quando viene trovato un identificatore come `rate`, l'analizzatore lessicale genera un token, chiamiamolo **id**, e inserisce il lexeme `rate` nella tabella dei simboli (se non c'è già). Il valore lessicale associato a questa occorrenza del token **id** sarà un puntatore all'entrata della tabella dei simboli che contiene il lexeme (e, abbiamo visto, anche altri attributi).

In questa sezione useremo $\mathbf{id}_1$, $\mathbf{id}_2$ ed $\mathbf{id}_3$ in luogo di `position`, `initial` e `rate` proprio per enfatizzare il fatto che la rappresentazione interna di un identificatore è diversa dal lexeme. Quindi la nostra stringa di esempio, dopo l'analisi lessicale, ha la seguente rappresentazione:

$\mathbf{id}_1$ `:=` $\mathbf{id}_2$ `+` $\mathbf{id}_3$ `* 60`

Avremmo dovuto definire anche la rappresentazione degli altri token, ma posticipiamo questo discorso al Capitolo 2. La rappresentazione del codice che viene fuori dalle due fasi successive cambia in accordo a questa rappresentazione dei token. La Figura 1.6 illustra questa nuova situazione e tutta la traduzione della nostra stringa di esempio.

1.3.4 Generazione del codice intermedio

Dopo l'analisi sintattica e semantica alcuni compilatori generano una rappresentazione intermedia esplicita del programma sorgente. Possiamo pensare a questa rappresentazione intermedia come ad un programma scritto nel linguaggio di una macchina astratta che è poco distante dalla macchina ospite verso cui stiamo compilando. Infatti le proprietà principali di questo linguaggio dovrebbero essere le seguenti: deve essere abbastanza semplice

compilata in un altro modo che riflette il significato dato nella specifica del linguaggio.

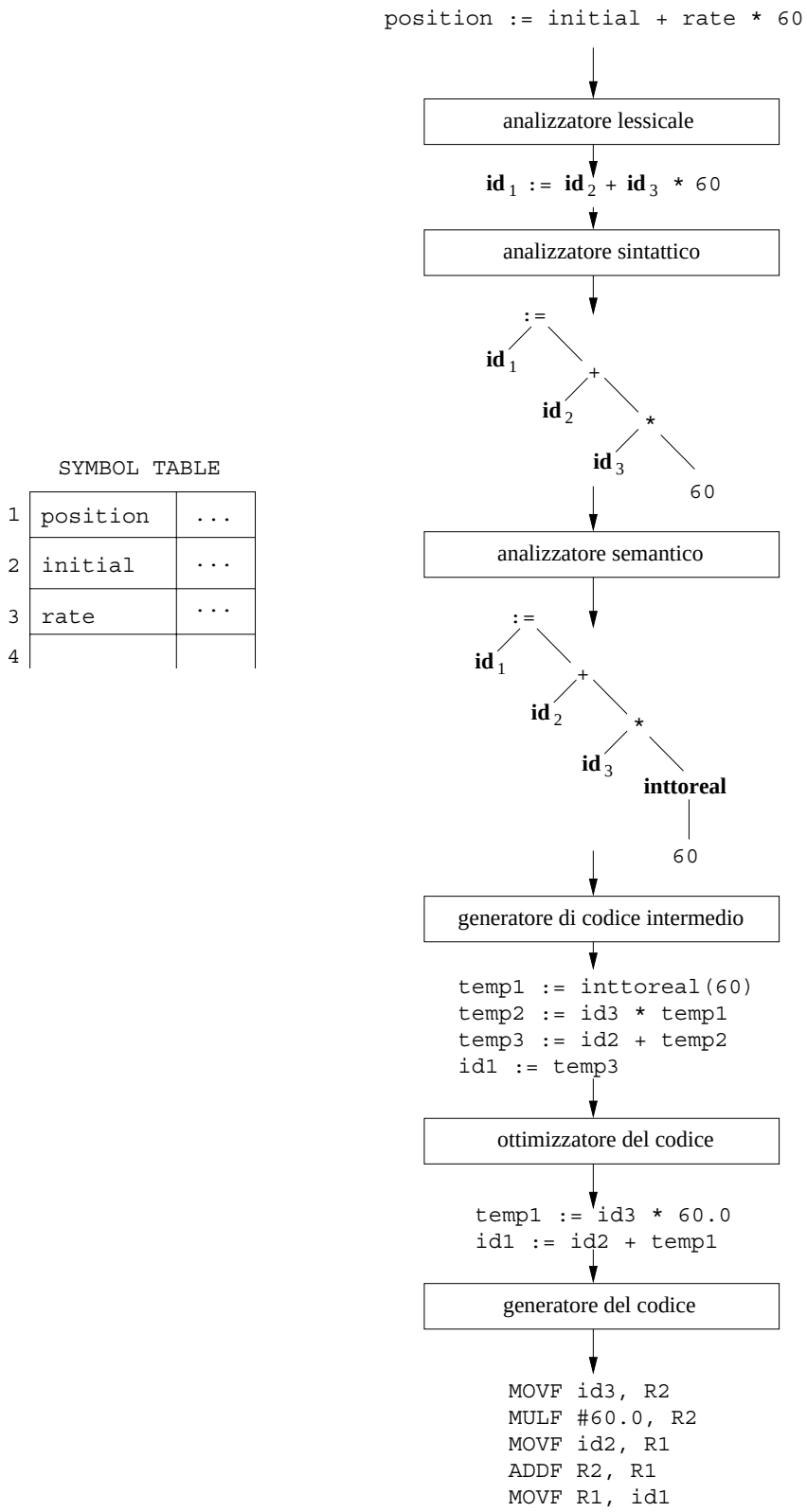


Figura 1.6: Traduzione di uno statement.

da produrre e anche abbastanza semplice da tradurre nel linguaggio della macchina ospite.

Esistono diversi tipi di rappresentazioni intermedie. Noi ne useremo una chiamata "codice a tre indirizzi" (three-address code) che è simile ad un linguaggio assembly di una macchina a registri in cui una cella di memoria può essere usata, nelle operazioni, allo stesso modo di un registro della CPU. Il codice a tre indirizzi consiste di sequenze di istruzioni ognuna delle quali ha al massimo tre operandi. In Figura 1.6 è indicato il codice a tre indirizzi generato per la stringa di esempio:

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Questo particolare tipo di forma intermedia ha le seguenti caratteristiche:

1. Ogni istruzione ha al più un operatore in aggiunta all'assegnamento. In questo modo, nel generare queste istruzioni, il compilatore deve decidere l'ordine in cui vengono fatte le operazioni (la moltiplicazione precede l'addizione nell'esempio)

2. Il compilatore deve generare un nome temporaneo per denotare il valore calcolato da ogni istruzione

3. Alcune istruzioni hanno meno di tre operandi (ad esempio la prima e l'ultima della traduzione della nostra stringa di esempio)

1.3.5 Ottimizzazione del codice

Questa fase cerca di migliorare il codice intermedio in modo da renderlo più veloce quando sarà eseguito sulla macchina ospite.

Alcune ottimizzazioni sono banali: ad esempio il codice generato con un semplice algoritmo che traduce automaticamente i costrutti a partire dall'albero sintattico risultante dall'analizzatore semantico (ad esempio il codice prodotto per la nostra stringa di esempio) contiene sicuramente dei nomi temporanei che non sono necessari.

Un esempio lampante è l'ultima istruzione dell'esempio (`id3 := temp3`). Questa è stata generata perchè il nome `temp3` è stato creato come riferimento per il risultato del sotto-albero destro dell'albero sintattico principale.

Inoltre alcune operazioni che sono indicate nell'albero sintattico possono essere fatte una volta per tutte a tempo di compilazione e quindi non devono essere tradotte. È il caso ad esempio dell'operazione `inttoreal(60)`.

Quindi il codice intermedio può essere trasformato nel seguente codice equivalente:

```
temp1 := id3 + 60.0
id1 := id2 + temp1
```

Ci sono moltissimi tipi di ottimizzazione che un compilatore può fare e si ha quasi sempre che il tempo di calcolo maggiore di tutta la compilazione viene speso in questa fase. In ogni caso ci sono diverse semplici ottimizzazioni che non rallentano in maniera eccessiva la compilazione, ma per contro riescono a velocizzare significativamente il tempo di esecuzione del programma target.

1.3.6 Generazione del codice

L'ultima fase di un compilatore è quella della generazione del codice target (della macchina ospite) che normalmente è un codice macchina (oppure assembly) rilocabile.

Celle di memoria in uno spazio logico sono assegnate ad ogni variabile del programma e il codice intermedio ottimizzato viene tradotto nel linguaggio della macchina ospite. Un aspetto cruciale di questa traduzione è l'assegnazione dei valori delle variabili ai registri. Ad esempio, usando due registri della CPU (R1 ed R2), il codice ottimizzato del nostro esempio diventa il seguente (come era già stato indicato in Figura 1.6):

```
MOVF id3, R2
MULF \#60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

In questo codice assembly i due operandi, separati da virgola, rappresentano rispettivamente una sorgente e una destinazione. La lettera `F` alla fine del nome mnemonico delle istruzioni indica che gli operandi sono numeri reali in rappresentazione floating-point. La prima istruzione copia il valore contenuto nella variabile `id3` nel registro 2 e poi lo moltiplica con la costante floating-point `60.0`[2]. Il risultato delle operazioni aritmetiche fra due registri viene posto nel registro destinazione. L'istruzione successiva carica nel registro 1 il valore della variabile `initial` e il risultato intermedio precedente viene aggiunto a quest'ultimo valore. Infine il risultato viene posto in `id1`, cioè la zona di memoria dove è stato allocato spazio per la variabile `position`.

[2]il fatto che è una costante è indicato dal carattere speciale `#` e il fatto che è floating-point dal punto decimale.

Capitolo 2

Analisi Lessicale

Il compito della fase di analisi lessicale è quello di fornire alle fasi successive uno stream di token a partire dallo stream di caratteri che rappresenta il programma che si vuole compilare. In Figura 2.1 è rappresentato il ruolo dell'analizzatore lessicale rispetto alle componenti del compilatore con cui coopera.

Tramite una chiamata di un metodo (o funzione in C) `get_next_token()` dell'analizzatore, il parser ottiene un nuovo token. Per far questo, l'analizzatore dovrà leggere i caratteri di input e analizzarli fino al riconoscimento di un token. Se questo è un identificatore, si occupa di creare una riga per lui nella tabella dei simboli. Nel caso che nessun token sia riconosciuto, viene segnalato un errore (in una implementazione Java, ad esempio, si potrebbe usare un'eccezione apposita).

L'analizzatore lessicale provvede ad eliminare dal testo gli spazi bianchi[1], che separano i token, ed i commenti. Questi ultimi, infatti, sono utili per il programmatore, ma vengono ignorati dal compilatore.

Un altro compito importante è quello di associare i giusti numeri di riga con i messaggi di errore, in modo tale che il programmatore possa trovare facilmente il punto del programma in cui è stato trovato un errore.

Si noti che l'analisi lessicale potrebbe venire facilmente considerata una parte dell'analisi sintattica. Le seguenti considerazioni sono alcuni dei principali motivi per cui conviene separarle:

- Semplicità della progettazione. Separare le due fasi permette di semplificare una delle due: un parser che non debba tener conto degli spazi bianchi o dei commenti è più semplice da scrivere.

- Efficienza della compilazione migliorata. Un analizzatore lessicale come parte distinta permette di utilizzare tecniche specifiche per questo tipo

[1] Si considerano spazi bianchi i blank, i caratteri di tabulazione ed i caratteri di newline.

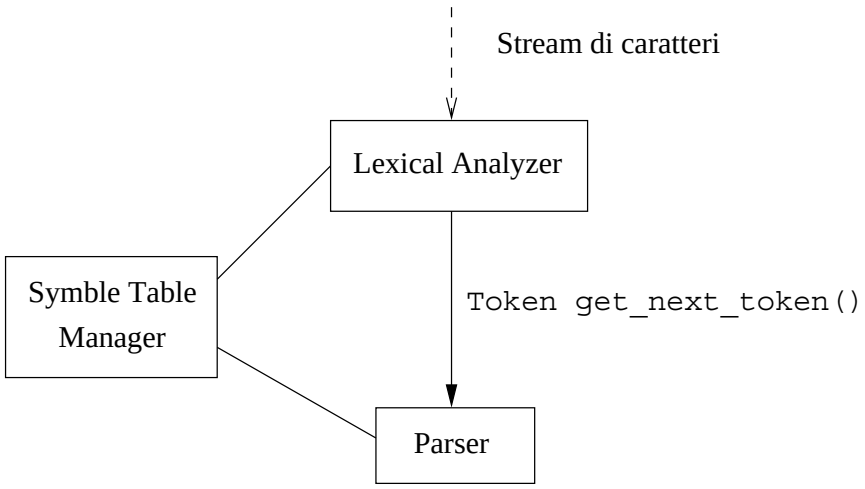


Figura 2.1: Interazioni dell'analizzatore lessicale.

di analisi che migliorano le prestazioni (es. tecniche di bufferizzazione). Inoltre, i formalismi e gli algoritmi usati per descrivere/riconoscere i token sono più semplici rispetto a quelli che sono necessari per riconoscere la struttura del programma. Ne consegue una implementazione più semplice ed efficiente.

2.1 Token, pattern e lexeme

In questa sezione diamo un significato più preciso a dei termini che useremo spesso. Abbiamo già visto informalmente che cosa si intende per token.

In generale ad ogni token dato in output dall'analizzatore lessicale corrisponde un insieme di stringhe dell'input. Questo insieme di stringhe viene descritto tramite da una regola che viene chiamata *pattern* associato al token. Diciamo che un pattern "fa *match*"[2] con ogni stringa nell'insieme associato al token. Un *lexeme* è una sequenza di caratteri nel programma sorgente che fa match con il pattern di un certo token. Ad esempio, nella frase Pascal `const pi = 3.1416;` la sottostringa `pi` è un lexeme per il token "identificatori".

In Tabella 2.2 ci sono una serie di token con relativo pattern (specificato per ora in maniera informale) e alcuni esempi di lexeme.

I token sono trattati come simboli terminali della grammatica che genera il linguaggio sorgente. Come convenzione useremo sempre il grassetto per indicare i token (e i simboli terminali). Ogni token può essere pensato come

[2]In alcuni casi si potrebbe essere tentati di coniare nuovi idiomi tipo "mecciare" (io meccio, tu mecci, egli meccia...), ma in queste note si è preferita la linea linguistica purista.

TOKEN	ESEMPI DI LEXEME	DESCR. INFORM. DEL PATTERN
const	`const`	`const`
if	`if`	`if`
relation	`<, <=, =, <>, >, >=`	`<` o `<=` o `=` $\cdots$
id	`pi, count, D2`	lettera seguita da cifre e/o lettere
num	`3.1416, 0, 6.02E32`	Una qualsiasi costante numerica
literal	`"Inserire codice: "`	Tutti i caratteri tra " e " eccetto "
punt	`; , : ( ) { }`	Un carattere di punteggiatura

Figura 2.2: Esempi di token, pattern e lexeme.

una *categoria lessicale* in cui tutti i lexeme con un significato comune vengono raggruppati. Il raggruppamento dipende dal tipo di linguaggio e dalle scelte del progettista. Ad esempio si può creare una categoria lessicale che raggruppa tutte le parole riservate del linguaggio oppure creare una categoria lessicale per ogni parola riservata.

Il ritorno di un token da parte dell'analizzatore lessicale viene sempre implementato ritornando un numero intero che corrisponde al token. Noi denoteremo tale numero intero con il nome della categoria lessicale in grassetto (es. **id**). Vedremo nel seguito che oltre a questo numero, in alcuni casi, l'analizzatore ritorna anche un altro valore.

Un pattern è una regola che descrive l'insieme dei lexeme che possono essere riconosciuti come token all'interno del programma sorgente. Il pattern per il token **const** in Tabella 2.2 è semplicemente lo spelling carattere per carattere dell'unico lexeme della categoria lessicale, ovvero la sequenza `const`. Il pattern per il token **relation** è un insieme di sequenze di caratteri (in questo caso gli operatori relazionali del Pascal). Per descrivere precisamente pattern più complicati, come ad esempio quelli per **id** o **num**, utilizzeremo il formalismo delle *espressioni regolari*, che introdurremo nel seguito.

2.2 Attributi per i token

Quando diversi lexeme fanno match con uno stesso pattern l'analizzatore lessicale deve dare un'informazione ulteriore oltre al numero identificativo del token corrispondente. Questa informazione è importante per le fasi successive della compilazione. Ad esempio sia il carattere `1` che il carattere `0` fanno match con il pattern associato a **num**, ma è essenziale per il generatore di codice intermedio (fase successiva) conoscere esattamente quale numero era presente nel codice sorgente.

Questo tipo di informazioni aggiuntive per i token vengono trattate dal-

l'analizzatore lessicale come *attributi* dei token stessi. Si può dire che il token influenza le decisioni del parser, mentre gli attributi influenzano la traduzione vera e propria dei token. Nella pratica ogni token ha al più un attributo che è un puntatore ad una entrata della tabella dei simboli in cui sono memorizzate tutte le varie informazioni relative al token.

Esempio 2.1 *Consideriamo il seguente statement Fortran:*

```
E = M * C ** 2
```

I seguenti sono i token che sono generati dall'analizzatore lessicale in corrispondenza di questa riga:

- <**id**, *puntatore all'entrata per* `E` *nella tabella dei simboli* >
- <**assign_op**, >
- <**id**, *puntatore all'entrata per* `M` *nella tabella dei simboli* >
- <**mult_op**, >
- <**id**, *puntatore all'entrata per* `C` *nella tabella dei simboli* >
- <**exp_op**, >
- <**num**, *numero intero 2* >

Dall'esempio precedente si vede che in alcuni casi non è necessario specificare nessun attributo perché la prima componente della coppia è sufficiente per identificare il lexeme. Inoltre in questo esempio il valore dell'attributo per il settimo token **num** è il numero intero 2 (cioè la costante intera corrispondente al lexeme). Equivalentemente l'analizzatore lessicale avrebbe potuto inserire il lexeme "`2`" nella tabella dei simboli e mettere come valore dell'attributo per **num** il puntatore all'entrata corrispondente. È una scelta del progettista del compilatore.

2.3 Errori lessicali

L'analizzatore lessicale ha una visione molto locale del programma sorgente e quindi non ha la possibilità di accorgersi di molti errori. Ad esempio, se in un programma C viene incontrata la stringa `fi` nel contesto

```
fi (a== f(x) ) ···
```

l'analizzatore lessicale non è in grado di dire se `fi` è in realtà la stringa `if` scritta male oppure un identificatore non ancora apparso nel programma sorgente. Dato che `fi` è un identificatore valido, esso viene inserito nella

tabella dei simboli e l'errore verrà riconosciuto più tardi in una delle fasi successive (probabilmente l'analisi sintattica).

L'unica situazione in cui l'analizzatore lessicale può riconoscere un errore è quando nessuno dei suoi pattern fa match con un prefisso dell'input rimanente. In questo caso viene segnalato un errore e l'analizzatore può ad esempio utilizzare la strategia "panic mode" per andare avanti e riconoscere il prossimo token.

2.4 Specificare i token

Quella delle espressioni regolari è una notazione importante per specificare i pattern. Le espressioni regolari sono usate in molti contesti, oltre a quello che vedremo in questo corso. Esse denotano insiemi di stringhe e possono essere considerate un metalinguaggio per specificare linguaggi (regolari).

2.4.1 Stringhe e linguaggi

Cominciamo ad introdurre alcune convenzioni che serviranno da qui in poi. I termini *alfabeto* o *classe di caratteri* denotano un qualsiasi insieme *finito* di simboli. Ad esempio l'insieme $\{0, 1\}$ è l'alfabeto binario. ASCII e Unicode sono esempi di alfabeti molto utilizzati nei computer.

Una *stringa su un alfabeto* è una sequenza *finita* di simboli presi dall'alfabeto. Nell'ambito della teoria dei linguaggi i termini "parola" o "frase" sono sinonimi di "stringa". Stringhe generiche verranno nel seguito indicate con le variabili $s, s', s'', \ldots, s_0, s_1, s_2, \ldots$ oppure $x, y, z, w, \ldots, x', x'', \ldots, x_0, x_1, \ldots$ La lunghezza di una stringa s verrà indicata con $|s|$ ed è il numero di simboli che costituiscono la sequenza s. Ad esempio, `banana` è una stringa lunga 6. La *stringa vuota* è denotata con ϵ e ha lunghezza 0. I seguenti sono altri termini usati in questo contesto con le relative definizioni:

- *Prefisso di s* Una stringa ottenuta togliendo zero o più simboli dalla fine di s. Es. `ban` è prefisso di `banana`.

- *Suffisso di s* Una stringa ottenuta togliendo zero o più simboli dalla testa di e. Es. `nana` è suffisso di `banana`.

- *Sottostringa di s* Una stringa ottenuta togliendo da s sia un suo suffisso che un suo prefisso. Es. `ana` è una sottostringa di `banana`. Si noti che ogni prefisso o suffisso di s è anche una sottostringa di s (ma non è vero il contrario). Si noti anche che s stessa è anche suo prefisso, suffisso e sottostringa.

- *Prefisso, suffisso e sottostringa propri di s* Una qualsiasi stringa non vuota x che sia, rispettivamente, un prefisso, un suffisso o una sottostringa di s e tale che $s \neq x$.

- *Sottosequenza di s* Una qualsiasi stringa ottenuta cancellando zero o più caratteri, non necessariamente contigui, da s. Es. `baaa` è una sottosequenza di `banana`.

Il termine *linguaggio* denota un qualsiasi insieme (finito o infinito) di stringhe su un certo alfabeto fissato. Questa definizione è molto generale. Ad esempio l'insieme vuoto ($\emptyset$ o $\{\}$) è un linguaggio (chiamato *linguaggio vuoto*) sotto questa definizione. Un altro linguaggio particolare è il linguaggio $\{\epsilon\}$, che contiene una sola stringa, la stringa vuota. Altri esempi possono essere: tutti i programmi Pascal sintatticamente ben formati o addirittura tutte le frasi in italiano che sono grammaticalmente corrette. Tuttavia c'è subito da notare che quest'ultimo linguaggio è molto difficile da definire precisamente, mentre altri linguaggi non banali possono essere definiti matematicamente (come facciamo noi con i linguaggi di programmazione tramite le grammatiche). Come ultima cosa si noti che la definizione che abbiamo dato non richiede che sia associato nessun significato particolare alle stringhe del linguaggio. La disciplina che si occupa dei metodi per dare significato alle stringhe di un linguaggio viene chiamata generalmente *semantica*.

Richiamiamo alcune operazioni di base sulle stringhe. Se x e y sono due stringhe, allora la *concatenazione* di x ed y, scritta come xy, è una stringa ottenuta attaccando y in fondo a x. Es. $x =$ `buon` e $y =$ `giorno` $xy =$ `buongiorno`. L'elemento neutro per la concatenazione è la stringa vuota: $\epsilon s = s\epsilon = s$ per ogni stringa s. Se pensiamo alla concatenazione come ad un prodotto, possiamo definire l'analogo dell'elevamento a potenza come segue:

- $s^0 = \epsilon$ per ogni stringa s

- $S^i = ss^{i-1}$ per ogni stringa s e per ogni $i > 0$

Ad esempio $\texttt{otto}^0 = \epsilon$, $\texttt{otto}^1 =$ `otto`, $\texttt{otto}^3 =$ `ottoottootto`.

2.4.2 Operazioni sui linguaggi

Ci sono diverse importanti operazioni che possono essere applicate ai linguaggi. Per quanto riguarda l'analisi lessicale ci limiteremo a considerare le seguenti:

- *Unione* $L_1 \cup L_2 = \{s \mid s \in L_1 \vee s \in L_2\}$

- *Concatenazione* $L_1 L_2 = \{s_1 s_2 \mid s_1 \in L_1 \wedge s_2 \in L_2\}$

- *Esponenziazione* $L^0 = \{\epsilon\}$, $L^i = LL^{i-1}$ se $i > 0$
- *Chiusura (o stella) di Kleene* $L^* = \bigcup_{i=0}^{\infty} L^i$
- *Chiusura (o stella) positiva di Kleene* $L^+ = \bigcup_{i=1}^{\infty} L^i$

Esempio 2.2 *Sia* L *l'insieme* $\{A, B, \ldots, Z, a, b, \ldots, z\}$ *e* D *l'insieme* $\{0, 1, \ldots, 9\}$. L *e* D *possono essere visti in due modi: come alfabeti finiti o come linguaggi (finiti) composti da parole che sono tutte lunghe un carattere. Vediamo alcuni linguaggi che possono essere definiti a partire da* L *e* D *tramite le operazioni che abbiamo appena introdotto:*

- $L \cup D$ *è l'insieme delle lettere e delle cifre*
- LD *è l'insieme di tutte le stringhe di lunghezza due formate da una lettera seguita da una cifra*
- L^4 *contiene tutte le stringhe di lunghezza 4 che sono formate da lettere*
- L^* *è un insieme infinito di stringhe ognuna delle quali ha una lunghezza finita ed è composta da lettere. Questo linguaggio contiene, per definizione, anche la stringa vuota* ϵ.
- $L(L \cup D)^*$ *è l'insieme di tutte le stringhe di lunghezza strettamente maggiore di 0, che iniziano con una lettera e sono composte, dopo la prima lettera, da cifre o lettere.*
- D^+ *è l'insieme di tutte le stringhe di cifre formate da almeno una cifra*

2.4.3 Espressioni regolari

In questa sezione definiamo un formalismo che ci permetterà di definire un certo tipo di linguaggi. Ad esempio potremo definire il linguaggio che contiene tutte le stringhe che possono essere viste come identificatori nel modo seguente:

letter (**letter** | **digit**)*

La barra verticale significa "or", le parentesi sono usate per raggruppare sottoespressioni, l'asterisco indica "zero o più occorrenze" dell'espressione fra parentesi e la giustapposizione di **letter** con il resto dell'espressione significa concatenazione.

L'insieme delle espressioni regolari su un certo alfabeto Σ è definito induttivamente dalle regole specificate nel seguito. Ogni espressione regolare su Σ definisce in maniera univoca un linguaggio su Σ. La definizione induttiva specifica (usando l'induzione sulla struttura delle espressioni) anche come viene formato il linguaggio definito da ogni espressione regolare:

Definizione 2.3 (Espressioni Regolari) *Sia Σ un alfabeto finito. L'insieme delle espressioni regolari su Σ è costituito da tutte e sole le espressioni generate induttivamente come segue. Associamo ad ogni espressione anche il linguaggio denotato.*

1. *ϵ è una espressione regolare che denota il linguaggio $\{\epsilon\}$*
2. *Se a è un simbolo di Σ allora a è un'espressione regolare che denota il linguaggio $\{a\}$*
3. *Siano r ed s due espressioni regolari che denotano i linguaggi $L(r)$ ed $L(s)$ rispettivamente. Allora:*
   - *$(r)|(s)$ è una espressione regolare che denota il linguaggio $L(r) \cup L(s)$*
   - *$(r)(s)$ è una espressione regolare che denota il linguaggio $L(r)L(s)$*
   - *$(r)^*$ è una espressione regolare che denota il linguaggio $L(r)^*$*
   - *(r) è una espressione regolare che denota il linguaggio $L(r)$*

Un linguaggio denotato da una espressione regolare viene chiamato insieme regolare.

Per evitare di scrivere troppe parentesi, assegniamo una precedenza ed una regola di associatività agli operatori che costruiscono espressioni regolari così da poter scrivere espressioni con meno parentesi, ma che abbiano un significato univoco. La precedenza è la seguente:

1. L'operatore unario * lega maggiormente
2. La concatenazione è il secondo operatore che lega maggiormente ed è associativa a sinistra (es. $abc = (ab)c$)
3. L'operatore che lega meno di tutti è l'or ($|$) ed è anch'esso associativo a sinistra

Con queste convenzioni, ad esempio, l'espressione $(a)|((b)^*(c))$ può essere scritta come $a|b^*c$. Entrambe denotano il linguaggio delle stringhe che o sono a oppure sono una sequenza, eventualmente vuota, di b seguite da una c.

Esempio 2.4 *Sia $\Sigma = \{a, b\}$.*

- *$a|b$ denota il linguaggio $\{a, b\}$*
- *$(a|b)(a|b)$ denota il linguaggio $\{aa, ab, ba, bb\}$,*
  denotabile anche con $aa|ab|ba|bb$

ASSIOMA	DESCRIZIONE
$r\|s = s\|r$	$\|$ è commutativo
$r\|(s\|t) = (r\|s)\|t$	$\|$ è associativo
$(rs)t = r(st)$	la concatenazione è associativa
$r(s\|t) = rs\|rt$ $(s\|t)r = sr\|tr$	la concatenazione è distributiva rispetto a $\|$
$\epsilon r = r$ $r\epsilon = r$	ϵ è l'elemento neutro per la concatenazione
$r^* = (r\|\epsilon)^*$	relazione tra * e $\|$
$r^{**} = r^*$	* è idempotente

Figura 2.3: Proprietà algebriche delle espressioni regolari

- a^* *denota il linguaggio* $\{\epsilon, a, aa, aaa, aaaa, \ldots\}$
- $(a|b)^*$ *denota il linguaggio di tutte le stringhe formate dalla concatenazione di un numero qualsiasi di* a *e* b *messe in un qualsiasi ordine, più la stringa vuota. Ad esempio* $\epsilon, aab, baababa, a, b, bbbb, aaaa, abab, \ldots$

Se due espressioni regolari denotano lo stesso linguaggio diciamo che sono *equivalenti*.

Le espressioni regolari possono essere viste come un'algebra i cui termini (sui simboli di Σ e su ϵ) sono formati attraverso le tre operazioni. Come tipicamente si fa per un'algebra, diamo, in Figura 2.3, delle equazioni che descrivono alcune proprietà degli operatori.

2.4.4 Definizioni regolari

Per motivi di chiarezza della notazione molto spesso è utile identificare delle espressioni regolari e dar loro un nome. Questo nome può poi essere usato all'interno di altre espressioni regolari negli stessi posti in cui puó essere inserito un simbolo dell'alfabeto. Per ottenere l'espressione regolare senza queste "macro" basta rimpiazzare ogni nome con la relativa espressione regolare. Una serie di definizione di nomi di espressioni regolari si chiama *definizione regolare* ed è una sequenza del tipo:

$$\begin{aligned} d_1 &\to r_1 \\ d_2 &\to r_2 \\ &\cdots \\ d_n &\to r_n \end{aligned}$$

dove i d_i $(i = 1, 2, \ldots, n)$ sono tutti nomi distinti e ogni r_i $(i = 2, 3, \ldots, n)$ è una espressione regolare sull'alfabeto $\Sigma \cup \{d_1, \ldots, d_{i-1}\}$. Si noti che per

definire l'espressione per un certo nome d_i è possibile usare i nomi definiti in precedenza.

Per distinguere i nomi definiti dai simboli dell'alfabeto vero e proprio, useremo la convenzione di scrivere i primi in grassetto.

Esempio 2.5 *Scriviamo una definizione regolare per gli identificatori e le costanti numeriche Pascal (queste ultime possono essere intere come ad esempio 343, razionali come 45.56 o con esponente: 6.343E4, 1.4322E-20).*

$$\begin{aligned}
\textbf{letter} &\rightarrow 0|1|\cdots|9 \\
\textbf{digit} &\rightarrow A|B|\cdots|Z|a|b|\cdots|z \\
\textbf{id} &\rightarrow \textbf{letter}(\textbf{letter}|\textbf{digit})^* \\
\textbf{digits} &\rightarrow \textbf{digit}\ \textbf{digit}^* \\
\textbf{optional_fraction} &\rightarrow .\textbf{digits}|\epsilon \\
\textbf{optional_exponent} &\rightarrow (E(+|-|\epsilon)\textbf{digits})|\epsilon \\
\textbf{num} &\rightarrow \textbf{digits optional_fraction optional_exponent}
\end{aligned}$$

I nomi **id** e **num**, una volta sostituiti ricorsivamente tutti i nomi contenuti all'interno delle loro espressioni regolari associate, corrisponde ad una definizione regolare che denota il linguaggio degli identificatore e delle costanti numeriche Pascal, rispettivamente.

2.5 Automi a stati finiti

In questa sezione definiamo gli automi a stati finiti non deterministici e deterministici. Essi sono dei riconoscitori di stringhe di un certo linguaggio. Vedremo che tutti i linguaggi regolari, cioè quelli denotabili da espressioni regolari, possono essere accettati anche da un automa a stati finiti (è vero anche il viceversa). Dopo la definizione degli automi vedremo due algoritmi importanti: la costruzione dei sottoinsiemi per trasformare un automa non deterministico in deterministico e l'algoritmo per minimizzare il numero degli stati di un automa deterministico.

2.5.1 Automi non deterministici

Definiamo la classe degli automi finiti non deterministici NFA (Non-deteriministic Finite Automata). Sia Σ un alfabeto finito.

Definizione 2.6 (NFA) *Un* NFA *su* Σ *è una tupla* $\langle S, \Sigma, move, s_0, F\rangle$ *dove:*

- S *è un insieme finito di* stati

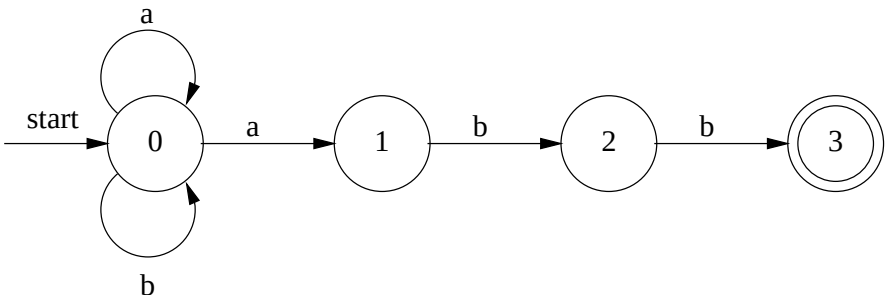


Figura 2.4: Un NFA.

- Σ *è l'alfabeto dei simboli*
- $s_0 \in S$ *è lo* stato iniziale
- $F \subseteq F$ *è l'insieme degli* stati di accettazione *o* stati finali
- *move*: $(S \times (\Sigma \cup \{\epsilon\})) \longrightarrow \wp(S)$ *è una funzione che specifica per ogni stato* s *e per ogni simbolo* x *di* Σ *le* transizioni etichettate x *dallo stato* s *ad un insieme, anche vuoto, di stati di destinazione (c'è una transizione per ogni stato di destinazione).*

Un NFA può essere rappresentato graficamente tramite un diagramma in cui i cerchi sono gli stati, le frecce etichettate sono le transizioni, lo stato iniziale ha una freccia entrante con scritto *start* e gli stati finali hanno una doppia cerchiatura. Ogni freccia può essere etichettata da un simbolo di Σ o da ϵ, la stringa vuota. Le transizioni di quest'ultimo tipo si chiamano ϵ-transizioni.

Un primo esempio di NFA si trova in Figura 2.4. S è l'insieme $\{0, 1, 2, 3\}$, $\Sigma = \{a, b\}$, lo stato iniziale è 0, l'unico stato finale è 3 ($F = \{3\}$) e non ci sono ϵ-transizioni. Le frecce rappresentano le transizioni: ad esempio le due frecce etichettate a che partono dallo stato 0 e vanno una nello stato 0 e l'altra nello stato 1 indicano che $move(0, a) = \{0, 1\}$. Inoltre $move(0, b) = \{0\}$, $move(1, a) = \{\}$ eccetera.

Un NFA può riconoscere le stringhe di un certo linguaggio. Per chiarire in quale modo un NFA accetta una stringa definiamo la nozione di cammino etichettato.

Definizione 2.7 (Cammino etichettato) *Sia* $N = \langle S, \Sigma, move, s_0, F \rangle$ *un NFA. Un cammino etichettato di lunghezza* $k \geq 0$ *su* N *è una sequenza* $s_0 \xrightarrow{x_1} s_1 \xrightarrow{x_2} s_2 \xrightarrow{x_3} \cdots \xrightarrow{x_k} s_k$ *dove:*

- s_0 *è lo stato iniziale*

- $\forall i \in \{1, 2, \ldots, k\}.\ s_i \in move(s_{i-1}, x_i)$

La stringa $x_1 x_2 \cdots x_k$ *si chiama* stringa associata al cammino *ed è in generale una stringa di* Σ^*. *Se* $k = 0$ *allora il cammino è costituito solo dallo stato iniziale e la stringa associata è la stringa vuota* ϵ.

Si noti che un cammino di lunghezza k *consiste di una sequenza di* $k + 1$ *stati dell'automa ed ha una stringa associata lunga al più* k. *Questo perchè qualcuno (o anche tutti) i simboli* x_i *potrebbero essere* ϵ.

Definizione 2.8 (Stringa accettata) *Sia* $N = \langle S, \Sigma, move, s_0, F \rangle$ *un NFA. Una stringa* $\alpha \in \Sigma^*$ *è accettata dall'automa* N **se e solo se** *esiste un cammino etichettato* $s_0 \xrightarrow{x_1} s_1 \xrightarrow{x_2} s_2 \xrightarrow{x_3} \cdots \xrightarrow{x_k} s_k$ *su* N *tale che* α *è la stringa associata al cammino e lo stato* s_k *è uno stato finale (in formule* $s_k \in \mathcal{F} \wedge \alpha = x_1 x_2 \cdots x_k$*). Se lo stato iniziale è di accettazione allora anche la stringa vuota* ϵ *è accettata dall'automa.*

Possiamo ora introdurre la nozione di linguaggio accettato dall'automa.

Definizione 2.9 (Linguaggio accettato) *Sia* $N = \langle S, \Sigma, move, s_0, F \rangle$ *un NFA. Il linguaggio accettato dall'automa è l'insieme*

$$L(N) = \{\alpha \in \Sigma^* \mid \alpha \text{ è accettata da } N\}$$

Esempio 2.10 *Consideriamo l'automa in Figura 2.4.*

$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

è un cammino etichettato la cui stringa associata è aabb. Lo stato in cui il cammino termina, 3, è anche uno stato finale e quindi questa stringa è accettata dall'automa.

Essendo l'automa non deterministico, tuttavia, esiste un altro cammino etichettato con la stringa precedente:

$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$

in questo caso lo stato 0 non è finale e quindi questo cammino non porta all'accettazione della stringa. Si noti che, comunque, la definizione richiede che ci sia almeno *un cammino etichettato che termina in uno stato finale per determinare se la stringa associata è accettata o no.*

Facendo altri esempi e considerando la struttura dell'automa è facile convincersi che il linguaggio accettato è quello denotato da $(a|b)^*abb$.

La costruzione di un cammino etichettato per una data stringa può essere vista come un algoritmo di riconoscimento di stringhe. Ad esempio, supponiamo che vogliamo controllare se la stringa *aba* fa parte del linguaggio

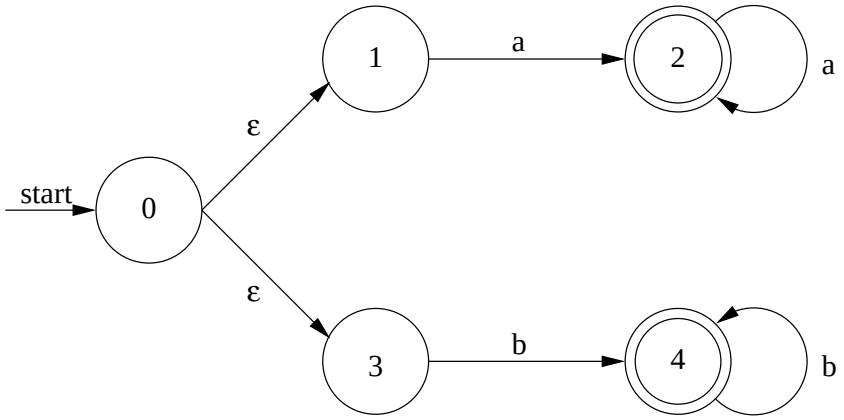


Figura 2.5: Un NFA contenente anche ϵ-transizioni.

accettato dall'automa di Figura 2.5. Partiamo dallo stato iniziale e notiamo subito che, senza considerare nessun simbolo, possiamo effettuare delle ϵ-transizioni. Manteniamo un insieme di stati "correnti" e poniamolo in questo momento a $\{0, 1, 3\}$ dato che questi sono gli stati in cui mi posso trovare cercando di costruire un cammino etichettato per la stringa *aba*.

Consideriamo ora il primo simbolo della stringa: a. A questo punto guardiamo quali sono gli stati in cui possiamo andare seguendo una transizione etichettata a uscente da uno qualsiasi degli stati correnti. L'insieme degli stati che posso raggiungere con la prima a è $\{2\}$ poichè dagli stati 0 e 3 non ci sono transizioni uscenti etichettate con a, mentre $move(1, a) = \{2\}$. Notiamo che dallo stato 2 non escono ϵ-transizioni e quindi nel processo di costruzione del cammino per *aba*, a questo stadio, possiamo trovarci solamente nello stato 2 (se ci fossero, dovremmo aggiungere agli stati correnti tutti gli stati raggiungibili con ϵ-transizioni).

Eliminiamo il primo simbolo dalla stringa in esame e consideriamo quello seguente: b. A partire da ogni stato in $\{2\}$ controlliamo in quali stati posso andare seguendo transizioni etichettate con b. Si ha che $move(2, b) = \{\}$ e quindi non possiamo più andare avanti nel riconoscimento della stringa. In questo caso diciamo che l'automa è *bloccato* e l'algoritmo di riconoscimento della stringa *termina senza accettare*.

Consideriamo invece la stringa *bb*. A partire da uno degli stati in $\{0, 1, 3\}$ posso arrivare, con una b, nell'insieme di stati $\{4\}$. Eliminiamo la prima b e consideriamo la seguente. A partire dallo stato 4 posso, con una b, arrivare nello stato 4 (utilizzando la transizione *self-loop* etichettata con b dello stato 4). Il nuovo insieme degli stati correnti è $\{4\}$.

A questo punto abbiamo esaurito la stringa in esame. L'algoritmo termina. Per dire se termina con accettazione o senza, basta controllare, in

generale, se uno degli stati correnti è finale. In questo caso è vero: 4 è uno stato finale e quindi la stringa *bb* è accettata dall'automa.

Un cammino etichettato che porta all'accettazione della stringa è il seguente:

$0 \xrightarrow{\epsilon} 3 \xrightarrow{b} 4 \xrightarrow{b} 4$

Facendo altre prove non è difficile convincersi che l'automa accetta il linguaggio denotato da $a^+|b^+$. Formalizzeremo meglio questo algoritmo di riconoscimento più avanti.

2.5.2 Automi finiti deterministici

La definizione di automa che abbiamo dato è quella più generale, cioè quella di automa non deterministico. Diamo ora una caratterizzazione degli automi deterministici DFA (Deterministic Finite Automata).

Definizione 2.11 (Automa deterministico) *Un automa finito deterministico (DFA) è una tupla* $\langle S, \Sigma, move, s_0, F \rangle$ *dove:*

- S *è un insieme finito di stati.*
- Σ *è un alfabeto finito di simboli.*
- $move: (S \times \Sigma) \longrightarrow \wp(S)$ *è una funzione di transizione tale che per ogni stato* $s \in S$ *e per ogni simbolo* $x \in \Sigma$ *l'insieme degli stati* $move(s, x)$ *o è vuoto oppure contiene un solo stato.*
- $s_0 \in S$ *è lo stato iniziale.*
- $F \subseteq S$ *è l'insieme degli stati finali.*

In altre parole un automa è deterministico se da ogni stato non escono mai due transizioni etichettate con lo stesso simbolo e non ci sono ϵ-transizioni.

La nozione di cammino etichettato e di accettazione per un DFA è analoga a quella di un NFA. Dato che in un DFA non ci sono ϵ-transizioni e, dato uno stato e un simbolo è possibile al più una transizione etichettata con quel simbolo, si ha che per ogni stringa accettata da un DFA esiste un unico cammino etichettato con la stringa e che termina in uno stato finale.

L'algoritmo di riconoscimento di una data stringa è uguale a quello di un NFA, ma in questo caso ad ogni passo l'insieme degli stati correnti contiene uno ed un solo stato.

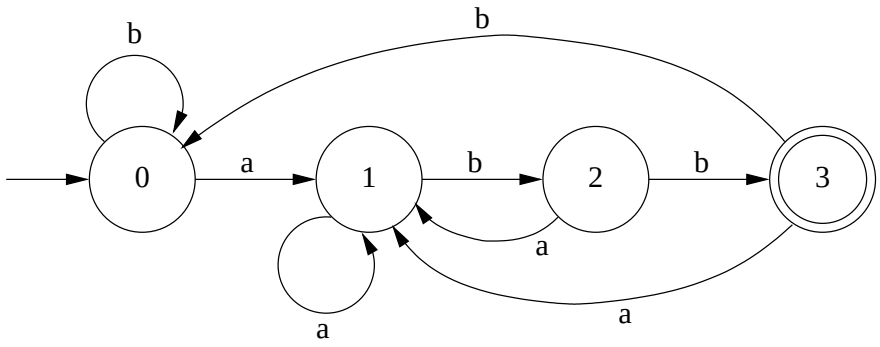


Figura 2.6: Un automa deterministico.

Esempio 2.12 *Consideriamo l'automa disegnato in Figura 2.6. Esso è un automa deterministico che accetta lo stesso linguaggio dell'automa di Figura 2.4, cioè* $(a|b)^*abb$. *L'unico cammino di accettazione per la stringa aabb è*

$0 \xrightarrow{a} 1 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

Il cammino di accettazione per la stringa abaabb è:

$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 1 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

Una funzione di transizione di un NFA o di un DFA può essere rappresentata anche con una tabella in cui le righe sono gli stati e le colonne sono i simboli dell'alfabeto (e anche ϵ nel caso non deterministico). La cella ad una riga T e ad una colonna x della tabella è l'insieme di stati che risulta da $move(T, x)$.

2.5.3 Costruzione dei sottoinsiemi

Gli automi deterministici e quelli non deterministici hanno lo stesso potere espressivo. Questo significa che se un linguaggio può essere accettato da un NFA allora esiste anche un DFA che lo accetta, e viceversa.

E' chiaro che l'algoritmo di riconoscimento di una certa stringa di un DFA è più immediato poichè non bisogna tener traccia di un insieme di stati (il cammino di accettazione, se esiste è unico). D'altra parte, da un punto di vista di progettazione, il non determinismo rende più facile scrivere un automa e/o determinare che tipo di linguaggio accetta, oltre a permettere di scrivere automi più concisi.

Basta ad esempio guardare i due automi delle Figure 2.4 e 2.6 che accettano lo stesso linguaggio. Il primo è non deterministico nello stato 0 e specifica in maniera naturale che si possono riconoscere a o b in qualsiasi numero ed ordine prima di passare ad una stringa finale obbligatoria abb.

Il secondo invece deve esplicitare una sorta di backtracking negli stati: la prima a che incontra potrebbe essere quella della stringa finale obbligatoria e quindi l'automa entra nello stato 1. Se il simbolo seguente non è una b allora l'automa continua a ciclare nello stato 1. Nello stato 2, se il simbolo seguente non è l'ultima b l'automa ritorna nello stato 1 ad aspettare di leggere un'altra a possibile candidata ad essere il primo simbolo della stringa finale obbligatoria. Infine nello stato 3 l'automa deve ritornare nello stato 0 o 1 se ci sono ancora simboli b o a, rispettivamente. Questo perchè i simboli letti precedentemente potrebbero essere il prefisso di una stringa che poi terminerà con la stringa finale obbligatoria.

In questa sezione formalizziamo una variante dell'algoritmo di cui abbiamo già parlato a parole nella Sezione 2.5.1. L'algoritmo che scriveremo è noto come *costruzione dei sottoinsiemi* (subset construction) e serve per costruire, a partire da un NFA dato, un DFA equivalente che simula il non determinismo, ma è deterministico.

Per specificare gli algoritmi, anche nel seguito, useremo uno pseudo-codice in cui le strutture dati vere e proprie non saranno specificate (in ogni caso non è difficile implementare questi algoritmi in un linguaggio di programmazione: basta definire le opportune strutture dati e le varie funzioni/procedure che specifichiamo).

Algoritmo 2.1 (Subset construction) Costruzione di un DFA a partire da un NFA.

Input: Un NFA $N = \langle S, \Sigma, move, s_0, F \rangle$

Output: Un DFA $D = \langle Dstates, \Sigma, Dtran, s'_0, F' \rangle$ equivalente a N e tale che $Dstates \subseteq \wp(S)$.

Metodo: Costruiamo la funzione di transizione *Dtran* simulando, attraverso insiemi di stati di N, i comportamenti di N "in parallelo" dovuti al non determinismo.

L'algoritmo è uno dei classici algoritmi di calcolo di un punto fisso. In pratica partiamo da un insieme di stati *Dstates* contentente solo uno stato iniziale non marcato. Ad ogni passo selezioniamo uno stato non marcato da *Dstates* e ne calcoliamo le transizioni uscenti aggiungendo a *DStates*, non marcati, eventuali nuovi stati trovati. Prima o poi, dato che il numero di stati possibili è finito (il numero di elementi di $\wp(S)$) non ci saranno più stati non marcati da considerare e l'algoritmo terminerà.

Operazioni utilizzate:

- $\epsilon\text{-}closure : S \longrightarrow \wp(S)$ tale che $\epsilon\text{-}closure(s)$ è un insieme composto da s

stesso più eventuali stati raggiungibili, in N, partendo da s e seguendo solo sequenze di ϵ-transizioni.

- ϵ-*closure* : $\wp(S) \longrightarrow \wp(S)$ tale che ϵ-*closure*(T) è un insieme composto dagli elementi di T stesso più eventuali stati raggiungibili, in N, partendo da un qualsiasi stato s di T e seguendo solo sequenze di ϵ-transizioni. In pratica è il lift dell'operazione sugli elementi alla stessa operazione su insiemi.
- *move*: $(\wp(S) \times \Sigma) \longrightarrow \wp(S)$ tale che $move(T, x) = \bigcup_{s \in T} move(s, x)$. In pratica si mettono in uno stesso insieme tutti gli stati raggiungibili con uno stesso simbolo x a partire da uno stato qualunque di T.

Algoritmo:

all'inizio ϵ-*closure*(s_0) è l'unico stato di *DStates* e non è marcato;
while c'è uno stato non marcato T in *DStates* **do begin**
  marca T;
  for each simbolo di input $x \in \Sigma$ **do begin**
    $U := \epsilon$-*closure*$(move(T, x))$;
    if U non è in *DStates* **then**
      aggiungi U, non marcato, a *DStates*;
    $Dtran(T, x) := U$;
  end;
end;
lo stato iniziale di D è ϵ-*closure*(s_0);
gli stati finali di D sono tutti quelli che contengono almeno uno stato finale di N **end**

L'operazione di ϵ-*closure* può essere realizzata semplicemente con l'uso di una pila (stack). In Figura 2.7 è specificato un semplice algoritmo di esplorazione di un grafo in profondità adattato al caso specifico.

Esempio 2.13 *Consideriamo l'automa in Figura 2.8. Il linguaggio accettato dall'automa è quello denotato dall'espressione regolare* $(a|b)^*abb$. *Applichiamo l'algoritmo della costruzione dei sottoinsiemi e troviamo un DFA equivalente.*

Calcoliamo lo stato iniziale: ϵ-*closure*$(0) = \{0, 1, 2, 4, 7\}$. *Chiamiamo questo insieme, per convenienza, A. A è il primo stato non marcato che fa parte di DStates.*

Input: Un NFA N e un insieme T di stati di N

Output: ϵ-*closure*(T)

Algoritmo:

```
metti tutti gli stati di T nella pila;
ε-closure(T) := T;
while la pila non è vuota do begin
    estrai lo stato t dalla testa della pila;
    for each stato u di N tale che t --ε--> u in N do
        if u non è in ε-closure(T) then begin
            aggiungi u ad ε-closure(T);
            inserisci u in testa alla pila;
        end
end
```

Figura 2.7: Algoritmo per il calcolo di ϵ-*closure*(T).

Alla prima iterazione selezioniamo per forza lo stato A e lo marchiamo. Calcoliamo:

- ϵ-*closure*$(move(A, a)) = \epsilon$-*closure*$(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$
- ϵ-*closure*$(move(A, b)) = \epsilon$-*closure*$(\{5\}) = \{1, 2, 4, 5, 6, 7\} = C$

Entrambi gli stati B e C sono nuovi (non si trovano attualmente in DStates) e quindi sono stati inseriti non marcati in DStates. La tabella che rappresenta la parte di Dtran calcolata fino a questo punto è la seguente:

Stato	*a*	*b*	*Marcato*
A	*B*	*C*	*Si*
B			*No*
C			*No*

Procediamo scegliendo uno stato non marcato. Prendiamo B e calcoliamo:

- ϵ-*closure*$(move(B, a)) = \epsilon$-*closure*$(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$

- ϵ-$closure(move(B, b)) = \epsilon$-$closure(\{5, 9\}) = \{1, 2, 4, 5, 6, 7, 9\} = D$

Questa volta per il simbolo a non è stato generato nessuno stato nuovo, mentre per b è stato generato il nuovo stato D. La tabella parzialmente costruita fino a questo punto è la seguente:

Stato	*a*	*b*	*Marcato*
A	*B*	*C*	*Si*
B	*B*	*D*	*Si*
C			*No*
D			*No*

Continuando ad applicare l'algoritmo si arriva a generare un ulteriore stato $E = \{1, 2, 4, 5, 6, 7, 10\}$ *dopodiché non si generano più nuovi stati e l'algoritmo termina. La tabella finale è la seguente:*

Stato	*a*	*b*	*Marcato*
A	*B*	*C*	*Si*
B	*B*	*D*	*Si*
C	*B*	*C*	*Si*
D	*B*	*E*	*Si*
E	*B*	*C*	*Si*

L'unico stato finale è E poiché è l'unico che contiene lo stato finale di N, cioè 10. L'automa è disegnato in Figura 2.9.

2.5.4 Minimizzazione di un DFA

La facilità di simulazione di un DFA rispetto a quella di un NFA è però compensata dal fatto che il DFA in genere ha un numero maggiore di stati. È lecito chiedersi, in quest'ottica, se è possibile, dato in certo DFA, trovarne uno equivalente, ma che abbia un numero minore di stati. La risposta è si ed esiste un algoritmo che trova un DFA equivalente ad un DFA dato e tale che l'automa risultato ha un numero *minimo* di stati. Qui minimo si riferisce agli stati necessari per poter accettare il linguaggio accettato dall'automa di partenza. Inoltre si ha che l'automa minimo è unico a meno di rinominare gli

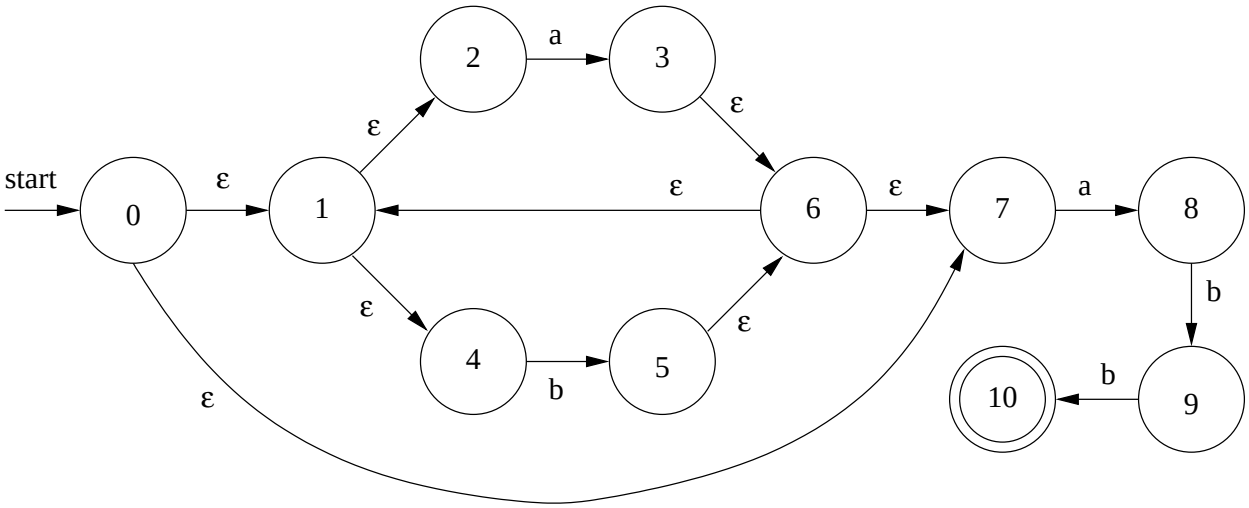


Figura 2.8: Un automa non deterministico per $(a|b)^*abb$.

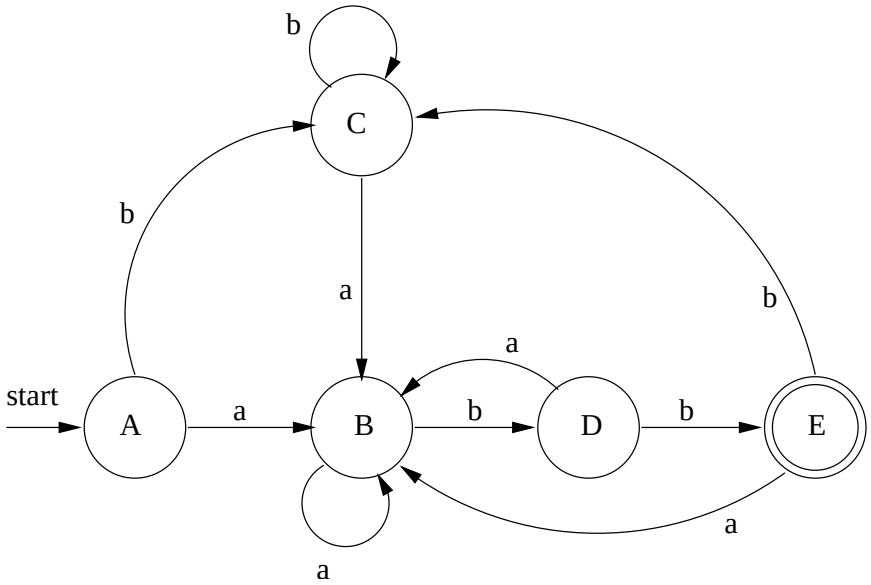


Figura 2.9: L'automa deterministico risultante dalla costruzione dei sottoinsiemi.

stati (in altre parole se trovo due automi minimi esiste sempre una funzione bigettiva fra gli stati dei due che rispetta tutte le transizioni).

L'algoritmo può essere applicato solo a DFA con una funzione di transizione che non restituisce mai l'insieme vuoto. Questa condizione non fa perdere di generalità all'algoritmo poiché è sempre possibile aggiungere un *dead state* ad un DFA nel modo seguente: si crea un nuovo stato d (il dead state appunto) che ha come transizioni uscenti dei self-loop etichettati con ognuno dei simboli di input. Questo significa che se l'automa entrerà nel dead state non ne uscirà più. Ovviamente il dead state *non* deve essere uno stato finale. Ogni volta che in un certo stato T e per un certo simbolo x si ha $move(T, x) = \{\}$ poniamo $move(T, x) := \{d\}$. In questo modo la funzione di transizione non restituisce mai l'insieme vuoto e l'automa ottenuto è equivalente a quello di partenza.

L'algoritmo di minimizzazione cerca di individuare gli stati dell'automa di partenza che si comportano nella stessa maniera e che quindi possono essere raggruppati insieme in un unico stato diminuendo così il numero totale di stati. Fare questo è la stessa cosa che individuare una partizione degli stati dell'automa tale che gli stati in ogni insieme della partizione siano indistinguibili.

Il criterio usato per distinguere gli stati è il seguente: diciamo che una stringa w distingue lo stato s dallo stato t se facendo partire il DFA di partenza dallo stato s e simulando il suo comportamento sulla stringa w arrivo in uno stato di accettazione, mentre facendo partire l'automa dallo stato t e simulandolo su w arrivo in uno stato di non accettazione.

Ad esempio la stringa ϵ distingue gli stati di accettazione da quelli di non accettazione. Ancora, la stringa bb distingue gli stati A e B del DFA in Figura 2.9 poichè partendo dallo stato A, con bb si arriva nello stato C, mentre partendo da B si arriva nello stato E.

Sia $M = \langle S, \Sigma, move, s_0, F\rangle$ un DFA con le caratteristiche richieste. L'algoritmo di minimizzazione è un algoritmo detto di *partition refining*. Esso parte dalla partizione iniziale di S formata dai due sottoinsiemi (chiamiamoli gruppi) F e $S - F$. Sicuramente gli stati di questi due gruppi sono distinti (dalla stringa ϵ). Tuttavia non possiamo ancora affermare che gli stati all'interno di uno stesso gruppo sono indistinguibili perché potrebbe esistere una certa stringa di input che li distingue. Ad ogni passo l'algoritmo raffina la partizione corrente cercando di individuare alcuni stati che si trovano nello stesso gruppo, ma che sono distinti da un simbolo x di Σ.

Sia $\Pi = \{G_1, G_2, \ldots, G_n\}$ la partizione corrente di S. Il passo di distinzione procede come descritto in Figura 2.10

Input: Π partizione di S.

Output: Π_{new} partizione di S.

Metodo:

$\Pi_{\text{new}} := \Pi$;
for each gruppo G_i $(i = 1, 2, \ldots, n)$ di Π **do begin**

partiziona G_i in sottogruppi $\mathbb{G}_1, \mathbb{G}_2, \ldots, \mathbb{G}_k$ tali che:

due stati qualsiasi s e t in G_i sono nello stesso gruppo $\mathbb{G}_j$ $(j = 1, 2, \ldots, k)$ se e solo se:

per tutti i simboli x di Σ si ha che, in M,

$s \xrightarrow{x} s'$, $t \xrightarrow{x} t'$ e s' e t' sono in uno stesso gruppo G_h $(h = 1, 2, \ldots, n)$ di Π;

$\Pi_{\text{new}} := \Pi_{\text{new}}\,[G_i \backslash \mathbb{G}_1, \mathbb{G}_2, \ldots, \mathbb{G}_k]$;

end

return Π_{new};

Figura 2.10: Passo di raffinamento dei gruppi.

Algoritmo 2.2 (Minimizzazione degli stati di un DFA)

Input: Un DFA $M = \langle S, \Sigma, move, s_0, F \rangle$ con la funzione di transizione che non restituisce mai l'insieme vuoto.

Output: Un DFA M' equivalente che ha un numero minimo di stati.

Metodo:

1. Costruisci una partizione iniziale Π in cui ci sono solo due gruppi: $G_1 = F$ (gli stati finali) e $G_2 = S - F$.

2. Applica la procedura descritta in Figura 2.10 per calcolare la nuova partizione Π_{new}.

3. Se $\Pi_{\text{new}} = \Pi$ allora poni $\Pi_{\text{final}} := \Pi$ e continua con il passo (4). Altrimenti ripeti il passo (2) con $\Pi := \Pi_{\text{new}}$.

4. Costruisci M' seguendo le indicazioni che seguono. Scegli uno stato per ogni gruppo di Π_{final} come rappresentante del gruppo. Gli stati rappresentanti sono gli stati dell'automa minimo M'. Le transizioni di M' sono costruite seguendo il seguente criterio: sia s un rappresentante e supponiamo che in M è presente una transizione $s \xrightarrow{x} t$, sia r il rappresentante del gruppo a cui appartiene lo stato t (r potrebbe essere t stesso). Allora in M' c'è la transizione $s \xrightarrow{x} r$. Lo stato iniziale di M' è il rappresentante del gruppo in cui si trova lo stato iniziale di M. Gli stati finali di M' sono i rappresentanti che sono anche stati finali di M (si noti che ogni gruppo di Π_{final} o è formato da tutti stati finali si M oppure non ne contiene nemmeno uno).

5. Se M' ha un dead state d allora rimuovi d da M'. Rimuovi anche tutti gli stati che non sono raggiungibili da un cammino che parte dallo stato iniziale. Cancella ogni transizione che entrava in d. Restituisci il risultato come automa di output.

La dimostrazione che questo algoritmo produce un automa minimo non la vediamo.

Esempio 2.14 *Minimizziamo il DFA di Figura 2.9. L'algoritmo si può applicare all'automa così com'è poiché in ogni stato ci sono transizioni uscenti per ogni simbolo dell'alfabeto. La prima partizione è formata dal gruppo* (E) *(che è l'unico stato finale) e dal gruppo* (A, B, C, D, E)*. Facciamo il primo passo di raffinamento. Notiamo subito che il primo gruppo non può essere*

raffinato ancora di più poiché è composto da un solo stato e quindi (E) *rimane tale anche in* Π_{new}*. Proviamo quindi a dividere il secondo gruppo e cominciamo con il considerare il simbolo* a *di* $\Sigma = \{a, b\}$*. Si ha:*

- $move(A, a) = B$
- $move(B, a) = B$
- $move(C, a) = B$
- $move(D, a) = B$

Quindi, per quanto concerne il simbolo a *tutti e quattro gli stati possono rimanere nello stesso gruppo. Ma vediamo l'altro simbolo,* b*. Si ha:*

- $move(A, b) = C$
- $move(B, b) = D$
- $move(C, b) = C$
- $move(D, b) = E$

In questo caso si ha che gli stati A, B *e* C *vanno a finire tutti in uno stesso gruppo della partizione corrente* Π *(il gruppo* $(ABCD)$*) mentre lo stato* D *no. Pertanto dobbiamo isolare lo stato* D *e inserire in* Π_{new} *i due nuovi gruppi* (ABC) *e* (D) *al posto del gruppo* $(ABCD)$*.*

La partizione è cambiata e quindi dobbiamo fare un'altro passo di raffinamento, con la partizione corrente Π *formata dai tre gruppi* (ABC)*,* (D) *ed* (E)*. L'unico gruppo che potrebbe raffinarsi è il primo dei tre. Si ha:*

- $move(A, a) = B$
- $move(B, a) = B$
- $move(C, a) = B$

- $move(A, b) = C$
- $move(B, b) = D$
- $move(C, b) = C$

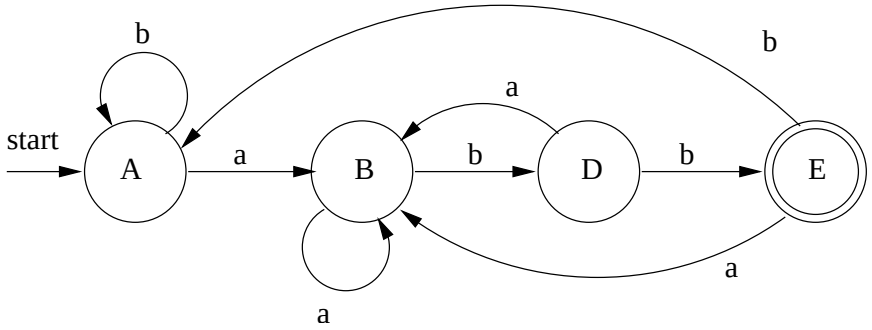


Figura 2.11: L'automa di Figura 2.9 minimizzato.

A questo stadio lo stato D non fa più parte dello stesso gruppo di C nella partizione corrente (D è stato isolato al passo precedente) e quindi siamo costretti ad isolare anche B che si differenzia dagli altri per il simbolo b. Π_{new} diventa formata da 4 gruppi: (AC), (B), (D) ed (E).

L'ultimo passo di raffinamento proviamo a farlo sul gruppo (AC). Sappiamo già che il simbolo a non discrimina. Con il simbolo b vediamo che entrambi gli stati del gruppo finiscono nello stesso stato C. Pertanto possono rimanere nello stesso gruppo. In questo passo non sono state fatte modifiche ai gruppi e quindi finiamo l'iterazione.

Dobbiamo ora scegliere i rappresentanti. L'unica scelta vera è per il gruppo (AC) (per gli altri il rappresentante può essere solo l'unico stato che li forma). Scegliamo A. L'automa che viene fuori dopo la costruzione dei passi (4) e (5) dell'algoritmo è disegnato in Figura 2.11. Si noti che è lo stesso automa di Figura 2.6 con i nomi degli stati cambiati.

2.6 Generatori automatici di analizzatori lessicali

Gli algoritmi che abbiamo studiato nelle sezioni precedenti possono essere combinati insieme in modo da ottenere, a partire dalla definizione di un certo numero di pattern, un riconoscitore dei pattern stessi. Questo processo può avvenire automaticamente ed esistono dei tool software, i generatori di analizzatori lessicali, che prendono una specifica dei pattern e restituiscono un programma che è l'analizzatore lessicale per i pattern dati. Un esempio di generatore di analizzatori lessicali è il programma `Lex`, di cui vedremo un esempio di specifica. `Lex` genera un programma in linguaggio C che riconosce i pattern dati.

In questa sezione schematizzeremo il funzionamento di un generatore di analizzatori lessicali usando gli algoritmi e le costruzioni visti nelle sezioni precedenti. Dopo ciò parleremo un po' più in dettaglio di `Lex`.

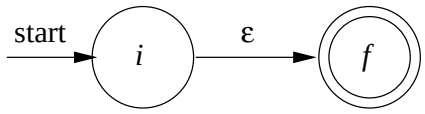
2.6.1 Dalle espressioni regolari agli automi deterministici

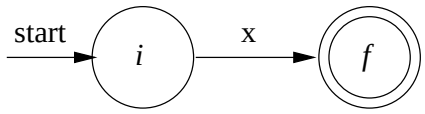
Le espressioni regolari sono il linguaggio più usato per specificare i pattern. Ciò è dovuto alla loro potenza espressiva, al fatto che sono un linguaggio formale, ma soprattutto al fatto che da un insieme di espressioni regolari si può automaticamente generare un automa che riconosce, seguendo un particolare procedimento, i pattern specificati.

Il primo passo della costruzione di un analizzatore lessicale consiste nel trasformare le espressioni regolari che definiscono i vari pattern in automi. Quella che vediamo adesso è una costruzione (detta di Thompson) che, data una qualunque espressione regolare r su un alfabeto Σ, genera un NFA che riconosce il linguaggio denotato da r e ha certe caratteristiche strutturali.

Sia r una espressione regolare su un certo alfabeto Σ. Costruiamo induttivamente un NFA N che riconosca $L(r)$.

Per prima cosa analizziamo la struttura sintattica dell'espressione regolare che risulta dalla sua definizione induttiva (vedi Definizione 2.3). Una volta individuata la struttura (un albero di derivazione) cominciamo a costruire un NFA per ogni simbolo di base, cioè ϵ o un simbolo di Σ (parte 1 e 2 di Definizione 2.3) nel modo seguente:

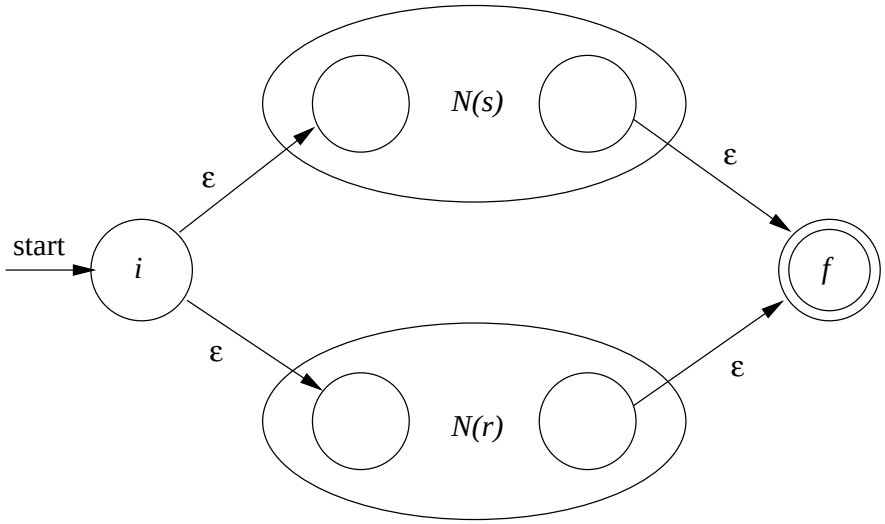




I precedenti sono i mattoni base della nostra costruzione induttiva. Ora, costruiamo induttivamente un NFA per un'espressione regolare qualsiasi partendo dagli NFA associati alle sue sottoespressioni che supponiamo di avere

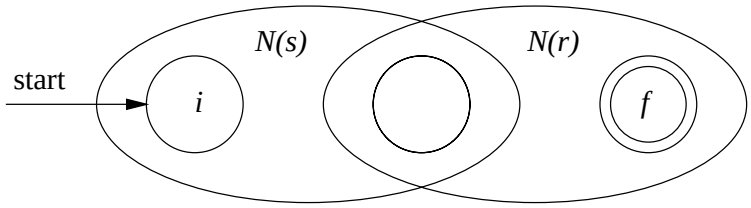
per ipotesi induttiva. Nel far questo usiamo l'accorgimento di costruire gli NFA intermedi in modo tale che: 1) abbiano un solo stato finale, 2) non abbiano nessuna transizione entrante nello stato iniziale e 3) non abbiano nessuna transizione uscente dallo stato finale (si noti che gli automi di base hanno tutte queste caratteristiche). Le seguenti sono le regole per i passi induttivi:

1. Supponiamo, per ipotesi induttiva, di aver ottenuto due NFA $N(s)$ ed $N(r)$ che riconoscono il linguaggio di s e r e con le caratteristiche volute. Un NFA che riconosce il linguaggio $L(s|r)$ e ha le caratteristiche richieste è il seguente



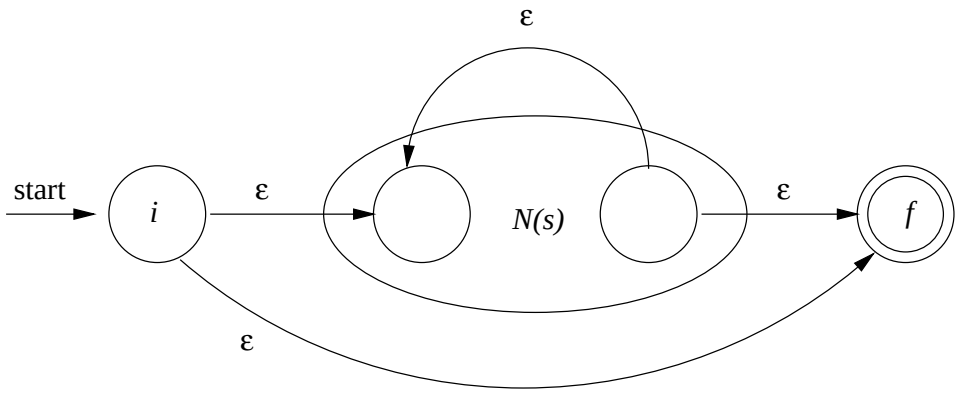
   Nella figura i è un nuovo stato iniziale ed f un nuovo stato finale. Gli ovali rappresentano gli automi $N(s)$ ed $N(r)$ che supponiamo di avere per ipotesi induttiva e i cerchi all'interno degli ovali rappresentano lo stato iniziale (quello a sinistra) e lo stato finale (quello a destra). Si noti che la doppia cerchiatura è stata eliminata dagli stati finali di $N(s)$ ed $N(r)$ e che l'unico stato finale di $N(s|r)$ è f. Stessa cosa è stata fatta per gli stati iniziali. In questo modo il nuovo stato iniziale non ha transizioni entranti e il nuovo stato finale non ha transizioni uscenti.

2. Supponiamo $N(s)$ ed $N(r)$ come sopra. Un NFA con le caratteristiche richieste che accetta $L(sr) = L(s)L(r)$ è il seguente:



Lo stato finale di $N(s)$ viene fatto coincidere con lo stato iniziale di $N(r)$ e lo stato risultante è un semplice stato intermedio, né iniziale né finale, di $N(sr)$. Lo stato iniziale di quest'ultimo è lo stesso di $N(s)$ e lo stato finale è lo stesso di $N(r)$.

3. Sia $N(s)$ un NFA che accetta $L(s)$. Costruiamo, per s^*, il seguente NFA che accetta $L(s)^*$



4. Infine, l'ultimo caso, cioè quello di un'espressione regolare tra parentesi (s), viene trattato semplicemente prendendo lo stesso $N(s)$ come NFA che accetta anche il linguaggio di (s).

Esempio 2.15 *Applichiamo la costruzione appena vista partendo dalla espressione regolare* $r = (a|b)^*abb$. *Innanzitutto individuiamo la struttura sintattica di* r *seguendo le regole di precedenza che abbiamo stabilito in Sezione 2.4.3. L'albero che rappresenta tale struttura è rappresentato in Figura 2.12*

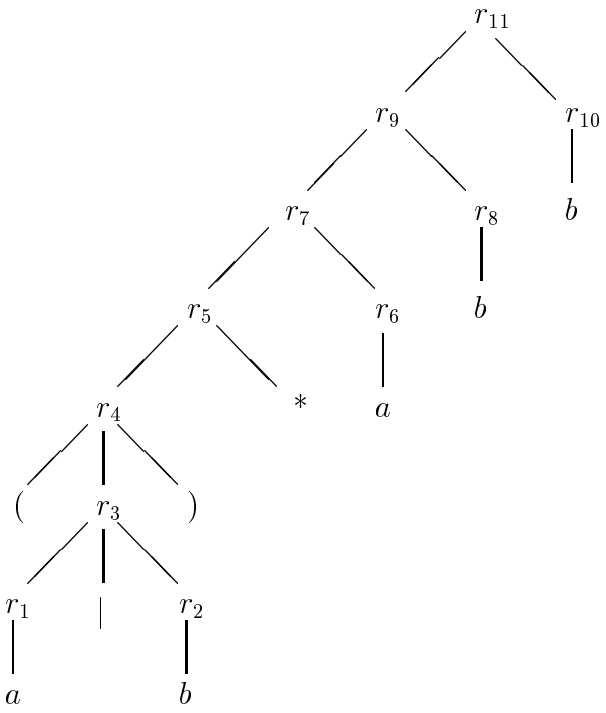
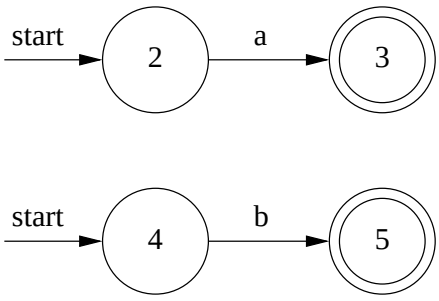


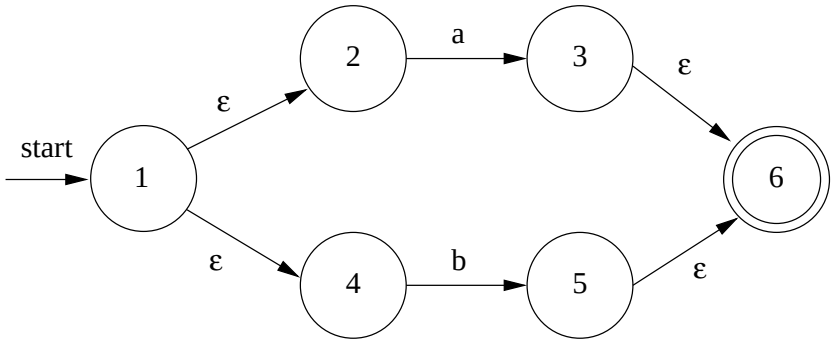
Figura 2.12: Struttura sintattica di $(a|b)^*abb$.

Abbiamo dato un nome ad ogni nodo intermedio e il nodo chiamato r_{11}, la radice dell'albero, corrisponde alla nostra r. Per ogni nodo interno, seguendo la definizione della costruzione di Thompson, costruiamo un NFA che riconosce il linguaggio denotato dalla corrispondente sottoespressione regolare. Per questo processo partiamo dalle foglie dell'albero e procediamo verso l'alto andando a costruire gli automi per le espressioni r_i utilizzando quelli precedentemente costruiti per le loro sottoespressioni (associate ai nodi figli).

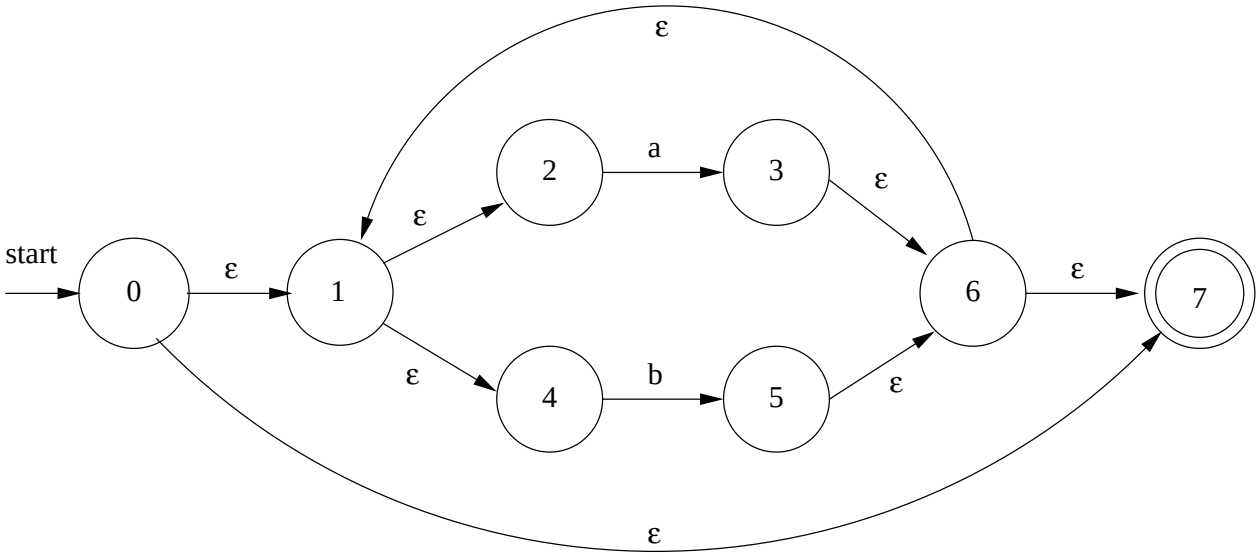
Cominciamo a considerare i nodi r_1 ed r_2. Per questi costruiamo i seguenti automi:



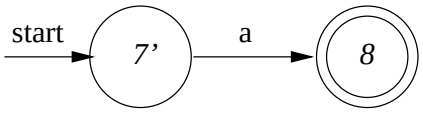
Adesso possiamo combinare $N(r_1)$ *ed* $N(r_2)$ *per ottenere* $N(r_3) = N(r_1|r_2) = N(a|b)$*:*



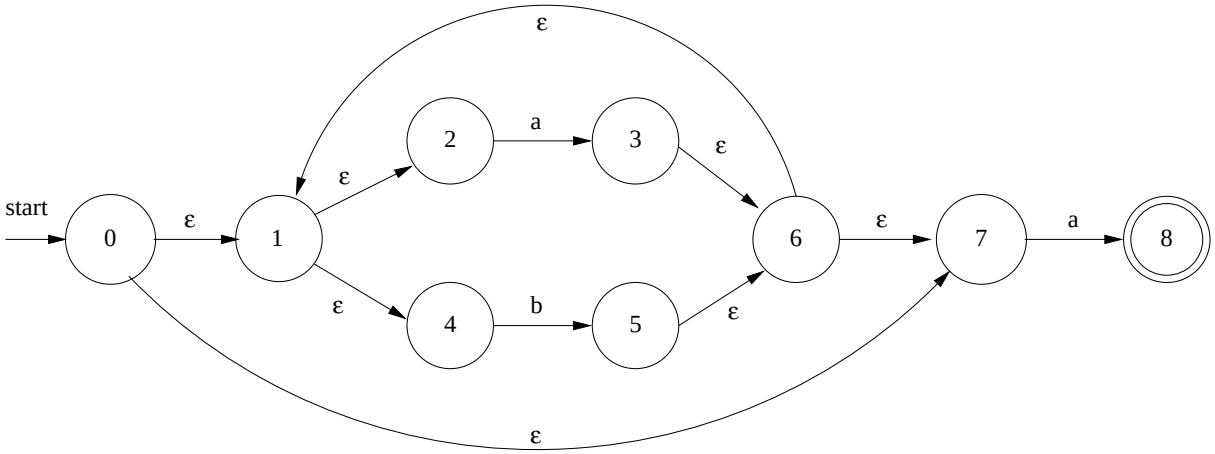
L'NFA per (r_3) *è lo stesso, abbiamo detto, di quello di* r_3*. Quindi andiamo avanti a costruire* $N(r_5) = N((r_3)^*)$*:*



L'NFA per r_6 *è semplicemente:*



L'automa risultante per r_7 è:



Proseguendo così fino alla fine otterremo l'automa in Figura 2.8

2.6.2 Dalle definizioni dei pattern al riconoscimento dei token

Un analizzatore lessicale deve riconoscere i lexeme che si susseguono nel programma sorgente individuando quelli che fanno match con i pattern che sono stati specificati per i token del linguaggio. Questa operazione si chiama *pattern matching* e può essere realizzata in maniera semplice utilizzando le nozioni che abbiamo visto finora.

Poniamoci quindi il seguente problema. Supponiamo data la definizione di un certo numero di pattern $p_1, p_2, \ldots p_n$ dove, per ogni pattern p_i, è specificata anche una sequenza di azioni a_i da effettuare:

$$\begin{array}{ll} p_1 & \{a_1\} \\ p_2 & \{a_2\} \\ \cdots & \\ p_n & \{a_n\} \end{array}$$

I pattern p_i, in generale, sono espressioni regolari. Ovviamente, per comodità, si permetterà di utilizzare all'interno dei pattern anche dei nomi definiti con una definizione regolare. Una sequenza di pattern e di azioni di questo genere è quello che tipicamente si dà in pasto ad un generatore automatico di analizzatori lessicali.

Quello che ci si aspetta è di ottenere un programma che legge una sequenza di caratteri e implementa un ciclo in cui, ad ogni passo, viene individuato nell'input il lexeme *più lungo* che fa match con uno dei pattern p_i.

Nel caso che una stessa sequenza di caratteri faccia match con due pattern contemporaneamente il programma sceglierà il pattern che sta più in alto nella definizione (ad esempio p_2 sta più in alto di p_3). Una volta deciso quale pattern p_i è stato riconosciuto il programma provvederà ad effettuare le operazioni a_i ad esso associate. Fatto questo il programma deve continuare il ciclo cercando un altro pattern a partire dal carattere di input successivo all'ultimo carattere dell'ultimo lexeme riconosciuto.

Per realizzare questo programma bisogna opportunamente utilizzare le costruzioni viste. Innanzitutto si noti che un NFA o un DFA, per definizione, riconosce una sola stringa per volta e non continua, una volta riconosciuta una stringa, a cercare la successiva. Inoltre un automa potrebbe fermarsi al primo stato finale che raggiunge non controllando se, andando avanti, potrebbe riconoscere una stringa più lunga. Per ovviare a questi problemi basterà utilizzare in maniera furba gli automi.

Il primo passo per la costruzione di un algoritmo di pattern matching con le caratteristiche che vogliamo è quello di convertire le espressioni regolari p_i (sostituendo opportunamente eventuali nomi definiti con una definizione regolare) in NFA utilizzando la costruzione di Thompson (Sezione 2.6.1). Otterremo così degli NFA $N(p_1), N(p_2), \ldots N(p_n)$ che riconoscono singolarmente il linguaggio formato da tutti i lexeme di ogni pattern. A questo punto costruiamo l'NFA mostrato in Figura 2.13. Questo NFA ha un nuovo stato iniziale ed esattamente n stati finali, uno per ogni pattern.

Per semplicità, descriviamo a parole la struttura dell'algoritmo di pattern matching basato su un NFA di questo tipo.

All'inizio di ogni iterazione del ciclo più esterno facciamo andare l'automa di Figura 2.13 sull'input attuale (il nondeterminismo può essere simulato semplicemente mantenendo ad ogni istante un insieme di stati attuali, come viene fatto nella costruzione dei sottoinsiemi) fino a quando non arriva ad uno stato finale.

Manteniamo in una variabile `p` un puntatore a questo punto dell'input e in una variabile `npattern` il numero corrispondente alla posizione nella lista, data all'inizio, del pattern che è stato riconosciuto. Per questo basta guardare lo stato finale che abbiamo incontrato (ricordiamo che esiste uno ed

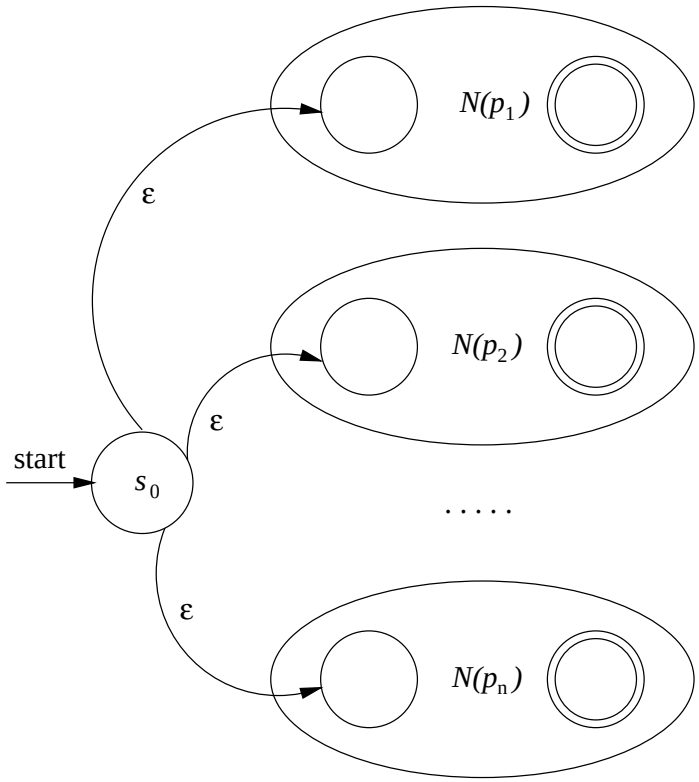


Figura 2.13: NFA usato per il pattern matching.

un solo stato finale per ogni pattern). Se abbiamo incontrato più stati finali scegliamo quello corrispondente al pattern che sta più in alto nella lista.

Facciamo ripartire l'automa dallo stesso insieme di stati in cui si trovava e dal carattere successivo a quello del lexeme individuato. Se l'automa incontra un nuovo pattern (va a finire in uno o più stati finali) aggiorniamo le variabili che mantenevano il puntatore all'input e il pattern riconosciuto con questi ultimi valori trovati.

L'automa deve continuare ad andare avanti fino a che non *termina*, cioè fino a che non può fare nessuna mossa (o perché è finito l'input o perché non può fare più nessuna transizione).

A questo punto il programma annuncia che è stato riconosciuto il pattern corrispondente all'ultimo pattern riconosciuto (che si è mantenuto in `npattern`) e che il lexeme corrispondente è quello che va dal primo carattere di input considerato nell'attuale passo del ciclo fino al carattere puntato da `p`.

Il programma passa poi ad eseguire le azioni associate al pattern riconosciuto (in genere sono delle sequenze di istruzioni scritte nello stesso

linguaggio di programmazione in cui viene generato l'analizzatore lessicale).

Il ciclo ricomincia facendo ripartire l'automa dallo stato iniziale e sull'input il cui primo carattere è quello successivo al lexeme riconosciuto (il carattere successivo a quello puntato da `p`).

Nel caso in cui l'automa termini senza avere riconosciuto nessun pattern, l'analizzatore lessicale deve segnalare, nei modi previsti, una situazione di errore.

Esempio 2.16 *Vediamo un esempio del procedimento appena descritto. Consideriamo la definizione dei seguenti tre pattern (per semplicità di presentazione non associamo azioni):*

$$\begin{array}{ll} a & \{\} \\ abb & \{\} \\ a^*b^+ & \{\} \end{array}$$

Innanzitutto dobbiamo costruire gli NFA relativi ad ognuno e poi metterli tutti insieme in un unico NFA come abbiamo mostrato in Figura 2.13. Per semplicità semplifichiamo l'NFA per il terzo pattern (quello risultante dalla costruzione di Thompson contiene più stati ed ϵ-transizioni) e otteniamo l'automa in Figura 2.14.

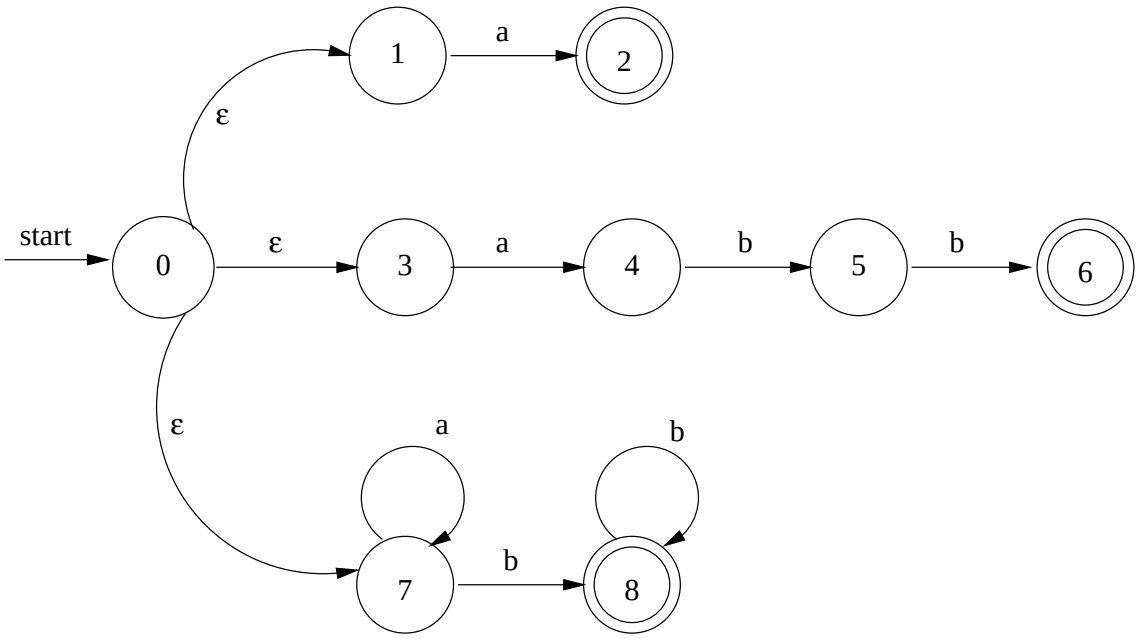


Figura 2.14: Un NFA che per il riconoscimento di tre pattern.

Proviamo ad applicare l'algoritmo di pattern matching descritto alla stringa di input aaba. In Figura 2.15 sono mostrati i momenti salienti della simulazione dell'automa e dell'algoritmo di riconoscimento dei pattern. Lo stato iniziale corrisponde all'insieme di stati ϵ*-closure*$(0) = \{0, 1, 3, 7\}$*. Leggendo una a (la prima dell'input) l'automa transisce nell'insieme di stati* ϵ*-closure*$(move(\{0, 1, 3, 7\}, a)) = \{2, 3, 7\}$ *come mostrato in figura. A questo punto ci accorgiamo che* 2 *è stato finale e che corrisponde al riconoscimento del primo pattern a. Come previsto dall'algoritmo andiamo avanti e memorizziamo questo risultato parziale, cioè assegniamo a* `npattern` *il valore 1 e facciamo puntare* `p` *alla prima a dell'input (in figura questo è indicato dal* p_1 *che si trova dopo la lettura della prima a).*

Dopo la lettura della seconda a dell'input l'automa si trova nell'insieme di stati $\{7\}$*, che non contiene stati finali. Andiamo quindi avanti e leggiamo il terzo simbolo dell'input: b. A questo punto l'automa si trova nell'insieme di stati* $\{8\}$ *che contiene lo stato finale relativo al terzo pattern. Come previsto dall'algoritmo aggiorniamo i nostri risultati parziali e memorizziamo che è stato riconosciuto il pattern* a^*b^+ *al terzo simbolo dell'input. Ma ancora dobbiamo andare avanti fino alla terminazione dell'automa.*

Leggendo il quarto simbolo a l'automa non può fare nessuna mossa perché dallo stato 8 *non ci sono transizioni uscenti etichettate con questo simbolo. Pertanto l'automa termina. L'algoritmo annuncia che il terzo pattern è stato riconosciuto e che il lexeme corrispondente è aab. Se ci fossero azioni associate al terzo pattern, questo sarebbe il momento di eseguirle.*

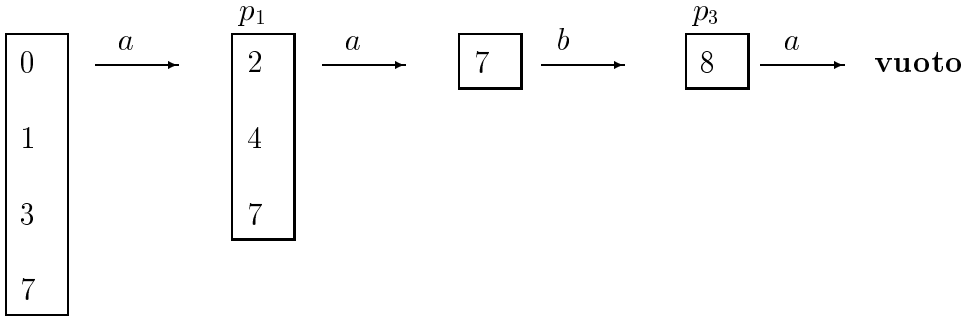


Figura 2.15: Sequenza di insiemi di stati attraversati leggendo l'input *aaba*.

Non essendo finito l'input, l'algoritmo riparte dallo stato iniziale per analizzare i caratteri che seguono il lexeme riconosciuto. In questo caso l'input ancora da analizzare è solo a (l'ultimo simbolo della stringa iniziale). Dallo stato iniziale arriviamo di nuovo all'insieme di stati $\{2, 4, 7\}$ *in cui* 2 *è lo stato finale associato al primo pattern. Memorizziamo questo risultato parziale ed andiamo avanti.*

L'automa termina perché non ci sono più simboli da analizzare e quindi l'algoritmo annuncia il riconoscimento del primo pattern con il lexeme a (che è anche l'unico possibile) dopodiché termina definitivamente.

Notiamo che, come volevamo, l'algoritmo riconosce i pattern che hanno il lexeme più lungo nell'input attuale.

In generale un analizzatore lessicale efficiente non utilizza un automa non deterministico per fare il pattern matching. Tramite l'algoritmo di costruzione dei sottoinsiemi si può ottenere un automa deterministico equivalente a quello usato per il pattern matching con l'algoritmo che abbiamo descritto. Inoltre questo automa può essere ancora migliorato, preservando il linguaggio accettato, con l'algoritmo di minimizzazione introdotto in Sezione 2.5.4.

Notiamo che queste trasformazioni non fanno altro che raggruppare gli stati dell'automa non deterministico di partenza. Pertanto è possibile eseguire lo stesso algoritmo per il pattern matching utilizzando l'automa deterministico minimo. In questo caso si avrà che ad ogni passo l'automa si troverà in un solo stato che però rappresenta un certo insieme di stati dell'automa di partenza. Per quanto riguarda quindi l'individuazione dei pattern attraverso gli stati finali raggiunti basta semplicemente controllare l'insieme di stati che l'unico stato di ogni passo rappresenta e agire come nel caso dell'automa non deterministico che abbiamo visto.

2.6.3 Il generatore automatico `Lex`

In questa sezione parliamo di un particolare generatore di analizzatori lessicali molto diffuso sui sistemi Unix che si chiama `Lex`. `Lex` riceve in input un file di testo la cui struttura analizzeremo in dettaglio e restituisce come output un programma sorgente scritto in linguaggio C che realizza il pattern matching (come lo abbiamo visto nella sezione precedente) dei pattern specificati nel file di input.

In Figura 2.16 sono sintetizzati i passi per creare un analizzatore lessicale scritto in C con `Lex`. Il file di input viene processato e viene generato un programma C che si chiama `lex.yy.c`. Una volta compilato, il programma riceve una sequenza di caratteri sullo standard input e restituisce sullo standard output la sequenza di token trovati.

Di seguito riportiamo un file di input che specifica i pattern di alcuni token del linguaggio Pascal.

```
%{ /* definizione delle costanti numeriche che
      rappresentano i vari token */
```

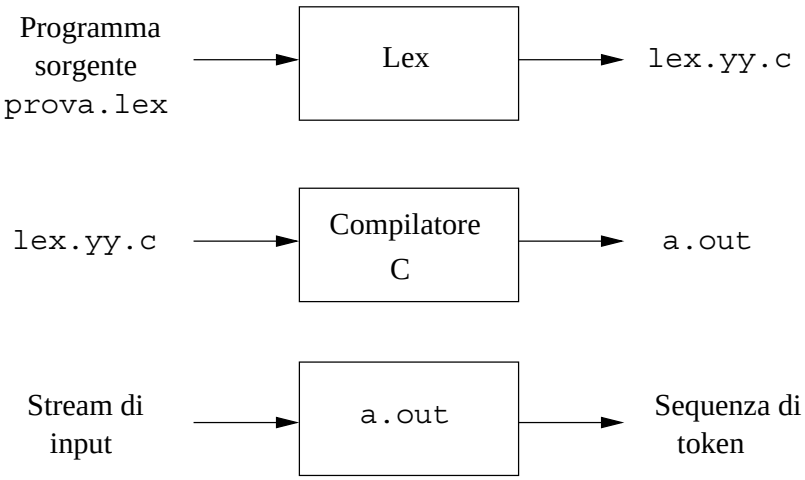


Figura 2.16: Passi per la creazione di un analizzatore lessicale con `Lex`.

```
#define IF 0
#define THEN 1
#define ELSE 2
#define ID 3
#define NUMBER 4

#define RELOP 5 // valore del token
#define LT 50   // valori dell'attributo per il token 5
#define LE 51   // ...
#define EQ 52
#define NE 53
#define GT 54
#define GE 55
%}

%%
/* Definizioni Regolari */
delim         [ \t\n]
ws            {delim}+    //white spaces
letter        [A-Za-z]
digit         [0-9]
id            {letter}({letter}|{digit})*
number        {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?

%%
```

```
/* Pattern e azioni associate */

{ws}          {/* Nessuna azione e nessun ritorno */}
if            {return(IF);}
then          {return(THEN);}
else          {return(ELSE);}
{id}          {yylval = install_id(); return(ID);}
{number}      {yylval = install_num(); return(NUMBER);}
''<''         {yylval = LT; return(RELOP);}
''<=''        {yylval = LE; return(RELOP);}
''=''         {yylval = EQ; return(RELOP);}
''<>''        {yylval = NE; return(RELOP);}
''>''         {yylval = GT; return(RELOP);}
''>=''        {yylval = GE; return(RELOP);}

%%
/* Funzioni da definire
   (qui va inserita anche l'implementazione) */
int install_id() {
     /* Installa il lexeme nella tabella dei simboli (se non
        c'e' gia'). Il primo carattere del lexeme e' puntato
        dalla variabile di tipo char* yytext, mentre la
        lunghezza del lexeme e' contenuta nella variabile
        yyleng. La funzione restituisce l'intero che e' il
        puntatore alla riga della tabella dei simboli che
        contiene le informazioni sul token appena installato
     */
}

int install_num () {
    /* Funzione simile, solo che il lexeme che viene installato
       nella tabella dei simboli e' un numero */
}
```

Come si vede, la prima parte di un file di input per `Lex` è racchiusa tra le due sequenze di caratteri %{ e %}. In questa sezione vanno inserite le definizioni di costanti e/o macro personalizzate del programma C che verrà generato. Questa parte di testo verrà copiata senza nessuna modifica all'inizio del programma C che verrà generato. Nell'esempio abbiamo inserito le costanti intere che rappresentano i vari token che vogliamo riconoscere. Si noti che per gli operatori binari di relazione fra interi abbiamo scelto

di creare un'unica categoria lessicale individuata dal token `RELOP`. Sarà poi l'attributo del token, anch'esso un intero, a specificare quale lexeme è stato effettivamente trovato.

Dopo la prima sezione può essere inserita una definizione regolare. `Lex` permette di specificare le espressioni regolari usando diversi operatori oltre a quelli che abbiamo già visto. Tutti questi operatori sono riportati nella seguente tabella[3]:

ESPRESSIONE	DESCRIZIONE	ESEMPIO
c	Un qualsiasi carattere non operatore c	`a`
$\backslash c$	Letteralmente il carattere c	`\*`
"s"	Letteralmente la stringa s	`"**"`
.	Un carattere qualsiasi tranne il *newline*	`a.*b`
^	Inizio della linea	`^ abc`
$	Fine della linea	`abc$`
$[s]$	Un qualsiasi carattere in s	`[abc]`
$[\hat{}\ s]$	Un qualsiasi carattere *non* in s	`[^ abc]`
r^*	Zero o più occorrenze di r	`a*`
r^+	Una o più occorrenze di r	`a+`
$r?$	Zero o una occorrenza di r	`a?`
$r\{m,n\}$	Da m a n occorrenze di r	`a{1,5}`
r_1r_2	Concatenazione	`ab`
$r_1\|r_2$	Or	`a|b`
(r)	Parentesi	`(a|b)`
r_1/r_2	r_1 se seguita da r_2	`abc/123`

Quando all'interno di una espressione regolare vogliamo utilizzare un nome definito precedentemente in una definizione regolare, il nome va messo tra parentesi graffe.

Nell'esempio abbiamo definito gli spazi bianchi, le costanti numeriche e gli identificatori.

Dopo la sequenza di caratteri `%%` inizia la sezione di definizione dei pattern veri e propri con le relative azioni da intraprendere una volta riconosciuti. L'ordine è importante in quanto a parità di lunghezza del lexeme prevale il pattern che viene prima nella lista di questa sezione. Notiamo che se viene riconosciuto il pattern degli spazi bianchi (sequenze di spazi, caratteri di tabulazione (`\t`) o caratteri *newline* (`\n`), non viene intrapresa nessuna

[3]In ogni caso si ha che tutti gli operatori sono riconducibili a quelli base, cioè concatenazione, or e stella di Kleene.

azione. In particolare il programma continuerà a cercare il prossimo pattern senza ritornare al chiamante.

Al pattern `if` abbiamo associato come azioni quella di ritornare al chiamante il token corrispondente alla parola chiave `if`. Notiamo che il chiamante è di solito il parser che richiede un token alla volta. Tuttavia nel caso degli spazi bianchi non è necessario restituire alcun token. Inoltre la parola chiave `if` è messa prima degli identificatori poiché alla visione del lexeme `if`, che fa match anche con il pattern degli identificatori, deve prevalere il pattern che identifica il lexeme come parola chiave.

Per il pattern {`id`} è specificata anche una chiamata di funzione che restituisce un intero assegnato alla variabile `yylval`. Questa variabile è una variabile intera globale del programma che contiene per convenzione, ad ogni nuovo riconoscimento di un token, il valore dell'attributo ad esso associato. Abbiamo visto che nel caso degli identificatori questo valore è spesso un intero che identifica l'entrata nella tabella dei simboli per il lexeme dell'identificatore riconosciuto. Dell'inserimento in questa tabella e della restituzione del valore si occupa la funzione `install_id()`. Tale funzione va scritta nella terza sezione del file di input di `Lex` (quella che segue le successive `%%`). In questo esempio l'implementazione viene lasciata vuota perché bisognerebbe definire anche una struttura dati per la tabella dei simboli con i relativi algoritmi di inserimento, ricerca, aggiornamento. In sede di una reale implementazione di un compilatore questa parte verrà organizzata in accordo con l'implementazione delle altre fasi.

Per il pattern {`number`} è prevista la chiamata ad una funzione analoga alla precedente ma che si dovrà occupare del riconoscimento del tipo di costante numerica (intera o floating point) e di altre cose strettamente relative ai numeri, oltre che all'inserimento del lexeme nella tabella dei simboli.

Per quanto riguarda l'implementazione di queste funzioni è utile sapere che `Lex` mette a disposizione nella variabile globale `yytext` un puntatore a carattere (che in C è lo stesso che dire una stringa) che punta all'inizio del lexeme del pattern corrente riconosciuto. Per completare l'informazione la variabile globale `yyleng` contiene la lunghezza del lexeme cosicchè quest'ultimo possa essere identificato univocamente e trattato adeguatamente.

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.

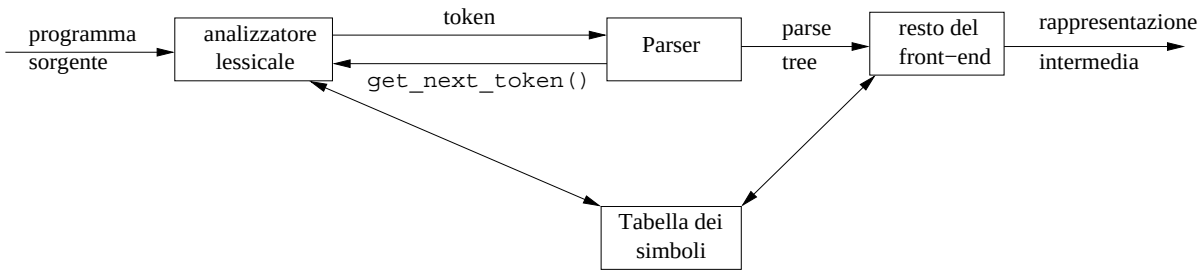


Figura 3.1: Il ruolo del parser.

- 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.*

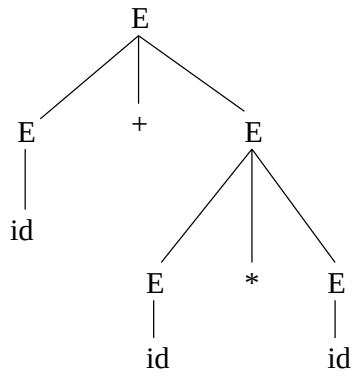


Figura 3.2: Un albero di derivazione per **id** + **id** ∗ **id**.

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.

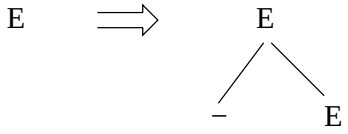
Una derivazione lunga 0 passi è la sequenza composta solo dal simbolo S.

Per indicare una derivazione di zero o più passi da S ad una certa stringa α scriviamo $S \overset{*}{\Rightarrow}_G \alpha$, mentre $S \overset{+}{\Rightarrow}_G \alpha$ 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 $L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow}_G w\}$. 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 $S \overset{*}{\Rightarrow} \beta$) 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: $\beta = \alpha A \gamma$

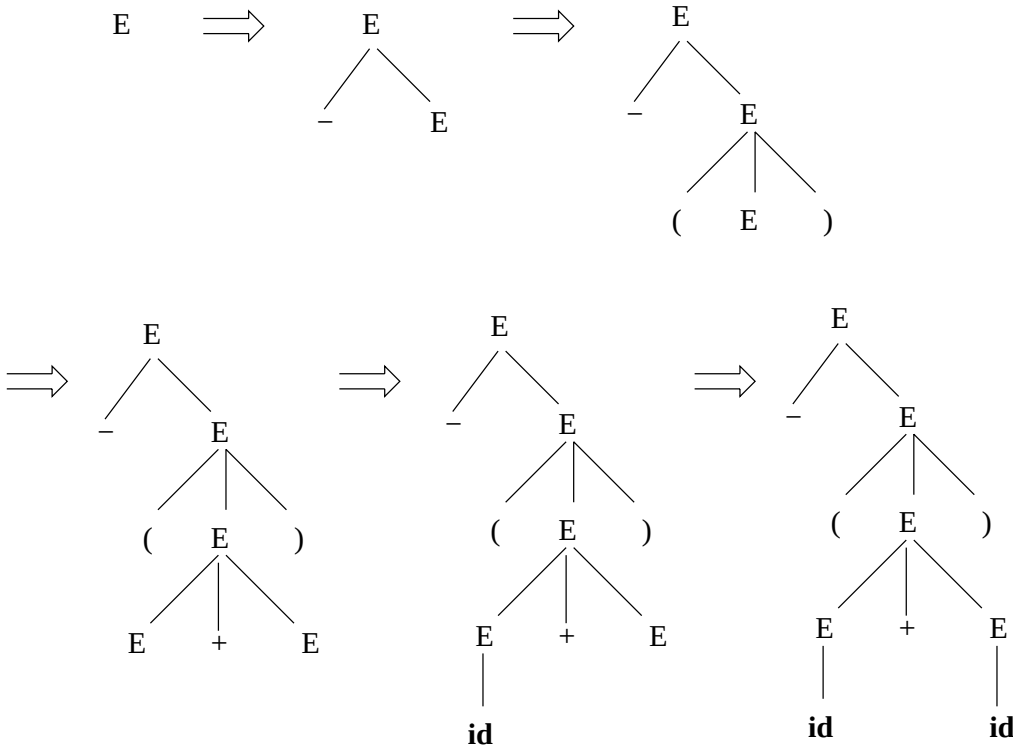


Figura 3.3: Da una derivazione a un albero di derivazione

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: $S \overset{*}{\Rightarrow}_{rm} \alpha A x \Rightarrow_{rm} \alpha \delta x$.

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):

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

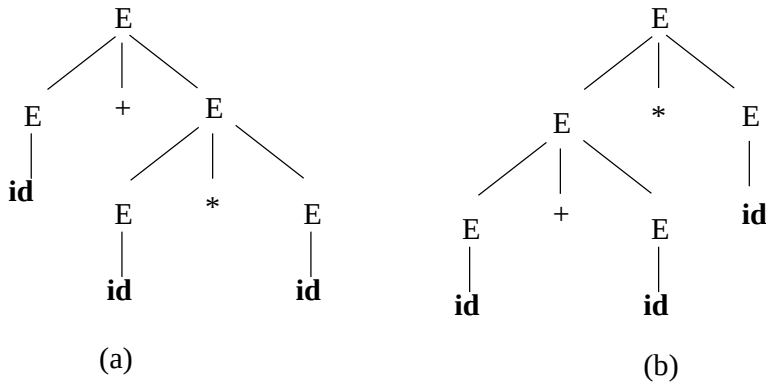


Figura 3.4: Due alberi di derivazione diversi per la stringa $\mathbf{id} + \mathbf{id} * \mathbf{id}$.

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

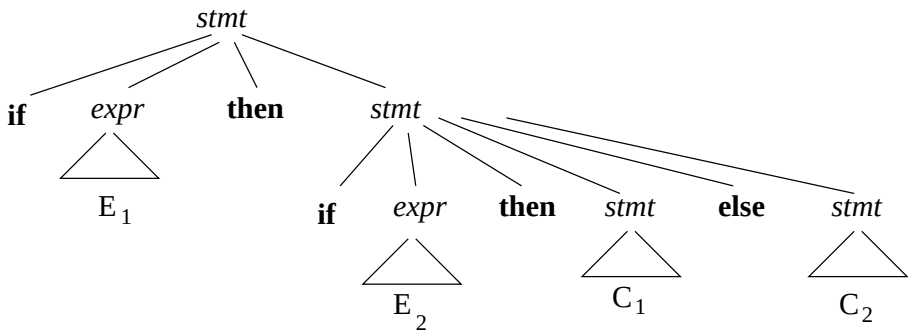


Figura 3.5: Un albero di derivazione per la stringa

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.

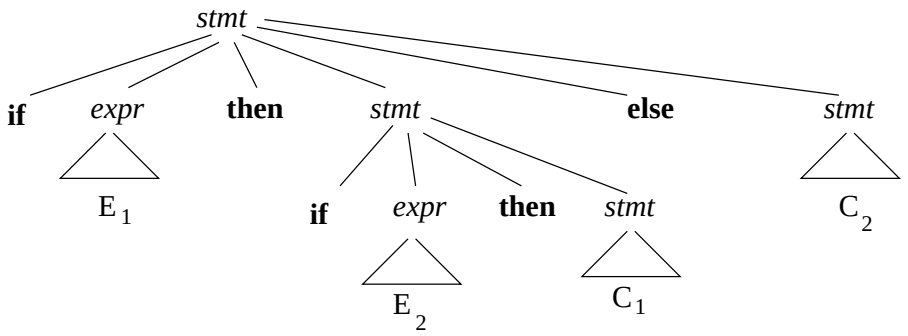


Figura 3.6: Un altro albero di derivazione per la stessa stringa

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**.

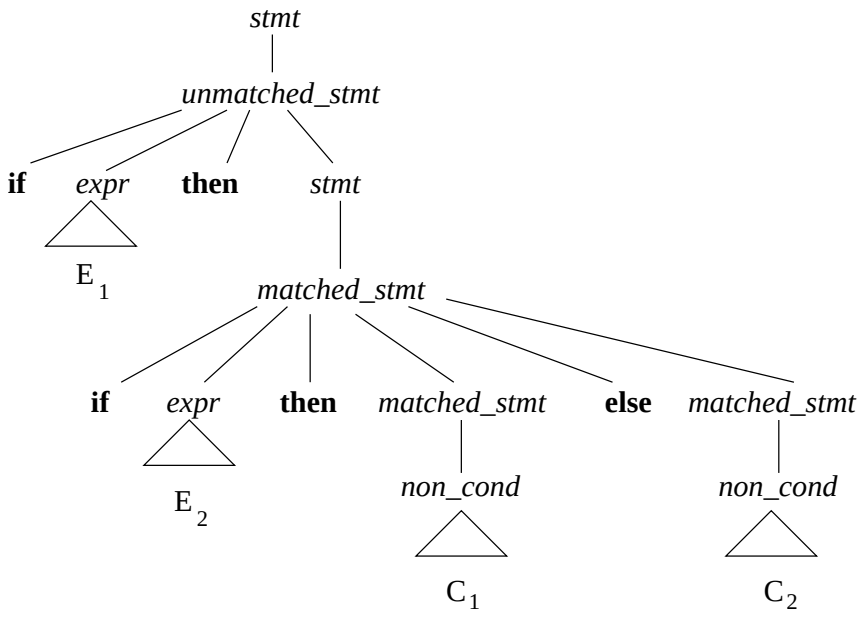


Figura 3.7: L'unico albero di derivazione per la stringa con la nuova grammatica.

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

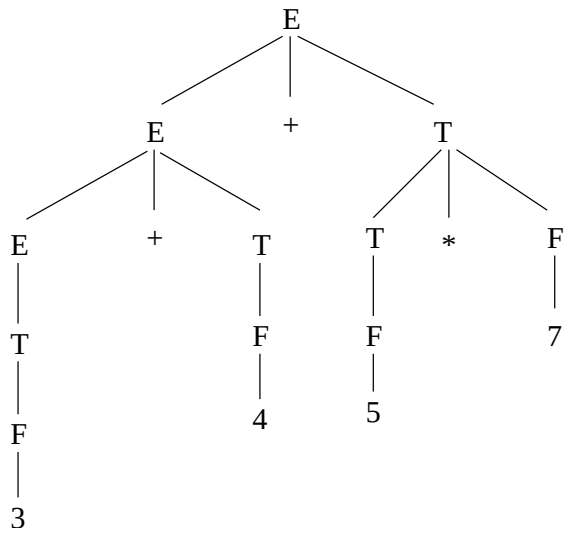


Figura 3.8: Un albero di derivazione per l'espressione $3 + 4 + 5 * 7$.

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

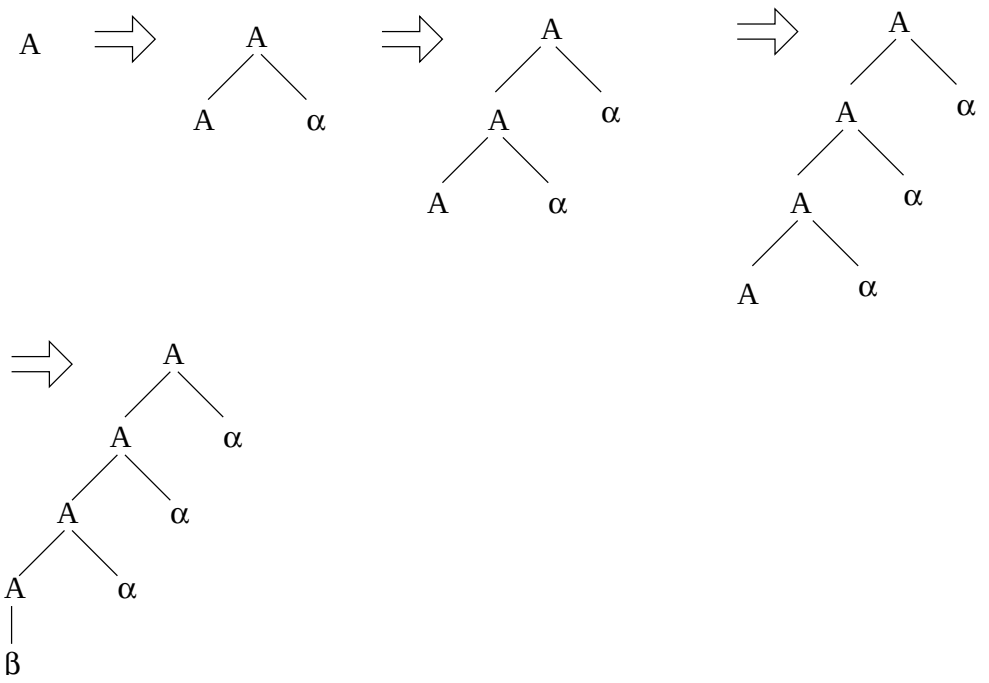


Figura 3.9: Generazione di una stringa con produzioni ricorsive a sinistra.

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-

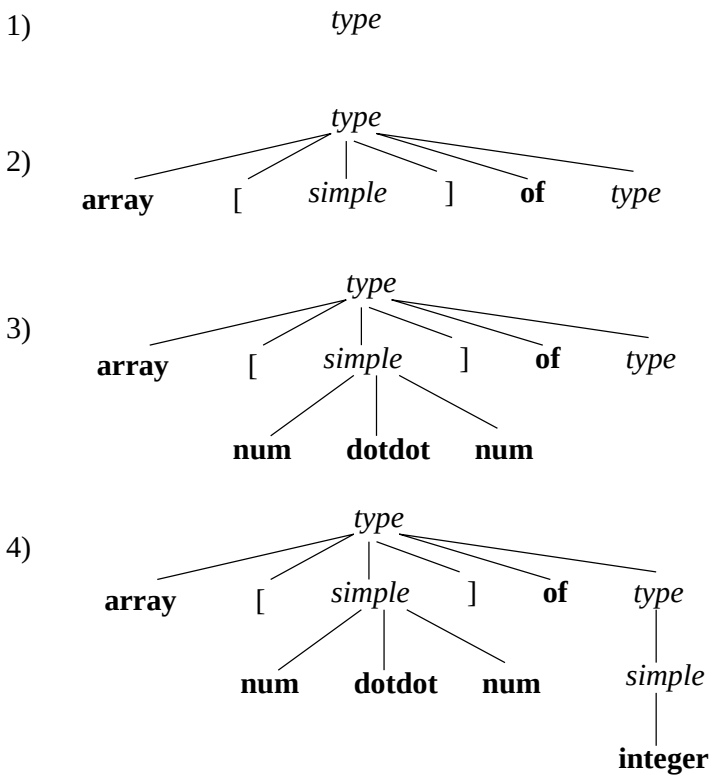


Figura 3.11: Passi nella costruzione top-down del parse tree.

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* $\rightarrow$ **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 $A \rightarrow \alpha_1 \mid \cdots \mid \alpha_k$ 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:

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

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

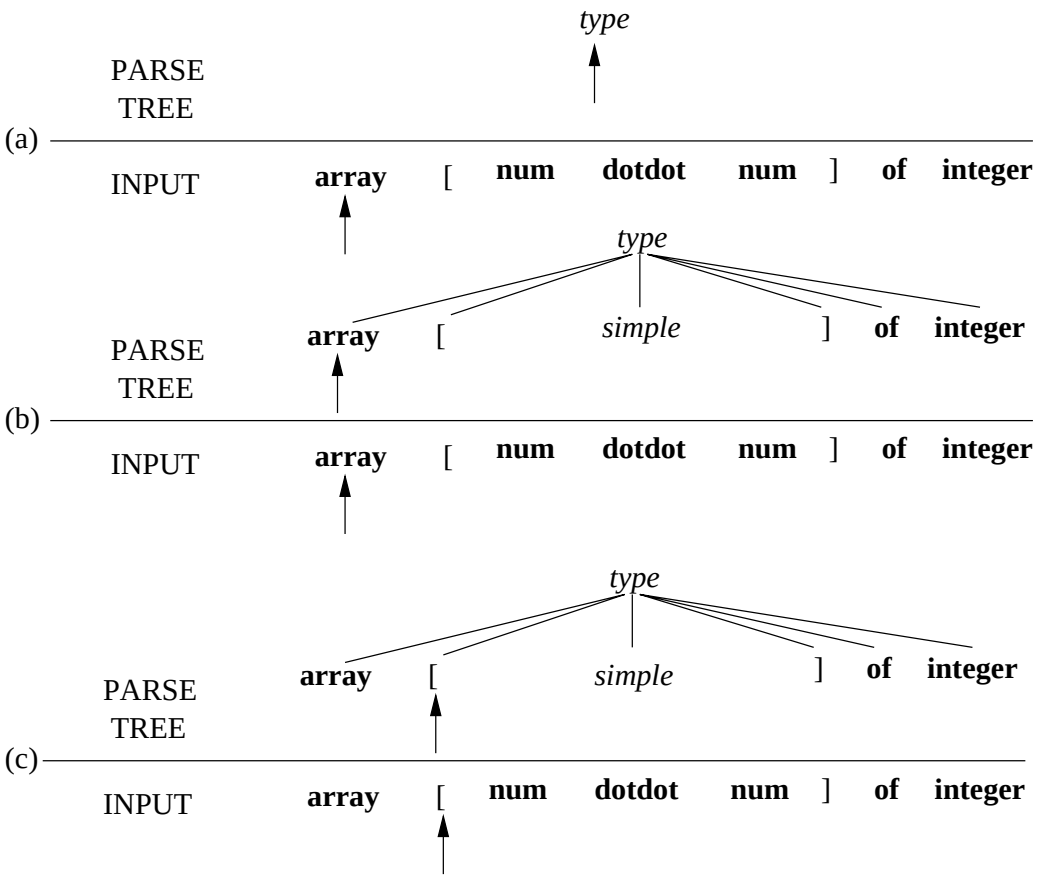


Figura 3.12: Parsing top-down durante la scansione dell'input da sinistra a destra.

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-

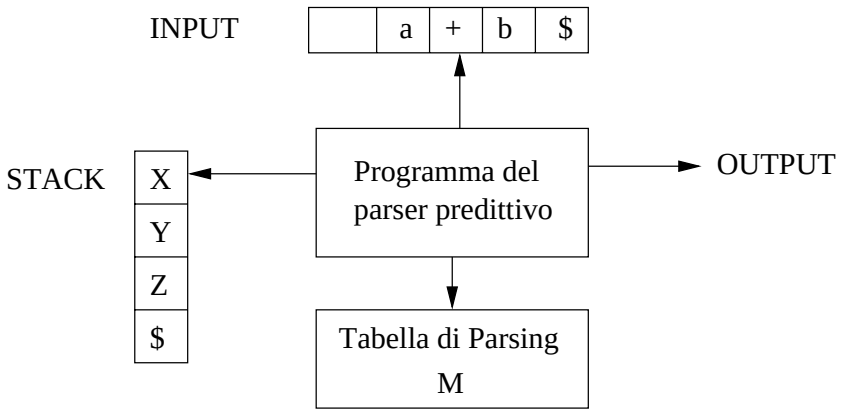


Figura 3.13: Struttura di un parser predittivo non ricorsivo.

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:*

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

[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 $X = top(\text{stack})$ e sia a il simbolo puntato da ip;
    if X is terminale oppure \$ **then**
        if $X = a$ **then**
            $pop(\text{stack})$ 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**
            $pop(\text{stack})$;
            $push(Y_k, \text{stack})$;
            $push(Y_{k-1}, \text{stack})$;
            $\vdots$
            $push(Y_1, \text{stack})$;
            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	$F \to \mathbf{id}$			$F \to (E)$		

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 \$ $\in$ 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.

Questo processo può equivalentemente essere pensato come la riduzione di una stringa w di simboli terminali in input al simbolo iniziale della grammatica. Ad ogni passo di questa riduzione una sottostringa, della forma sentenziale corrente, che fa match con la parte destra di una produzione della grammatica viene riscritta nel simbolo non terminale a sinistra della produzione in questione. Se ad ogni passo di riduzione la sottostringa viene scelta correttamente allora la sequenza di riduzioni fatte termina con una forma sentenziale che contiene solo il simbolo iniziale della grammatica e tutta la sequenza di riduzione corrisponde ad una derivazione rightmost della stringa w a partire dal simbolo iniziale.

Esempio 3.19 *Chiariamo meglio questo processo con l'aiuto di un esempio. Consideriamo la seguente grammatica:*

$$\begin{array}{lcl} S & \rightarrow & aABe \\ A & \rightarrow & Abc \mid b \\ B & \rightarrow & d \end{array}$$

e consideriamo la stringa di input $w = abbcde$. Questa stringa può essere ridotta a S tramite i seguenti passi di riduzione:

$$\begin{array}{lcl} a\underline{b}bcde & A & \rightarrow b \\ a\underline{Abc}de & A & \rightarrow Abc \\ aA\underline{d}e & B & \rightarrow d \\ \underline{aABe} & S & \rightarrow aABe \\ S & & \end{array}$$

dove la parte sottolineata è la sottostringa uguale alla parte destra della produzione, indicata a fianco, che utilizziamo nella riduzione.

Il procedimento generale è quello di cercare, ad ogni passo, una sottostringa uguale alla parte destra di una produzione della grammatica. C'è la possibilità che ne esista più di una: ad esempio al primo passo entrambe le b in seconda e terza posizione e la d in quinta posizione possono essere ridotte. Nell'esempio abbiamo scelto la b in seconda posizione e l'abbiamo ridotta ad A. Il punto chiave è fare la scelta giusta.

Al secondo passo dobbiamo effettuare un'altra scelta poichè b, Abc e d sono tutte passibili di riduzione. Andando avanti, e facendo le scelte giuste, siamo riusciti a ricostruire la seguente derivazione rightmost per la stringa data:

$$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$$

3.5.1 Handle

Informalmente una *handle* (maniglia) di una stringa è una sottostringa che è esattamente uguale alla parte destra di una certa produzione e la cui riduzione al simbolo a sinistra della produzione rappresenta un passo lungo una derivazione rightmost rovesciata.

È importante far notare subito che in molti casi la sottostringa più a sinistra che è uguale alla parte destra di una certa produzione non è una handle perché una riduzione tramite questa produzione produce una stringa dalla quale non esiste una sequenza di riduzioni che portano al simbolo iniziale della grammatica. Per questo motivo è importante dare una definizione precisa di "handle".

Sia γ una forma sentenziale destra, cioè esiste una derivazione del tipo $S \stackrel{*}{\Rightarrow}_{rm} \gamma$. Diciamo che una handle di γ è formata da una produzione $A \to \beta$ e da una posizione in γ dove appare la stringa β tali che se quest'ultima viene rimpiazzata con A allora il risultato è la forma sentenziale destra precedente a γ in una derivazione rightmost di γ stessa.

In altre parole, se $S \stackrel{*}{\Rightarrow}_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$ allora la produzione $A \to \beta$ e la posizione successiva all'occorrenza di α è una handle per la forma sentenziale destra $\alpha\beta w$. La stringa w a destra della handle contiene solo simboli terminali.

Finora abbiamo detto "una handle" invece che "la handle" perché se la grammatica che consideriamo è ambigua allora possono esistere più derivazioni rightmost per la stessa stringa e, quindi, diverse handle all'interno della stessa forma sentenziale destra. Tuttavia se la grammatica non è ambigua sappiamo che per ogni forma sentenziale destra esiste una sola derivazione rightmost che la genera a partire dal simbolo iniziale e, quindi, ad ogni passo di riduzione la handle è unica.

Nell'esempio precedente *abbcde* è una forma sentenziale destra la cui handle è la produzione $A \to b$ e la posizione 2. Allo stesso modo, *aAbcde* ha come handle la produzione $A \to Abc$ e la posizione 2. Alcune volte possiamo dire direttamente "la sottostringa β è una handle di $\alpha\beta w$" se la produzione $A \to \beta$ e la posizione di β sono chiare dal contesto.

Figura 3.18 visualizza la handle β in un albero di derivazione (incompleto) di $\alpha\beta w$. La handle è il sottoalbero più a sinistra che è completo e che consiste di un nodo interno e di tutti i suoi figli. Graficamente, A è il nodo interno più in basso e a sinistra che ha tutti i figli nell'albero. Ridurre β ad A in $\alpha\beta w$ può essere pensato come ad una "potatura" dell'albero, cioè alla rimozione di tutti i figli di A.

Esempio 3.20 *Consideriamo la seguente grammatica:*

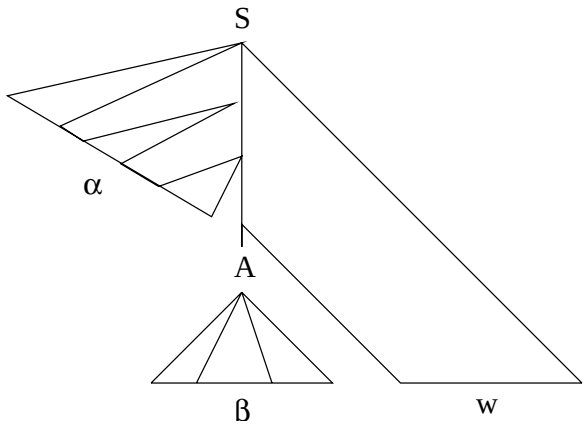


Figura 3.18: La handle $A \to \beta$ nel parse tree per $\alpha\beta w$.

$$
\begin{array}{lllll}
(1) & E & \to & E + E \\
(2) & E & \to & E * E \\
(3) & E & \to & (E) \\
(4) & E & \to & \mathbf{id}
\end{array}
$$

e la seguente derivazione rightmost:

$$
\begin{array}{lll}
E & \Rightarrow_{rm} & \underline{E + E} \\
 & \Rightarrow_{rm} & E + \underline{E * E} \\
 & \Rightarrow_{rm} & E + E * \underline{\mathbf{id}_3} \\
 & \Rightarrow_{rm} & E + \underline{\mathbf{id}_2} * \mathbf{id}_3 \\
 & \Rightarrow_{rm} & \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3
\end{array}
$$

Abbiamo numerato gli **id** *per convenienza di notazione e la parte di stringa sottolineata in ogni forma sentenziale destra è una handle. Ad esempio* $\mathbf{id}_1$ *è una handle in* $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$ *poiché* **id** *è la parte destra della produzione* $E \to \mathbf{id}$ *e rimpiazzando* $\mathbf{id}_1$ *con* E *otteniamo la forma sentenziale precedente* $E + \mathbf{id}_2 * \mathbf{id}_3$*. Notiamo, come abbiamo già detto, che in ogni caso la stringa che appare a destra della handle è sempre formata solo da simboli terminali.*

Sappiamo già che questa grammatica è ambigua. La seguente è un'altra derivazione rightmost, con relativa indicazione delle handle, per la stringa considerata prima:

$$
\begin{array}{lll}
E & \Rightarrow_{rm} & \underline{E * E} \\
 & \Rightarrow_{rm} & E * \underline{\mathbf{id}_3} \\
 & \Rightarrow_{rm} & \underline{E + E} * \mathbf{id}_3 \\
 & \Rightarrow_{rm} & E + \underline{\mathbf{id}_2} * \mathbf{id}_3 \\
 & \Rightarrow_{rm} & \underline{\mathbf{id}_1} + \mathbf{id}_2 * \mathbf{id}_3
\end{array}
$$

Consideriamo la forma sentenziale destra $E + E + \mathbf{id}_3$. *In questa ultima derivazione* $E + E$ *è una handle, ma anche* $\mathbf{id}_3$ *è una handle della stessa forma sentenziale considerando l'altra derivazione rightmost.*

3.5.2 Handle pruning

Possiamo pensare al processo di costruire una derivazione rightmost rovesciata in termini di handle pruning (potatura delle maniglie) dal parse tree della forma sentenziale destra che dobbiamo ridurre. Partiamo dalla stringa di terminali (token) w che vogliamo analizzare. Se w è una stringa generata dalla grammatica a cui facciamo riferimento allora $w = \gamma_n$ dove γ_n l'n-esima forma sentenziale destra di una qualche derivazione rightmost ancora sconosciuta:

$$S = \gamma_0 \Rightarrow_{rm} \gamma_1 \Rightarrow_{rm} \cdots \Rightarrow_{rm} \gamma_{n-1} \Rightarrow_{rm} \gamma_n = w$$

Per ricostruire questa derivazione in ordine inverso dobbiamo trovare la handle β_n in γ_n e rimpiazzare β_n con la parte sinistra di una produzione $A_n \to \beta_n$. Otterremo in questo modo γ_{n-1}. *Ancora non sappiamo come fare a trovare le maniglie, ma presto vedremo dei modi per farlo.*

Questo processo viene ripetuto fino a quando non risaliamo a $\gamma_0 = S$. A questo punto l'analisi termina con successo.

3.5.3 Shift-Reduce Parsing

Ci sono due problemi principali che vanno risolti durante il parsing tramite potatura delle maniglie:

1. Trovare la sottostringa β che deve essere ridotta
2. Scegliere quale produzione utilizzare nel caso ci siano più produzioni con la stessa parte destra β, ma con simbolo a sinistra diverso.

Prima di occuparci in dettaglio di questi problemi vediamo che tipo di strutture dati conviene utilizzare per il parsing *shift-reduce* (analisi impila-riduci). È utile utilizzare uno stack che possa contenere simboli della grammatica (sia terminali sia non) e un buffer di input che possa contenere la stringa w che deve essere utilizzata. Come nel caso del top-down parsing usiamo un simbolo speciale \$ per marcare il fondo dello stack e la fine della stringa di input.

Inizialmente lo stack deve essere vuoto e tutta la stringa deve essere nell'input:

STACK	INPUT
\$	w \$

L'analizzatore shift-reduce opera nel seguente modo: mette sullo stack zero o più simboli di input, prelevandoli dal buffer, fino a quando non riconosce che sulla testa dello stack c'è una handle β. A questo punto riduce la stringa β eliminandola dallo stack e mettendo al suo posto (in testa) il simbolo non terminale a sinistra della produzione della handle.

Questo processo viene iterato fino a che non viene incontrato un errore (sintattico) oppure si raggiunge la configurazione di accettazione:

STACK	INPUT
\$S	\$

Esempio 3.21 *Vediamo tutti i passi che dovrebbe fare un parser shift-reduce che analizza* $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$*, fa riferimento alla grammatica dell'esempio 3.20 e ricostruisce la prima derivazione rightmost presentata nell'esempio. Ovviamente esiste un'altra sequenza di passi che un parser shift-reduce può fare per analizzare questa stringa con la grammatica data, cioè quella corrispondente all'altra derivazione rightmost.*

	STACK	*INPUT*	*AZIONE*
(1)	\$	$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$ \$	*shift*
(2)	\$ $\mathbf{id}_1$	$+\mathbf{id}_2 * \mathbf{id}_3$ \$	*riduci con* $E \to \mathbf{id}$
(3)	\$ E	$+\mathbf{id}_2 * \mathbf{id}_3$ \$	*shift*
(4)	\$ $E+$	$\mathbf{id}_2 * \mathbf{id}_3$ \$	*shift*
(5)	\$ $E + \mathbf{id}_2$	$*\mathbf{id}_3$ \$	*riduci con* $E \to \mathbf{id}$
(6)	\$ $E + E$	$*\mathbf{id}_3$ \$	*shift*
(7)	\$ $E + E*$	$\mathbf{id}_3$ \$	*shift*
(8)	\$ $E + E * \mathbf{id}_3$	\$	*riduci con* $E \to \mathbf{id}$
(9)	\$ $E + E * E$	\$	*riduci con* $E \to E * E$
(10)	\$ $E + E$	\$	*riduci con* $E \to E + E$
(11)	\$ E	\$	*accetta*

Le operazioni principali sono shift e reduce, ma ce ne sono altre due possibili:

1. *accetta* in cui il parser si ferma e annuncia che il parsing ha avuto successo

2. *errore* in cui il parser ha trovato un errore sintattico. Si ferma e chiama una routine di gestione degli errori.