UNIVERSITÀ DEGLI STUDI DI PISA

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Specialistica in Informatica

Tesi di Laurea

Compiler Generator:
Considerazioni e Confronti

Candidato:

Gianfilippo Bortone

Relatore:

Prof. Marco Bellia

Controrelatore:

Prof. Andrea Maggiolo Schettini

Anno Accademico 2006/2007

Ai miei genitori Francesca e Donato

che mi hanno sostenuto ed incoraggiato

in tutti questi anni.

"Una parola non riuscirebbe a racchiudere

tutto ciò che è dentro di lui;

difficile, a volte, entrare nel suo particolarissimo mondo,

ma chiunque sia toccato dal suo affetto

non può rimanere impassibile

di fronte a quella miscela di stravaganza,

dolcezza e fantasia che è...

Gianfy B.! "

Alessandra e Francesca

A tutti i miei amici.

Indice

Capitolo 3: Parser generator YACC

Capitolo 4: Parser generator Bison

Capitolo 5: Parser generator Essence

Capitolo 6: Parser generator Coco/R

Introduzione

Il problema della compilazione può essere espresso come quello di tradurre una generica stringa in codice eseguibile. Esso si presenta come un caso particolare del problema fondamentale in informatica, che è quello di tradurre un linguaggio di ingresso in un linguaggio di uscita. La traduzione compiuta da un compilatore ha tale vincolo aggiuntivo: deve trasformare l'ingresso in una uscita equivalente.

Dopo aver definito e descritto in modo accurato il linguaggio, la grammatica, gli automi e i riconoscitori deterministici, il presente lavoro si incentra sul descrivere ed esaminare alcuni strumenti atti allo sviluppo di prototipi di traduttori, allo scopo di sottolineare le differenze di gestione e di analisi tra i vari prototipi presi in esame. Vengono messi in evidenza, inoltre, anche l'aspetto sintattico e l'aspetto semantico dei parsers esaminati.

Quello su cui la nostra tesi si è maggiormente concentrata è il parser generator Yacc per le grammatiche LALR(1), del quale sono stati messi in evidenza le caratteristiche principali e il modo in cui esso si serve delle grammatiche. Sono state quindi ricercate, negli strumenti successivi, le innovazioni rispetto a Yacc, che resta comunque, ancora oggi, uno degli strumenti più diffusi nel settore.

Oltre a Yacc sono stati, dunque, studiati altri tre parser generators: Bison per le grammatiche LALR(1), Essence, scritto nel linguaggio di

programmazione Scheme48, per le grammatiche LR(k) e SLR(k), e Coco/R per le grammatiche LL(1).

Questo lavoro di tesi è diviso nei seguenti capitoli:

Capitolo 1 : apre la tesi dando definizioni e strutture riguardanti il linguaggio, le stringhe, le grammatiche, la rappresentazione di tali grammatiche, gli automi a stati finiti deterministici e non deterministici.

Capitolo 2 : descrive i concetti teorici che sono alla base dei parsers presi in esame. Vengono, pertanto, dati definizioni e metodi di derivazione riguardanti grammatiche context-free e viene illustrato il modo in cui tali grammatiche sono analizzate.

Capitolo 3 : descrive il parser generator Yacc mettendo in evidenza come deve essere strutturata una grammatica Yacc e il modo in cui il parser lavora e risolve conflitti ed errori. Sono inoltre analizzati i file *y.output* e *y.tab.c* .

Capitolo 4 : descrive il parser generator Bison mettendo in evidenza come deve essere strutturato un file di input Bison, le dichiarazioni di cui il parser fa uso, i valori semantici e la gestione degli errori. Si discute, inoltre, la generalizzazione dei parsers LR.

Capitolo 5 : descrive il parser generator Essence mettendo in evidenza come deve essere strutturata una grammatica Essence, il modo in cui

il parser lavora, l'implementazione dello stream di input e il recupero degli errori.

Capitolo 6 : descrive il parser generator Coco/R mettendone in evidenza le caratteristiche e la struttura, e illustrando il modo in cui vengono risolti conflitti ed errori.

Capitolo 7 : vengono riportate considerazioni riguardanti lo studio e i confronti effettuati sui parsers presi in esame.

Capitolo 1

Grammatiche e linguaggi formali

1.1 Linguaggio: introduzione

Cerchiamo di dare una definizione precisa del termine linguaggio e di altri termini correlati ad esso. In generale, un linguaggio è un insieme di frasi in cui, dato un insieme o vocabolario di simboli (ad esempio cifre, lettere, numeri, parole, segni di punteggiatura), si definisce frase o stringa una giustapposizione[1] di un numero qualsiasi di simboli del vocabolario che dà luogo ad una nuova stringa. Se non si prende alcun simbolo si ha la stringa vuota o stringa nulla, denotata dal simbolo ε. In altre parole, una volta fissato un vocabolario, o alfabeto, un linguaggio non sarà altro che un sottoinsieme di tutte le frasi ottenibili come sequenza di simboli. Tali sequenze saranno anche riferite con il termine di stringhe. Le definizioni di simbolo, frase e vocabolario sono la base per la definizione formale di linguaggio.

Un linguaggio, quindi, è semplicemente un insieme, finito o infinito, di stringhe costruite su un vocabolario [2].

In termini matematici un linguaggio può essere definito come segue:

[1] Si definiscono stringhe per giustapposizione quelle scritte l'una di seguito all'altra.

Definizione (Linguaggio) [2] Sia Σ un insieme finito o infinito di simboli, detto alfabeto o vocabolario. Σ^+ denoti l'insieme di tutte le stringhe finite di Σ, ossia successioni ordinate di elementi di Σ. Per convenzione si aggiunga a Σ^+ la stringa nulla, denotata dal simbolo ε, costituita da nessun carattere e si indichi con Σ^* l'insieme $\Sigma^+ \cup \{\varepsilon\}$. Un Σ-linguaggio (o, qualora Σ sia sottinteso, un linguaggio) L è un sottoinsieme di Σ^*. Gli elementi di L sono detti frasi del linguaggio. Inoltre, poiché $\Sigma \subseteq \Sigma^*$, un alfabeto è a sua volta un linguaggio.

1.1.1 Operazioni su stringhe e su linguaggi

Fra le strutture matematiche utilizzate nell'ambito dell'informatica, i linguaggi svolgono un ruolo particolarmente importante.

In questo paragrafo introduciamo alcune operazioni su stringhe e su linguaggi che nel seguito avremo occasione di utilizzare.

Definizione (Concatenazione tra stringhe) [2,4,5,12] Dato un alfabeto Σ, l'elemento ε (stringa vuota) e l'operatore ∘, l'operazione $\circ : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ è chiamata concatenazione e consiste nel giustapporre due stringhe di Σ^*:

$$x_{i_1} \ldots x_{i_n} \circ y_{j_1} \ldots y_{j_m} = x_{i_1} \ldots x_{i_n} y_{j_1} \ldots y_{j_m}$$

con $x_{i_1}, \ldots, x_{i_n}, y_{j_1}, \ldots, y_{j_m} \in \Sigma$.

La concatenazione di due stringhe x e y è frequentemente indicata omettendo il simbolo ∘, cioè scrivendo xy anziché $x \circ y$.

Con la notazione $| x |$ indichiamo la lunghezza di una parola x, ovvero il numero di caratteri che la costituiscono, vale a dire

$$| xy | = | x | + | y |$$

La concatenazione è un'operazione binaria che gode della proprietà associativa, ossia $(x \circ y) \circ z = x \circ (y \circ z)$, ma non gode della proprietà commutativa, ossia $x \circ y \neq y \circ x$.

Una stringa particolare, su un alfabeto Σ, è la stringa nulla, indicata come ε, in cui vale la proprietà:

$$\forall x \quad x \circ \varepsilon = \varepsilon \circ x = x$$

dove $| \varepsilon | = 0$. In termini algebrici, ε è l'unità destra e sinistra rispetto alla concatenazione.

Un caso particolare di concatenazione è quello in cui la stringa viene concatenata h volte con se stessa. Si parla allora di potenza h-esima x^h di x, in cui risulta:

$$\begin{cases} x^h = x^{h-1} \circ x, \text{ per h} > 0 \\ x^0 = \varepsilon \end{cases}$$

Introduciamo ora alcune delle fondamentali operazioni sui linguaggi.

Definizione (Concatenazione tra linguaggi) [2,4,5,12] La concatenazione di due linguaggi L_1 e L_2 (scritto $L_1L_2 = L_1 \circ L_2$) è il linguaggio le cui stringhe sono la concatenazione di una stringa di L_1 con una stringa di L_2, ossia:

$$L_1L_2 = \{ xy \mid x \in L_1 \wedge y \in L_2 \}.$$

Si noti che $L \circ \{\varepsilon\} = \{\varepsilon\} \circ L = L$, per qualsiasi L.

Si chiama linguaggio vuoto, e lo si indica con Λ, il linguaggio che non contiene nessuna stringa, tale che

$$\forall\ L : L \circ \Lambda = \Lambda \circ L = \Lambda$$

Ciò implica che $|\ \Lambda\ | = 0$. Λ è dunque l'elemento zero rispetto alla concatenazione di linguaggi.

Si osservi che $\Lambda \neq \{\varepsilon\}$, in quanto $|\ \Lambda\ | = 0 \neq |\{\varepsilon\}| = 1$.

Similmente l'operazione di potenza viene estesa da una stringa a un linguaggio L attraverso la definizione:

$$\begin{cases} L^h = L^{h-1} \circ L, \text{ per h} > 0 \\ L^0 = \{\varepsilon\} \\ \Lambda^0 = \{\varepsilon\} \end{cases}$$

L'alfabeto Σ può essere descritto come un insieme Σ^h contenente tutte e solo le stringhe di lunghezza h. Questo insieme infinito di stringhe è detto linguaggio universale basato sull'unione di tutte le potenze Σ^h, per h = 0,1,…

Infine indichiamo la chiusura riflessiva (chiusura di Kleene) e transitiva di un linguaggio L rispetto all'operazione di concatenazione come segue:

$$L^* = \bigcup_{h=0}^{\infty} L^h = \{\varepsilon\} \cup L^1 \cup L^2 \cup \ldots$$

Si noti che $\varepsilon \in L^*$.

Si usa anche la notazione

$L^+ = \bigcup_{h=1}^{\infty} L^h$ (chiusura non riflessiva) per la quale risulta $L^* = L^+ \cup \{\varepsilon\}$.

Dunque Σ^* è il linguaggio universale dell'alfabeto Σ. Ogni linguaggio formale è un sottoinsieme del linguaggio universale di eguale alfabeto.

1.1.2 Linguaggi formali

Tra i concetti principali alla base dell'informatica quello di linguaggio riveste un ruolo molto importante. Con il termine linguaggio intendiamo un insieme – generalmente infinito – di stringhe caratterizzate da qualche particolare proprietà. Lo studio formale dei linguaggi è una parte importante dell'informatica teorica e trova applicazioni in vari settori. La prima, più evidente, è appunto lo studio delle proprietà sintattiche, cioè lo studio dei metodi di definizione della sintassi di un linguaggio e dei metodi di verifica che soddisfino le proprietà sintattiche volute. Inoltre, al di là di questa importante applicazione, la familiarità con i concetti di generazione e riconoscimento di linguaggi è fondamentale in quanto stringhe di caratteri aventi struttura particolare vengono frequentemente utilizzate.

Pertanto un linguaggio formale indica una notazione o un formalismo con sintassi e semantica definite in modo preciso e, in molti casi, tali da consentire qualche forma di elaborazione automatica del linguaggio stesso [12].

1.1.3 Rappresentazione di linguaggi: generatori e riconoscitori

Un linguaggio L è un insieme di parole su un dato alfabeto. Come è possibile dare una descrizione di L? Un approccio in tal senso è il cosiddetto approccio generativo, che vede l'utilizzo di opportuni strumenti formali, le grammatiche formali (abbreviato: grammatiche), appunto, che consentono di costruire le stringhe di un linguaggio tramite un insieme prefissato di regole, dette regole di produzione.

Un altro tipo di approccio per la descrizione di L è quello riconoscitivo, che consiste nell'utilizzare dei riconoscitori, tipo automi a stati finiti, che, per un dato linguaggio $L \subseteq \Sigma^*$, stabiliscono se una stringa $x \in \Sigma^*$ appartiene a L o meno [3,12].

1.2 Grammatica: introduzione

Introduciamo, dunque, un meccanismo per definire i linguaggi. Non a caso viene dato il nome di grammatica a tale meccanismo, che costituisce uno strumento generativo per la rappresentazione di linguaggi: la grammatica è, infatti, un insieme di regole per costruire tutte e solo le stringhe che appartengono al linguaggio. Queste regole sono meccanismi di riscrittura che permettono di descrivere le stringhe del linguaggio mediante un processo di sostituzioni successive [2].

Definizione (Grammatica) [4,12] Una grammatica G è una quadrupla (V, Σ, P, S) dove:

- V è un insieme finito non vuoto di simboli, detto alfabeto non terminale del linguaggio. Gli elementi di V sono detti tipi sintattici o

caratteri non terminali (in breve: non terminali), e vengono convenzionalmente indicati con lettere maiuscole dell'alfabeto. Tali elementi non compaiono nelle frasi finite perché vengono sostituiti con altri non terminali e/o con simboli terminali durante il processo di generazione delle frasi del linguaggio secondo quanto stabilito dalle regole della grammatica.

- Σ è un insieme finito non vuoto di simboli detto alfabeto terminale. Gli elementi di Σ sono detti caratteri terminali (in breve terminali) e vengono convenzionalmente indicati con lettere minuscole.
- P è l'insieme delle regole sintattiche o produzioni, ossia delle regole che danno la definizione ricorsiva del linguaggio. È un insieme di coppie (α,β), dette produzioni scritte $\alpha \rightarrow \beta$, dove:
    - α è detta parte sinistra della produzione ed è una stringa non vuota contenente almeno un simbolo non terminale;
    - β è detta parte destra della produzione e può contenere sia simboli terminali che simboli non terminali.
- S, simbolo distinto o assioma, è un particolare elemento di V ed è il simbolo iniziale che rappresenta il linguaggio da definire.

Per convenienza di notazione, le produzioni che hanno come parte sinistra lo stesso simbolo non terminale possono essere raggruppate mettendo nella parte destra tutte le loro forme separate dal simbolo | , letto come "oppure" [1,2].

Esempio di una grammatica:

$$S \rightarrow A$$
$$A \rightarrow aAb \mid B \mid ab$$
$$B \rightarrow aBbb \mid abb$$

dove:

$V = \{S,A,B\}$

$T = \{a,b\}$

$P = \{ S \rightarrow A , A \rightarrow aAb \mid B \mid ab , B \rightarrow aBbb \mid abb \}$

$S \in V$

□

Definizione (Linguaggio di una grammatica) [2] Il linguaggio generato da una grammatica G, L(G), è costituito da tutte le stringhe di Σ derivabili da S, cioè, con notazione matematica:

$$L(G) = \{ x \mid x \in \Sigma \wedge S \xrightarrow[G]{*} x \}.$$

Ovverosia, applicando iterativamente il meccanismo di derivazione a partire da S fino ad ottenere stringhe costituite da soli elementi di Σ, si genera il linguaggio della grammatica. Si noti che, una volta prodotta una stringa terminale, il meccanismo di derivazione non è più applicabile perché le parti sinistre delle produzioni sintattiche sono elementi di V [2].

Si osservi che una grammatica può generare un linguaggio infinito grazie al fatto che qualche non terminale può essere riscritto in modi diversi, cioè possono esistere produzioni del tipo $A \rightarrow \alpha$ e $A \rightarrow \beta$. Inoltre è possibile ottenere derivazioni autoinclusive, cioè del tipo $A \rightarrow \alpha A \beta$, il che permette di ottenere stringhe di lunghezza illimitata. Ad esempio, prendendo le produzioni $A \rightarrow aAb$, $B \rightarrow aBbb$, queste derivazioni possono essere ripetute un numero arbitrario di volte finché non si decida di chiudere rispettivamente le derivazioni con le produzioni $A \rightarrow ab$, $B \rightarrow abb$ [2].

1.2.1 Grammatiche regolari

Le grammatiche formali definite precedentemente hanno un'origine storica nei lavori di Noam Chomsky il quale propose una classificazione delle grammatiche basata sulla struttura delle regole:

- Grammatiche di tipo 0 ;
- Grammatiche contestuali o di tipo 1 ;
- Grammatiche libere dal contesto o di tipo 2 ;
- Grammatiche regolari o di tipo 3.

Prenderemo in esame l'ultimo dei tipi, le grammatiche regolari [4].

Definizione (Grammatica regolare) [4] Una grammatica si dice regolare o lineare si ammette produzioni del tipo:

$$A \rightarrow \delta$$

dove $A \in V$, $\delta \in (\Sigma \circ V) \cup \Sigma$.

Il termine "regolare" deriva dal fatto che i linguaggi appartenenti a questo tipo di grammatica sono rappresentabili per mezzo di espressioni regolari su un alfabeto E definite dalle seguenti regole:

1. ε è un'espressione regolare che denota il linguaggio $\{\varepsilon\}$;
2. se $a \in E$, a è un'espressione regolare che denota il linguaggio $\{a\}$;
3. supponiamo r e s espressioni regolari che denotano rispettivamente linguaggi L_r e L_s:
    a. r|s è un'espressione regolare che denota il linguaggio $L_r \cup L_s$;
    b. rs è un'espressione regolare che denota il linguaggio $L_r L_s$ (concatenazione);

c. r^* è un'espressione regolare che denota il linguaggio L_r^* (stella di Kleene);
d. r è un'espressione regolare che denota il linguaggio L_r.

Le espressioni regolari costituiscono uno strumento per la definizione di linguaggi equivalente agli automi a stati finiti: consentono di descrivere tutti e solo i linguaggi che è possibile definire con gli automi a stati finiti deterministici e non deterministici. Tali linguaggi sono chiamati linguaggi regolari [4,12].

Il termine "lineare" deriva dal fatto che al lato destro di ogni produzione compare al più un simbolo non terminale. Pertanto esistono due tipi di grammatica [4,12]:

- *regolare destra* se le sue regole sono tutte del tipo $A \rightarrow aB$ e $A \rightarrow b$ dove $A,B \in V$ e $a,b \in \Sigma$;
- *regolare sinistra* se le sue regole sono tutte del tipo $A \rightarrow Ba$ e $A \rightarrow b$ dove $A,B \in V$ e $a,b \in \Sigma$.

In entrambi i casi esiste anche un'eventuale produzione $S \rightarrow \varepsilon$.

Definizione (Linguaggio regolare) [12] Si dice regolare un linguaggio $L \subseteq \Sigma^+$ per il quale esiste una grammatica regolare G tale che $L = L(G)$.

1.2.2 Grammatica ambigua

Sfortunatamente non c'è un unico modo di descrivere una grammatica, anche se si descrive lo stesso linguaggio. Possono esserci più

grammatiche del tutto equivalenti, ma che differiscono per la scelta dei non terminali, per il numero e la struttura delle regole. Il fatto che lo stesso linguaggio può essere generato da grammatiche diverse complica le cose perché non tutte le grammatiche equivalenti hanno le stesse proprietà, anche se generano lo stesso linguaggio. Ad esempio, alcune grammatiche consentono di generare la stessa frase tramite alberi sintattici strutturalmente diversi. Ogni generazione corrisponde a un particolare modo di "visitare" l'albero di derivazione. Tra le diverse derivazioni possibili si possono, ad esempio, distinguere le derivazioni destre e le derivazioni sinistre: se ad ogni passo di una derivazione la produzione è applicata al simbolo non terminale più a sinistra, allora la derivazione è detta sinistra (leftmost); similmente, se viene applicata sempre a quello a destra, allora è detta destra (rightmost) [1,2,7,8].

Questo significa che la struttura sintattica di una frase, come definita dalla grammatica, non è unica, ma si presta a diverse interpretazioni. In tal caso si dice che la grammatica è ambigua. Una grammatica che genera almeno due alberi di derivazione distinti aventi la stessa frontiera è detta grammatica ambigua. In altre parole, una grammatica è ambigua se esistono più di una derivazione destra o sinistra per la stessa stringa. In una grammatica non ambigua, per ogni stringa s esiste al più un albero di derivazione il cui prodotto è s e la cui radice è il simbolo iniziale S [1,7,8].

1.2.3 Albero sintattico

Come abbiamo visto, è possibile scoprire se una stringa appartiene al linguaggio applicando ripetutamente le produzioni della grammatica. Spesso è utile rappresentare la dimostrazione che una stringa appartiene al

linguaggio generato da una grammatica mediante una struttura ad albero (parse tree - albero di riconoscimento o di derivazione).

Tale albero di derivazione è un grafo privo di cicli, tale che per ogni coppia di nodi N_1, N_2 esiste un solo cammino che li congiunge. Un arco (N_1,N_2) definisce la relazione (padre,figlio). Esiste un solo nodo S, la radice, privo di padre.

I nodi di un albero di derivazione sono etichettati da simboli terminali, oppure da simboli non terminali, oppure dal simbolo ϵ: le foglie sono etichettate solo da simboli terminali o da ε, mentre i nodi interni sono etichettati solo dai simboli non terminali.

La stringa è costituita dalla frontiera dell'albero, ovvero dalle etichette delle foglie dell'albero, prese da sinistra a destra. Se un albero ha un solo nodo, allora quel nodo, essendo una foglia, sarà etichettato da un simbolo terminale o da ϵ. Se un albero, invece, ha più di un nodo, allora la radice sarà etichettata dal simbolo non terminale iniziale S, dal momento che la radice di un albero con due o più nodi è comunque un nodo interno: tale simbolo conterrà sempre, tra le sue stringhe, il prodotto dell'albero. Si noti che la definizione di albero non impone che tutte le foglie siano etichettate da simboli terminali. Se vi sono foglie etichettate da simboli non terminali, allora l'albero rappresenta una derivazione parziale. Diremo che un albero descrive una stringa $\alpha \in (V \cup T)^*$ se α è proprio la stringa che possiamo leggere dalle etichette delle foglie da sinistra a destra.

Definizione (Albero di derivazione) [1] Sia $G = <V, \Sigma, P, S>$ una grammatica. Un albero è un albero di derivazione per G se:

1. ogni foglia è etichettata da un simbolo terminale o dal simbolo ε;
2. la radice è etichettata dal simbolo iniziale $S \in V$;

3. ogni nodo interno (ovvero non una foglia) è etichettato da un simbolo non terminale appartenente a V;
4. se un nodo n etichettato con A (simbolo non terminale) e $n_1,\ldots, n_k$ sono (ordinatamente, da sinistra a destra) i nodi figli di A etichettati con $X_1,\ldots, X_k$ allora $A \rightarrow X_1,\ldots, X_k$ è una produzione. Qui $X_1,\ldots, X_k$ sono simboli terminali o non terminali;
5. se un nodo n ha etichetta ε, allora n è una foglia ed è l'unico figlio di suo padre.

In altri termini, ogni nodo interno n rappresenta l'applicazione di una produzione, ovvero deve esistere una produzione tale che:

- i simboli non terminali che n etichetta siano la parte sinistra della produzione;
- le etichette dei figli di n, da sinistra a destra, costituiscano la parte destra della produzione.

Un albero di derivazione rappresenta dunque un certo insieme di possibili derivazioni distinte di una stessa stringa.

Come si è detto nel paragrafo delle grammatiche, ci possono tuttavia essere più alberi di derivazione per la stessa parola:

Esempio Si consideri la grammatica

$$E \rightarrow E + E \mid E * E \mid 0 \mid 1 \mid 2$$

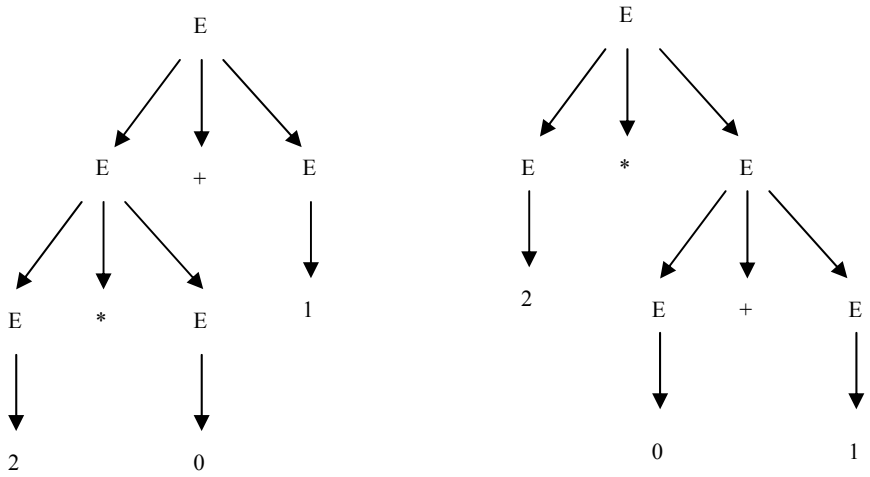
Una derivazione sinistra è:

$$E \xrightarrow{G} E + E \xrightarrow{G} E * E + E \xrightarrow{G} 2 * 0 + 1$$

Un'altra derivazione sinistra per la stessa stringa è:

$$E \xrightarrow{G} E * E \xrightarrow{G} 2 * E \xrightarrow{G} 2 * E + E \xrightarrow{G} 2 * 0 + 1$$

Tuttavia gli alberi associati alle due derivazioni (che potrebbero essere visti come alberi per la computazione dell'espressione associata alla stringa generata) sono diversi:



Il primo è intuitivamente associato a (2∗0)+1=1, il secondo, invece, a 2∗(0+1) = 2.

□

1.3 Automi a stati finiti: introduzione

In questo paragrafo parliamo dell'automa a stati finiti (ASF), un altro strumento equivalente alle grammatiche regolari e in grado di essere usato ugualmente bene per descrivere un linguaggio regolare. Gli automi a stati finiti sono dotati di un insieme finito di stati i quali, in particolare,

permettono di risolvere il problema di riconoscere se una stringa, una volta scandita in maniera ordinata, appartiene o no al linguaggio che l'automa rappresenta. Pertanto, una stringa appartiene al linguaggio se viene accettata dal corrispondente automa, ovvero se porta l'automa dallo stato iniziale allo stato finale o di accettazione [4,13].

Formalmente gli automi a stati finiti sono delle quintuple, $<Q, \Sigma, \delta, q_0, F>$, formate da un alfabeto finito dei simboli in ingresso Σ, un insieme finito di stati Q tra cui si distinguono lo stato iniziale q_0 ed un sottoinsieme di stati, detti finali F, ed una funzione di transizione δ. Tale funzione, descritta mediante una tabella di transizione degli stati, è definita per coppie (stato corrente, simbolo scandito) e stabilisce la transizione da compiere, ossia lo stato in cui si transita leggendo il simbolo dato [13].

Un automa a stati finiti può essere rappresentato mediante un grafo orientato ed etichettato, detto diagramma di stato, i cui nodi rappresentano gli stati e i cui archi rappresentano le transizioni tra gli stati. Il grafo ha un arco q_i a q_j con etichetta x se e solo se $q_j \in \delta(q_j,a)$. Lo stato iniziale viene indicato mediante una freccia ondulata, mentre lo stato finale è indicato con un nodo con doppia cerchiatura [4].

In altre parole il funzionamento dell'automa può essere così descritto: partendo dallo stato iniziale e dal primo simbolo della stringa si decide, in base alla funzione, di transitare in un determinato stato (potrebbe anche essere lo stato iniziale stesso). Finché esiste un altro simbolo nella stringa da scandire si opera alla stessa maniera fino ad esaurire la stringa in ingresso.

La stringa si dirà accettata se si giunge in uno stato appartenente al sottoinsieme degli stati finali [13].

Di seguito caratterizziamo gli automi a stati finiti deterministici e gli automi a stati finiti non deterministici.

1.3.1 Automa a stati finiti deterministici

Definizione (Automa a stati finiti deterministici) [4,12] Un automa a stati finiti deterministici (ASFD) A è una quintupla <Q, Σ, δ, q_0, F> dove:

- $Q = \{q_1,\ldots, q_m\}$ è un insieme finito e non vuoto di stati;
- $\Sigma = \{a_1,\ldots, a_n\}$ è un insieme finito e non vuoto di simboli detto alfabeto di ingresso;
- $\delta : Q \times (\Sigma \cup \varepsilon) \rightarrow Q$ è la funzione (totale) di transizione che ad ogni coppia <stato,carattere in input> associa uno stato successivo;
- $q_0 \in Q$ è lo stato iniziale;
- $F \subseteq Q$ è l'insieme (non vuoto) degli stati finali.

Useremo p, q, r con o senza pedici per denotare stati; P, Q, R, S per insiemi di stati; *a*, *b* con o senza pedici per denotare simboli di Σ; *x*, *y*, *z*, *u*, *v*, *w* sempre con o senza pedice per denotare stringhe.

Il comportamento dell'automa così definito è caratterizzato in particolare dalla funzione di transizione. La funzione δ indica la computazione dell'automa: il significato di $\delta(q_i,x) = q_j$ è che A, trovandosi nello stato q_i e leggendo *x*, si porta nello stato q_j. In particolare si conviene che, per ogni q_i, $\delta(q_i,\varepsilon) = q_i$.

Per elaborare una stringa *x*, con $| x | > 1$, l'automa farà una serie di computazioni.

Quindi δ è una funzione con dominio finito, per cui può venire rappresentata mediante una tabella, detta tabella di transizione (o, equivalentemente, matrice di transizione), alle cui righe vengono associati gli stati, alle colonne i caratteri in input, ed i cui elementi rappresentano il

risultato dell'applicazione del δ allo stato identificato dalla riga ed al carattere associato alla colonna della tabella 1 [12].

$\boldsymbol{\delta}$	***a***	***b***
$\mathbf{q_0}$	q_0	q_1
$\mathbf{q_1}$	q_2	q_2
$\mathbf{q_2}$	q_2	q_2

Tabella 1. Esempio di tabella di transizione di un ASFD

Un'altra rappresentazione è costituita dal cosiddetto diagramma degli stati (o grafo di transizione), in cui l'automa è rappresentato mediante un grafo orientato: i nodi rappresentano gli stati, mentre gli archi rappresentano le transizioni, per cui sono etichettati con il carattere la cui lettura determina la transizione. Gli stati finali sono rappresentati da nodi con un doppio circolo, mentre quello iniziale è individuato tramite una freccia entrante [12].

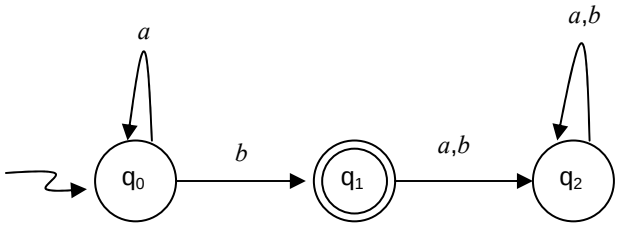


Figura 1. Diagrammi degli stati dell'ASFD della tabella 1

Esempio Data la stringa $x = aab$, allora si avrà che $\delta(q_0,a) = q_0$, $\delta(q_0,b) = q_1$; per comodità si scriverà $\delta(q_0,aab) = q_1$.

Essenzialmente, un ASFD esegue una computazione nel modo seguente: a partire dalla situazione in cui lo stato attuale è q_0, l'automa ad ogni passo legge il carattere successivo della stringa in input ed applica la funzione δ alla coppia formata dallo stato attuale e da tale carattere per determinare lo stato successivo. Terminata la lettura della stringa, essa viene accettata se lo stato corrente è uno stato finale, altrimenti rifiutata.

Al fine di formalizzare meglio il modo di operare di un ASFD possiamo osservare, ricordando quanto detto in precedenza, che il comportamento futuro di un automa a stati finiti deterministici dipende esclusivamente dallo stato corrente e dalla stringa da leggere. Da ciò, e da quanto esposto sopra, conseguono le seguenti definizioni [4,5,12][2].

Definizione (Configurazione) [4] Dato un automa a stati finiti $A = <Q, \Sigma, \delta, q_0, F>$, una configurazione di A è una coppia (q,x), con $q \in Q$ e $x \in \Sigma^*$.

Una configurazione (q,x), $q \in Q$ ed $x \in \Sigma^*$, di A, è detta:

- iniziale se $q = q_0$;
- finale se $x = \varepsilon$;
- accettante se $x = \varepsilon$ e $q \in F$.

La funzione di transizione permette di definire la relazione di transizione $\rightarrow$ tra configurazioni, che associa ad una configurazione la configurazione successiva nel modo seguente.

[2] Cfr. anche [1,4,5,8,9,10,11,12].

Definizione (Relazione di transizione) [4] Dato un ASFD A = <Q, Σ, δ, q_0, F> e due configurazioni (q,x) e (q',y) di A, avremo che (q,x) $\xrightarrow[A]{}$ (q',y) se e solo se valgono le due condizioni:

1. esiste $a \in \Sigma$ tale che $x = ay$;
2. δ(q,a) = q'.

A questo punto possiamo dare le definizioni che seguono.

Definizione (Linguaggio riconosciuto da ASFD) [4] Dato un automa a stati finiti A = <Q, Σ, δ, q_0, F>, una stringa $x \in \Sigma$ è accettata da A se e solo se (q_0,x) $\xrightarrow[A]{*}$ (q,ε), q∈F, e possiamo definire il linguaggio riconosciuto da A come

$$L(A) = \{ x \in \Sigma^* \mid (q_0,x) \xrightarrow[A]{*} (q,\varepsilon),\ q \in F \}$$

Possiamo definire il linguaggio riconosciuto da un automa a stati finiti deterministici utilizzando un'estensione della funzione di transizione da caratteri a stringhe.

Definizione (Funzione di transizione estesa) [4] La funzione di transizione estesa di un automa a stati finiti deterministici A = <Q, Σ, δ, q_0, F> è la funzione $\delta^* : Q \times \Sigma^* \rightarrow Q$, definita nel modo seguente:

$$\begin{cases} \delta^*(q,\varepsilon) = q \\ \delta^*(q,a) = \delta(q,a) \\ \delta^*(q, xa) = \delta(\delta^*(q,x),a) \end{cases}$$

dove $a \in \Sigma$, $x \in \Sigma^*$.

Dalla definizione precedente avremo quindi che, dati uno stato q ed

una stringa $x \in \Sigma^*$, lo stato q' è tale che q' = δ^*(q,x) se e solo se la computazione eseguita dall'automa a partire da q ed in conseguenza della lettura della stringa x conduce l'automa stesso nello stato q'. Più formalmente, possiamo anche dire che q' = δ^*(q,x) se e solo se esiste $y \in \Sigma^*$ tale che $(q,xy) \xrightarrow{*} (q',y)$. Di conseguenza, avremo che una stringa $x \in \Sigma^*$ è accettata da un ASFD A = <Q, Σ, δ, q_0, F> se e solo se $\delta^*(q_0,x) \in F$ [4,5,8,9,12].

È allora possibile introdurre la seguente definizione di linguaggio riconosciuto da una ASFD.

Definizione [12] Il linguaggio riconosciuto da un automa a stati finiti deterministici A è l'insieme

$$L(A) = \{ x \in \Sigma^* \mid \delta^*(q_0,x) \in F \}.$$

1.3.2 Automa a stati finiti non deterministici

In questa sezione prenderemo in esame la classe più estesa degli automi a stati finiti, costituita dalla classe degli automi a stati finiti non deterministici. Tale estensione si ottiene definendo l'insieme delle transizioni come una funzione nell'insieme delle parti dell'insieme degli stati. In altre parole tali automi sono caratterizzati dal fatto che, ad ogni passo, un'applicazione della funzione di transizione definisce più stati anziché uno solo.

Definizione (Automa a stati finiti non–deterministici) [4,12] Un automa a stati finiti non deterministici (ASFND) è una quintupla A_N = <Q, Σ, δ, q_0, F> in cui:

- $Q = \{ q_1,\ldots, q_m \}$ è un insieme finito e non vuoto degli stati interni;
- $\Sigma = \{a_1,\ldots, a_n\}$ è un insieme finito e non vuoto di simboli detto alfabeto di ingresso;
- $\delta_N : Q \times \Sigma \rightarrow \wp(Q)$ è una funzione (parziale), detta funzione di transizione, che ad ogni coppia <stato,carattere> su cui è definita associa un sottoinsieme di Q (eventualmente vuoto);
- $q_0 \in Q$ è lo stato iniziale;
- $F \subseteq Q$ è l'insieme (non vuoto) degli stati finali.

Mentre la rappresentazione mediante tabella degli automi non deterministici prevede, in ogni casella, l'inserimento di un insieme di stati, a differenza di quella per i deterministici, la rappresentazione a grafo rimane pressoché immutata. L'unica differenza è che da un nodo possono uscire più archi (o nessuno) etichettati dallo stesso simbolo.

Nella tabella 2 viene presentata la funzione di transizione di un automa a stati finiti non deterministici: come si può osservare, in corrispondenza di alcune coppie stato-carattere (segnatamente le coppie (q_0,b) e (q_1,a)) la funzione fornisce più stati, corrispondenti a diverse possibili continuazioni di una computazione.

δ_N	***a***	***b***
$\mathbf{q_0}$	$\{q_0\}$	$\{q_0, q_1\}$
$\mathbf{q_1}$	$\{q_2, q_3\}$	$\{q_3\}$
$\mathbf{q_2}$	$\{q_3\}$	Ø
$\mathbf{q_3}$	Ø	Ø

Tabella 2. Esempio di tabella di transizione di un ASFND

Nella figura 2, invece, viene riportato il grafo di transizione corrispondente all'automa a stati finiti non deterministici la cui funzione di transizione è descritta nella tabella sopra. La presenza del non determinismo fa sì che dal nodo etichettato q_0 escano due archi etichettati con il carattere b, così come che dal nodo etichettato q_1 escano due archi etichettati con il carattere a.

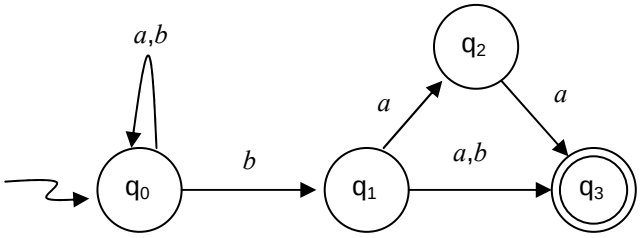


Figura 2. Grafo di transizione dell'ASFND della tabella 2

Una stringa viene accettata da un automa a stati finiti non deterministici se almeno una delle computazioni definite per la stringa stessa è di accettazione.

Definizione (Linguaggio riconosciuto da ASFND) [12] Una stringa *x* è accettata se $(\{q_0\},x) \xrightarrow{*} (Q,\varepsilon)$, dove $Q \subseteq Q$ e $Q \cap F \neq \emptyset$.

Possiamo allora definire nel modo seguente il linguaggio L(A) accettato da un ASFND:

$$L(A) = \{ x \in \Sigma^* \mid (\{q_0\},x) \xrightarrow{*} (Q,\varepsilon), \quad Q \cap F \neq \emptyset \}$$

Un modo alternativo di definire il linguaggio accettato da un ASFND richiede, analogamente a quanto fatto nel caso degli ASFD, di

estendere alle stringhe la definizione di funzione di transizione.

Definizione (Funzione di transizione estesa) [5,12] Dato un ASFND, la funzione di transizione estesa è la funzione δ_N^* : Q × Σ* → $\wp$(Q), definita nel seguente modo

$$\begin{cases} \delta_N^*(q,\varepsilon) = \{q\} \\ \delta_N^*(q,xa) = \bigcup_{p \in \delta_N^*(q,x)} \delta_N(p,a) \end{cases}$$

dove a∈Σ, $x \in \Sigma^*$, p∈Q.

Dalla definizione precedente avremo quindi che, dati uno stato q ed una stringa *x*, lo stato q' appartiene all'insieme $\delta_N^*(q,x)$ se e solo se esiste una computazione dell'automa la quale, a partire da q ed in conseguenza della lettura della stringa *x*, conduce allo stato q'.

Definiamo ora la classe di linguaggi accettati dagli ASFND.

Definizione [12] Il linguaggio accettato da un ASFND A_N è l'insieme

$$L(A_N) = \{ x \in \Sigma^* \mid \delta_N^*(q_0,x) \cap F \neq \emptyset \}.$$

Capitolo 2

Riconoscitori deterministici

2.1 Introduzione

L'attenzione finora dedicata alle grammatiche formali è dovuta alla loro capacità di generare linguaggi di cardinalità infinita e alle loro connessioni con gli automi a stati finiti. In questo capitolo consideriamo gli aspetti relativi alla struttura del linguaggio. Tale studio è giustificato da due considerazioni:

1. la struttura gioca un ruolo fondamentale in alcune teorie sulla comprensione e sull'elaborazione del linguaggio;
2. nel trattamento del linguaggio la struttura di una frase ha un ruolo predominante.

Le grammatiche offrono dei vantaggi significativi:

- una grammatica dà ad un linguaggio una specifica sintattica precisa e facile da capire;
- per alcune classi di grammatica possiamo automaticamente costruire dei parsers efficienti che determinano se un certo linguaggio è sintatticamente corretto 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;

- una grammatica impone una struttura al linguaggio;
- un linguaggio può evolversi 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.

Pertanto in questo capitolo ci occuperemo di definire dei linguaggi tramite grammatiche libere dal contesto, tutto ciò che riguarda le derivazioni di un albero di derivazione e le metodologie di parsing per alcune grammatiche [4].

2.2 Analisi sintattica

Le grammatiche permettono, dunque, di definire rigorosamente, in modo quasi completo, un linguaggio. Occorrono, perciò, strumenti per il riconoscimento di un linguaggio e per la costruzione dell'albero sintattico. Questi due procedimenti vengono generalmente raggruppati in un'unica fase di elaborazione, detta analisi sintattica.

L'analisi sintattica è il procedimento di costruzione della derivazione di una frase rispetto a una grammatica data G. Un analizzatore sintattico è uno strumento che opera su una stringa x: se $x \in L(G)$ l'analizzatore sintattico produce una derivazione di x e, in tal caso, costruisce l'albero sintattico di x (o gli alberi sintattici, qualora G sia ambigua), altrimenti si

ferma indicando il punto in cui si sono presentati l'errore ed il tipo d'errore (diagnosi); eventualmente tenta di proseguire analizzando certe sottostringhe di *x* non troppo contaminate dall'errore, ossia tenta di riparare l'effetto dell'errore [5].

In generale accade che un analizzatore sintattico, quando si accorge di aver percorso una strada sbagliata, torni al punto in cui si è verificato l'errore e riparta prendendo una strada alternativa. Ciò significa anche ritornare, nella sequenza di simboli, ad un simbolo già esaminato precedentemente. Questo evento prende il nome di backtracking o backup (rintracciamento).

Esempio Data la grammatica

1. $S \rightarrow Ax$
2. $A \rightarrow v$
3. $A \rightarrow B$
4. $B \rightarrow vw$

si verifichi se la stringa *vwx* è derivabile da S o meno. Per riconoscere S dobbiamo prima riconoscere A. Considerando la produzione 2., nel simbolo *v* possiamo riconoscere A. Per riconoscere S, *x* dovrebbe coincidere con il resto della stringa, cioè con *wx*. Questo, ovviamente, è falso, perciò dobbiamo ritornare alla testa della stringa (backtracking) e riconoscere in *vw* prima B e poi A; allora il rimanente *x* della sentenza coincide con la *x* della produzione 1., quindi S è riconosciuta.

□

A parte, dunque, il trattamento degli errori, un analizzatore è un riconoscitore con, in più, la capacità di ricostruire la derivazione della stringa. In questo caso non si parla di analisi sintattica ma soltanto di

riconoscimento, poiché la struttura delle stringhe del linguaggio è nota a priori, trattandosi di alberi sintattici che crescono soltanto verso destra (o soltanto verso sinistra) se le produzioni sono del tipo $A \rightarrow aB$ (o $A \rightarrow Ba$). Il riconoscimento viene effettuato da un automa a stati finiti [5].

Le principali caratteristiche degli strumenti di analisi sintattica sono: la velocità, la quantità di memoria occupata, la capacità di riconoscere ed elaborare errori sintattici, la generalità della classe di linguaggi riconosciuti e la possibilità di costruzione automatica dell'analizzatore.

In un analizzatore non deterministico può succedere che si effettuino dei tentativi di costruzione dell'albero sintattico che possono successivamente rivelarsi sbagliati; l'analizzatore richiede, nel caso peggiore, un tempo $O(n^p)$ per riconoscere se una stringa appartiene al linguaggio definito dalla sintassi, dove n è la lunghezza del testo e p è il numero delle produzioni nell'ordine di 2 o 3. Anche se la teoria dice che non tutti i linguaggi liberi dal contesto sono analizzabili in modo deterministico, in pratica, per ragioni di efficienza, si usano quasi sempre gli analizzatori deterministici che operano in tempo lineare rispettivamente alla lunghezza della stringa (complessità di calcolo $O(n)$). Questo è possibile solo nel caso in cui non si abbiano costrutti ambigui o non deterministici [3].

A seconda di come viene costruito un albero sintattico si hanno due tecniche di analisi:

- analisi discendente deterministica (top-down parsing): si costruisce la derivazione canonica sinistra (predittiva), cioè l'automa ricostruisce l'albero sintattico della stringa dall'alto (la radice) verso il basso (le foglie) leggendo la stringa da sinistra a destra;
- analisi ascendente deterministica (bottom-up parsing): si costruisce la derivazione canonica destra della stringa in ordine riflesso, cioè

dalle foglie alla radice dell'albero (a spostamento e riduzione) leggendo la stringa da sinistra a destra.

L'analisi discendente deterministica si applica alle grammatiche che soddisfano la tecnica LL(k), con $k \geq 1$, purché si verifichino certe condizioni: occorre che, per ogni coppia di alternative $A \rightarrow \alpha$ e $A \rightarrow \beta$, l'analizzatore possa fare la scelta guardando i successivi k elementi. Ciò implica che α e β non possano iniziare con gli stessi elementi [2,5,6].

L'analisi ascendente si applica alle grammatiche che soddisfano la condizione di adeguatezza LR(k). La costante k è la lunghezza della prospezione, cioè il numero di caratteri terminali da esaminare per decidere la mossa dell'analizzatore. Di solito l'algoritmo può procedere senza considerare alcun elemento oltre quello corrente (cioè con $k = 0$), tranne in alcune situazioni in cui occorre esaminare l'elemento successivo ($k = 1$). Le grammatiche che si adattano a questo modo di procedere sono quelle SLR(1) e LALR(1) [2,5,6].

2.3 Grammatiche libere dal contesto

I linguaggi liberi dal contesto sono particolarmente interessanti in quanto sono un'estensione dei linguaggi regolari; tali linguaggi sono definiti dalle grammatiche libere dal contesto (o context free), che descrivono le regole con cui possono essere derivate tutte le stringhe che appartengono al linguaggio. Tali grammatiche sono state introdotte come primo strumento per definire le strutture sintattiche del linguaggio.

Una grammatica libera dal contesto è una quaterna $G = (V, \Sigma, P, S)$, dove V è l'insieme finito che rappresenta i simboli non terminali, Σ

l'insieme finito dei simboli terminali o alfabeto, P l'insieme delle produzioni, e S il simbolo iniziale che rappresenta il linguaggio da definire.

Definizione (Grammatiche context free) [13] Secondo la classificazione di Chomsky, le grammatiche context free ammettono produzioni del tipo:

$$A \rightarrow \beta, \quad A \in V, \beta \in (V \cup \Sigma)^*$$

cioè produzioni in cui ogni non terminale A può essere riscritto in una stringa β indipendentemente dal posto in cui A compare in una forma di frase.

I linguaggi generabili da grammatiche context free vengono detti linguaggi context free o liberi dal contesto.

Definizione (Linguaggio context free) [13] Un linguaggio L è detto linguaggio libero dal contesto se esiste una grammatica libera dal contesto G tale che L = L(G).

2.4 Alberi di derivazione

La sequenza delle regole sintattiche utilizzate per generare una stringa *x* da una grammatica libera dal contesto G di assioma S definisce la struttura di *x*, che dunque potrebbe essere rappresentata da una delle derivazioni $S \xrightarrow[G]{*} x$. Al fine di disporre di una rappresentazione univoca si preferisce, però, ricorrere agli alberi sintattici definiti nel capitolo 1. Il processo di costruzione dell'albero sintattico associato a una stringa prende il nome di analisi sintattica.

Definizione (Stringa associata) [7,8] La stringa *x* 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 *x* se *x* è la sua stringa associata.

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

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

Definizione (Linguaggio generato) [7,8] Sia G = (V, Σ, P, S) una grammatica libera dal contesto. Il linguaggio generato da G (indicato con L(G)) è l'insieme delle stringhe su $x \in L(G)$ tali che esiste un albero di derivazione di G, di radice S, la cui stringa associata è *x*.

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 = (V, Σ, S, P) una grammatica libera da contesto. Sia α una stringa che contiene almeno un simbolo non terminale A, cioè α = δAγ. Possiamo applicare ad α un passo di derivazione utilizzando una produzione dell'insieme 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 (Passo di derivazione) [1,7,8] Sia G = (V, Σ, S, P) una grammatica libera dal contesto e sia α una stringa che contiene almeno un simbolo non terminale A. Se $A \rightarrow X_1X_2\ldots X_k \in P$, allora possiamo riscrivere:

$$\alpha = \delta A\gamma \underset{G}{\rightarrow} \delta X_1X_2\ldots X_k\gamma$$

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

Definizione (Derivazione) [1,7,8] Sia G = (V, Σ, S, P) una grammatica libera dal contesto. Una derivazione di una stringa $w \in \Sigma^*$ a partire da S è una sequenza:

$$S = \alpha_0 \underset{G}{\rightarrow} \alpha_1 \underset{G}{\rightarrow} \alpha_2 \ldots \underset{G}{\rightarrow} \alpha_n = w$$

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

Si noti che se i<1 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 zero 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 $S \xrightarrow[G]{*} \alpha$, mentre $S \xrightarrow[G]{+} \alpha$ indica una derivazione lunga almeno un passo da S a α.

Possiamo definire il linguaggio generato dalla grammatica anche con la nozione di derivazione, dicendo che $L(G) = \{ w \in \Sigma^* \mid S \xrightarrow[G]{*} w \}$.

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 alla costruzione di un albero di derivazione [1].

Esempio Consideriamo la grammatica:

$$E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid id$$

una derivazione per la stringa $- (id + id)$ è

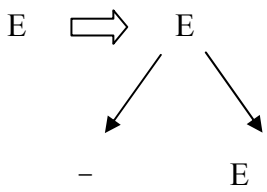
$$E \rightarrow - E \rightarrow - (E) \rightarrow - (E + E) \rightarrow - (id + E) \rightarrow - (id + id)$$

Ogni passo di derivazione si riflette, nella generazione del corrispondente albero di derivazione, nell'aggiunta di figli al nodo corrispondente al simbolo non terminale che si sta visitando.

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 $E \rightarrow - E$ e quindi vengono aggiunti i figli (– ed E) alla radice:



Il secondo passo di derivazione è $-E \rightarrow -(E)$ dove aggiungiamo tre figli, etichettati (, E, e), ottenendo così un albero con foglie $-(E)$.

Continuando in questo modo otteniamo l'albero di derivazione completo in sei passi.

Tutte le altre trasformazioni sono riportate nella figura 3.

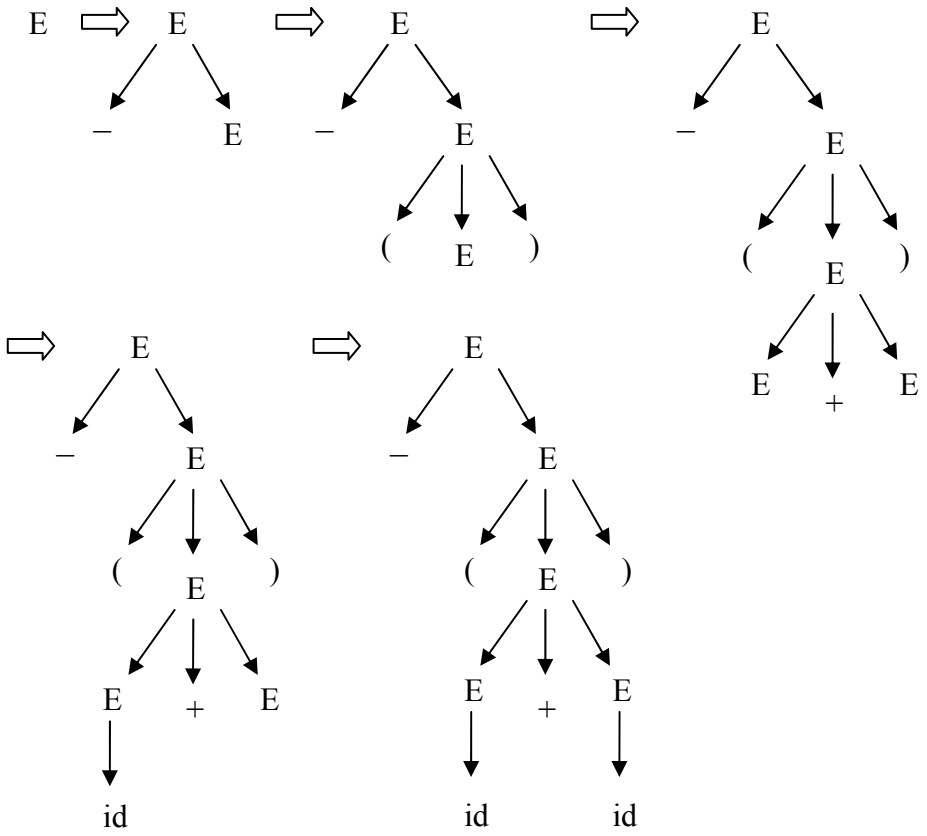


Figura 3 Albero di derivazione

Si noti che, durante la derivazione di una stringa dal simbolo iniziale della grammatica, occorre fare due scelte ad ogni passo. Supponiamo di

avere una forma sentenziale β (cioè $S \xrightarrow{*} \beta$) che non sia una stringa di soli terminali. In generale, per fare un passo di derivazione β → β', bisogna:

1. individuare un simbolo non terminale in β che sarà il candidato per la riscrittura: β = αAγ ;
2. scegliere una produzione A → δ, fra le eventuali possibili scelte per A, con la quale effettuare la riscrittura β = αAγ → αδγ.

Per il primo tipo di scelta possiamo fissare una regola che identifichi univocamente ad ogni passo il simbolo non terminale da riscrivere. 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: β = *x*Aα (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 $x\text{A}\alpha \rightarrow_{lm} x\delta\alpha$ [3]. In caso di più passi di derivazione leftmost usiamo $\rightarrow_{lm}^{*}$ per zero o più passi e $\rightarrow_{lm}^{+}$ 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 [2,4,11,13].

- **Derivazione Rightmost** Ad ogni passo riscriviamo il simbolo non terminale più a destra. Le forme sentenziali sono dette forme

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

sentenziali destre e la notazione $\rightarrow_{rm}$ è usata con le stesse estensioni viste nel caso leftmost: $S \rightarrow_{rm}^{*} \alpha A x \rightarrow_{rm} \alpha\delta x$ [2,4,11,13].

2.5 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 riscrittura di una grammatica come la definizione di un algoritmo (ricorsivo) finalizzato a 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 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 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.

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

Definizione (Ambiguità) [3,6,13] 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.

Viceversa, 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 [1].

Quindi, ad esempio, prendiamo la regola leftmost (o rightmost): si ha una grammatica non ambigua se e solo se per ogni stringa esiste un'unica derivazione leftmost (o rightmost).

Esempio Consideriamo la seguente grammatica:

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

Questa grammatica è ambigua poiché per la stringa id+id*id esistono i due alberi di derivazione mostrati in figura 4. Equivalentemente esistono due derivazioni leftmost.

La seguente derivazione corrisponde all'albero in figura 4 (a):

$$E \rightarrow_{lm} E + E \rightarrow_{lm} id + E \rightarrow_{lm} id + E * E \rightarrow_{lm} id + id * E \rightarrow_{lm} id + id * id$$

La seguente, invece corrisponde, all'albero in figura 4 (b):

$$E \rightarrow_{lm} E * E \rightarrow_{lm} E + E * E \rightarrow_{lm} id + E * E \rightarrow_{lm} id + id * E \rightarrow_{lm} id + id * id$$

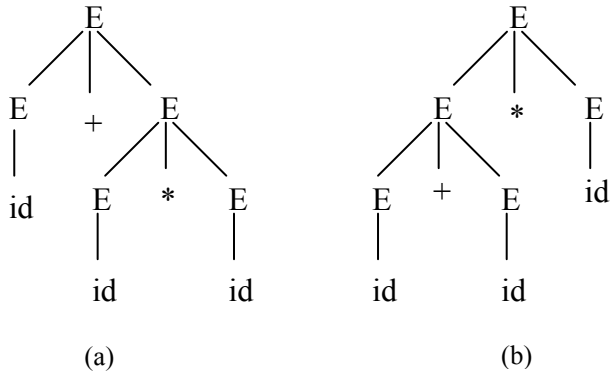


Figura 4 Albero di derivazione

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, l'ambiguità è semplice: basta esibire un controesempio, cioè una stringa per cui esistono due alberi di derivazione oppure due derivazioni leftmost (o rightmost). Per vedere se una grammatica è ambigua oppure no è bene innanzitutto cercare di capire il linguaggio generato dalla stessa e il modo in cui vengono generate le stringhe, cioè vedere se c'è una grammatica 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 rendere la grammatica non ambigua. Quest'ultima 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 alberi di derivazione associati nella grammatica di partenza [3,6,13].

2.6 Automi a pila

In questo paragrafo descriviamo una particolare classe di riconoscitori di linguaggi che corrisponde ad un arricchimento degli automi a stati finiti. Questi non sono in grado di riconoscere linguaggi che, per la loro struttura, richiedono di ricordare una quantità di "informazioni" non limitate, come nel caso del linguaggio $a^n b^n$ che richiede di ricordare il numero di "*a*" letti per poter controllare che di seguito ci sia uno stesso numero di "*b*". Per aumentare la capacità di riconoscimento dell'automa a stati finiti, occorre avere la possibilità di ricordare una quantità illimitata di informazioni e far dipendere l'azione da compiere non solo dal carattere letto e dallo stato ma anche da un altro parametro che dipende in qualche modo dalle informazioni memorizzate. Precisiamo meglio questi concetti

descrivendo un modello che chiameremo *automa a pila* corrispondente, quindi, all'automa a stati finiti con, in più, una struttura informativa comunemente detta pila. La pila è una struttura di tipo "LIFO" (Last In First Out) e, da un punto di vista architetturale, è una sequenza illimitata di informazioni, nel nostro caso di simboli $A_1,A_2,\ldots,A_k$, alla quale si accede mediante le seguenti operazioni [3,5,12]:

- l'istruzione push(X) (impilamento) inserisce il simbolo X in testa alla pila; la pila cresce, così, di un elemento;
- l'istruzione pop(X) (disimpilamento) preleva il simbolo A_k in testa alla pila (purchè empty = false); la pila si accorcia, così, di un elemento;
- l'istruzione empty (vuoto) verifica, restituendo valore booleano, se la pila è vuota o meno. La pila è vuota solo se k = 0.

Più in generale push(X_1, X_2,…, X_n) equivale a push(X_1), push(X_2),…, push(X_n) [5].

Definizione (Automa a pila) [5,12] Un automa a pila (o automa push-down) M è una settupla:

$$M = < Q, \Sigma, \Gamma, \delta, q_0, Z_0, F >$$

dove:

- Q è un insieme finito e non vuoto di stati;
- Σ è un alfabeto finito;
- Γ è l'alfabeto finito della pila;
- $q_0 \in Q$ è lo stato iniziale;
- $Z_0 \in \Gamma$ è il simbolo iniziale nella pila;
- $F \subseteq Q$ è l'insieme degli stati finali;

- δ è la funzione che rappresenta le transizioni dell'automa (funzione di transizione). Il risultato di una transizione è rappresentato da uno stato e da una stringa di simboli che viene inserita in cima alla pila al posto del simbolo corrente:

  $$\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \wp_f(Q \times \Gamma^*)$$

  Con $\wp_f(Q \times \Gamma^*)$ indichiamo l'insieme delle parti finite di $(Q \times \Gamma^*)$.

  Osserviamo che Γ^* contiene infinite stringhe.

Dalla definizione appare chiaro che l'automa a pila è non deterministico sia perché il codominio di δ è un insieme di insiemi (come nel caso degli automi finiti), sia perché la presenza di ε-transizioni può comportare che esistano continuazioni diverse di una stessa computazione anche in presenza di uno stesso carattere letto [12].

Per ogni $q \in Q$, per ogni $A \in \Gamma$ e per ogni $w \in \Sigma$, l'insieme $\delta(q,w,A)$, quando si trova nello stato q, legge w dalla stringa di ingresso e il simbolo A si trova in cima alla pila, definisce l'insieme delle possibili transizioni dell'automa.

Pertanto se

$$\delta(q,w,A) = \{(q_1,\beta_1), (q_2, \beta_2), \ldots,(q_k,\beta_k)\}$$

per ogni i = 1,…, k, l'automa può entrare nello stato q_i e sostituire A con la stringa β_i in cima alla pila. Analogamente, per ogni $q \in Q$ e per ogni $A \in \Gamma$ $\delta(q,\varepsilon,A)$ definisce l'insieme delle ε-transizioni che l'automa può compiere trovandosi nello stato q e leggendo il simbolo A in cima alla pila. Tali transizioni non dipendono dal simbolo letto nella stringa. Se $\delta(q,\varepsilon,A) = \{(p_1,\gamma_1), (p_2,\gamma_2), \ldots,(p_k,\gamma_k)\}$, allora per ogni i = 1,…, k, l'automa può entrare nello stato p_i e sostituire il simbolo A con la stringa γ_i in cima alla pila. In altre parole, dalla definizione di transizione deriva che ad ogni passo

l'automa, a partire dallo stato in cui si trova, dal carattere letto dalla stringa e dal carattere in cima alla pila, sostituisce il simbolo in cima alla pila con una stringa di caratteri e si porta in un nuovo stato.

Esempio Se un automa a pila si trova in uno stato q_1, legge il carattere $w \in \Sigma$ dalla stringa ed il simbolo $A \in \Gamma$ in cima alla pila; se la funzione di transizione è tale che $\delta(q_1, w, A) = (q_3, BA)$, ne deriva che l'automa va nello stato q_3 sostituendo al simbolo A sulla pila la stringa BA.

Esempio Se un automa a pila si trova in uno stato q_1, legge il carattere $w \in \Sigma$ dalla stringa ed il simbolo $A \in \Gamma$ in cima alla pila; se la funzione di transizione è tale che $\delta(q_1,w,A) = (q_3,\varepsilon)$, ne deriva che l'automa va nello stato q_3 cancellando il simbolo A in cima alla pila.

Per rappresentare la funzione di transizione di un automa a pila utilizzeremo una semplice estensione della tabella di transizione usata per gli automi a stati finiti, in cui le righe corrispondono agli stati dell'automa ed ogni colonna corrisponde ad una coppia composta dal carattere letto in ingresso e dal carattere in cima alla pila.

Come si può osservare, il comportamento futuro di un automa a pila è determinato esclusivamente dallo stato attuale, dalla stringa da leggere e dal contenuto della pila.

Esistono due criteri di riconoscimento di stringhe per gli automi a pila: per stato finale o per svuotamento della pila [5].

Definizione (Accettazione per stato finale) [12] Sia M un automa a pila. Una configurazione (q,x,γ) di M è di accettazione se $x = \varepsilon$ e $q \in F$.

Secondo tale definizione una stringa *x* è quindi accettata da M se e solo se al termine della scansione della stringa l'automa si trova in uno stato finale. Indicheremo con $L_F(M)$ il linguaggio accettato per stato finale dell'automa.

Definizione (Accettazione per pila vuota) [12] Sia M un automa a pila. Una configurazione (q,x,γ) di M è di accettazione se $x = \gamma = \varepsilon$. Secondo tale definizione una stringa *x* è quindi accettata da M se e solo se al termine della scansione della stringa la pila è vuota. Indicheremo con $L_P(M)$ il linguaggio accettato, per pila vuota, dell'automa M.

Per definire questi criteri dobbiamo formalmente introdurre la nozione di configurazione.

Definizione (Configurazione) [5,12] Dato un automa a pila M, una configurazione di M è data dalla terna (q,x,γ) dove:

- $q \in Q$ è lo stato corrente;
- $x \in \Sigma^*$ è la stringa ancora da leggere;
- $\gamma \in \Gamma^*$ è la sequenza di simboli contenuti nella pila.

La configurazione iniziale è (q_0,x,Z_0), mentre (q,ε,γ), con $q \in F$, è quella finale.

La funzione di transizione δ permette di definire la relazione di transizione $\rightarrow$, che associa ad una configurazione la configurazione successiva.

Definizione (Relazione di transizione) [12] Sia M un automa a pila. La relazione di transizione fra le configurazioni di M è una relazione

binaria $\underset{M}{\rightarrow}$ nell'insieme delle configurazioni di M tale che

$$(q,x,\gamma) \underset{M}{\rightarrow} (q',x',\gamma')$$

se e solo se valgono le tre condizioni:

1. esiste $w \in \Sigma \cup \{\varepsilon\}$ tale che $x = wx'$;
2. esistono $A \in \Gamma$ e $\eta, \zeta \in \Gamma^*$ tali che $\gamma = A\eta$ e $\gamma' = \zeta\eta$;
3. $\delta(q,w,A) = (q',\zeta)$.

Analogamente al caso degli automi a stati finiti, una computazione è allora definita come una sequenza $c_0,\ldots, c_k$ di configurazioni di M tale che $c_i \underset{M}{\rightarrow} c_{i+1}$. La chiusura riflessiva e transitiva di $\underset{M}{\rightarrow}$, indicata con $\overset{*}{\underset{M}{\rightarrow}}$, denota la relazione di raggiungibilità tra configurazioni: avremo cioè che date due configurazioni c_i e c_j, $c_i \overset{*}{\underset{M}{\rightarrow}} c_j$ se e solo se esiste una computazione che inizia con c_i e termina con c_j.

Definizione (Relazione di computazione) [5] La relazione di computazione $\overset{*}{\underset{M}{\rightarrow}}$ è la chiusura transitiva e riflessiva della relazione $\underset{M}{\rightarrow}$ di transizione tra configurazioni.

Una stringa x è accettata da M mediante stato finale se:

$$(q_0,x,Z_0) \overset{*}{\underset{M}{\rightarrow}} (q,\varepsilon,\gamma),\ q \in F.$$

A questo punto siamo in grado di definire il linguaggio riconosciuto da un automa a pila secondo i due modi differenti di accettazione [5]:

$$L_F(M) = \{ x \in \Sigma^* : (q_0,x,Z_0) \overset{*}{\underset{M}{\rightarrow}} (q,\varepsilon,\gamma),\ \gamma \in R^*, q \in F \}$$

$$L_p(M) = \{ x \in \Sigma^* : (q_0,x,Z_0) \overset{*}{\underset{M}{\rightarrow}} (q,\varepsilon,\varepsilon),\ q \in Q \}$$

Facciamo notare che affinché una stringa sia accettata da un automa a pila non deterministico basta che esista un'opportuna sequenza di mosse che conduca ad una configurazione finale. Quindi affinché la parola sia rifiutata non basta che per una particolare scelta delle mosse non si giunga ad una configurazione finale, ma occorre che tutte le possibili scelte portino a configurazioni non finali [3].

2.7 Analisi discendente deterministica: introduzione

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 un 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.

Nell'analisi discendente l'analizzatore è predittivo, nel senso che si espande un non terminale A mediante la regola $A \rightarrow A_1A_2\ldots A_k$; si predice che dovranno essere incontrate successivamente nella stringa da analizzare anche le sottostringhe derivate da $A_2\ldots A_k$.

Quindi l'analisi discendente costruisce una derivazione sinistra dell'albero, partendo dal simbolo distinto ed espandendolo con una produzione opportuna, e così via, fino a visitare tutta la stringa *x*, o a trovare un errore.

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

che permettono di modificare una grammatica in modo da renderla più adatta ad un parsing di questo tipo. Nella seconda parte introdurremo più precisamente il concetto di parser top-down e la costruzione di parsers predittivi mediante tabelle di parsing appropriate utilizzando grammatiche LL(k) [1,5].

2.7.1 Eliminazione della ricorsione a sinistra

Una grammatica si dice ricorsiva a sinistra [1,2,7] se esiste un simbolo non terminale A tale che $A \xrightarrow{+} A\alpha$ per una qualche stringa α. I metodi di parsing top-down non possono trattare grammatiche ricorsive a sinistra, quindi spesso si ha bisogno di una trasformazione che le eliminini.

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

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

Un albero di derivazione che usa queste regole sarà tutto rivolto a sinistra e genererà una stringa della forma βαα…α. Si noti il modo in cui viene generata questa stringa: al primo passo viene generata l'ultima (a destra), cioè α, al secondo passo la penultima e così via fino all'ultimo passo, in cui viene generato β iniziale. Questo procedimento è raffigurato nella figura 5.

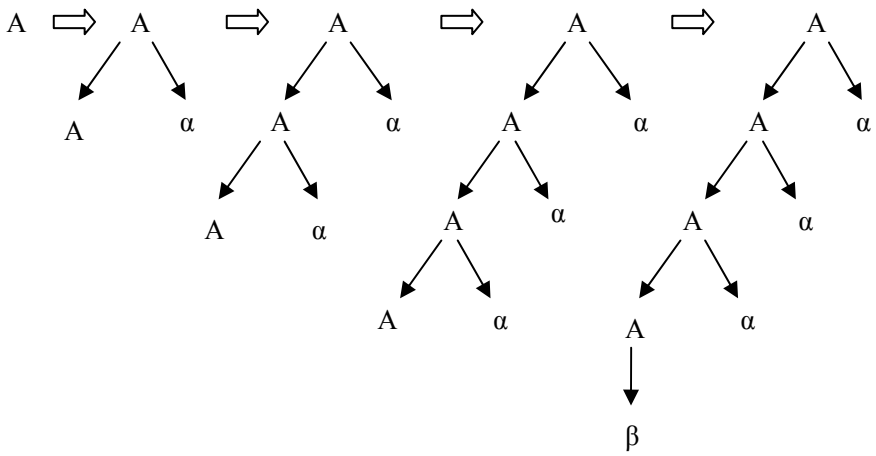


Figura 5 Generazione di una stringa con produzioni ricorsive a sinistra

Un modo alternativo per ottenere la stessa stringa è quello di generare dapprima β e poi via via tutte le α procedendo verso destra. Le produzioni che descrivono questo procedimento sono le seguenti:

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

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

Esempio Consideriamo la grammatica:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

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

$$E \rightarrow TE'$$
$$E' \rightarrow + TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow * FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid id$$

□

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 nella grammatica si raggruppano le produzioni secondo il seguente schema:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$$

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

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

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 deduce dalle produzioni).

Si consideri, ad **esempio**, la seguente grammatica:

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \varepsilon$$

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 procedere in questo modo:

1. si mettono i simboli non terminali in un ordine qualsiasi: $A_1, A_2, \ldots, A_n$
2. per ogni non terminale A_i, con $1 \leq i \leq n$,
3. si ripete per ogni $j < i$, con $1 \leq j \leq i-1$,
    - rimpiazza ogni produzione della forma $A_i \rightarrow A_j\gamma$, con le produzioni $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ se $A_j \rightarrow \delta_1 \mid \ldots \mid \delta_k$ sono le produzioni correnti per A_j;

Si noti che questo metodo funziona sempre per grammatiche prive di cicli (cioè situazioni del tipo $A \xrightarrow{+} A$) e di ε-produzioni (cioè produzioni del tipo $A \rightarrow \varepsilon$).

Consideriamo la grammatica:

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \varepsilon$$

Tecnicamente non è garantito che l'algoritmo sopra descritto funzioni, poiché la grammatica contiene ε-produzioni, 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.

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

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

e togliendo la ricorsione a sinistra immediata si ottiene la seguente grammatica finale:

$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid A'$$
$$A' \rightarrow cA' \mid adA' \mid \varepsilon$$

Questa grammatica non presenta ricorsione a sinistra non immediata, e nemmeno immediata.

In una grammatica possono però presentarsi anche ricorsioni indirette, ad esempio come nella seguente grammatica:

$$A \rightarrow Bcd$$
$$A \rightarrow dc$$
$$B \rightarrow Cd$$
$$C \rightarrow Add$$

che produce il linguaggio $dc(dddcd)^*$. In questo caso possiamo ottenere lo stesso linguaggio sostituendo l'ultima produzione con le produzioni:

$$C \rightarrow B\mathit{cddd}$$
$$C \rightarrow \mathit{dcdd}$$

Abbiamo quindi eliminato la ricorsione sinistra indiretta su A, ma ne abbiamo introdotta una su B. Per eliminare questa, sostituiamo la produzione $C \rightarrow B\mathit{cddd}$ con:

$$C \rightarrow C\mathit{dcddd}$$

Abbiamo ora portato la ricorsione a sinistra indiretta ad essere una ricorsione immediata su C, che possiamo eliminare secondo il metodo precedentemente considerato. Otteniamo quindi l'insieme delle produzioni finali:

$$A \rightarrow B\mathit{cd}$$
$$A \rightarrow \mathit{dc}$$
$$B \rightarrow C\mathit{d}$$
$$C \rightarrow \mathit{dcdd}D$$
$$D \rightarrow \mathit{dcddd}D \mid \varepsilon$$

che produce lo stesso linguaggio.

Si noti che in questo caso la forma tipica di una derivazione è:
$A \rightarrow B\mathit{cd} \rightarrow C\mathit{dcd} \rightarrow \mathit{dcdd}D\mathit{dcd} \underbrace{\rightarrow \ldots \rightarrow}_{n-1} \mathit{dcdd}(\mathit{dcddd})^{n-1}D\mathit{dcd} \rightarrow$
$\mathit{dcdd}(\mathit{dcddd})^{n-1}\mathit{dcd} = \mathit{dc}(\mathit{dddcd})^*$ [1,2,7,14].

2.7.2 Fattorizzazione a sinistra

Questa trasformazione [1,2,7,13] è utile per ottenere grammatiche

per le quali è possibile definire parsers 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.

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 $\alpha\beta_2$. Quello che possiamo fare, però, è espandere A in αA' 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 o β_2 generano stringhe che iniziano in maniera differente.

Vediamo un metodo generale che ci permette di fattorizzare a sinistra le produzioni di una grammatica G e preservare il linguaggio generato dando come risultato una grammatica equivalente a G fattorizzata a sinistra.

Per ogni non terminale A bisogna trovare il prefisso α più lungo delle parti destre delle produzioni di A. Se $\alpha \neq \varepsilon$ (cioè α è un prefisso non banale comune) allora la fattorizzazione a sinistra rimpiazza tutte le produzioni $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_k$ (dove nessun γ_i inizia con α) con:

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_k$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

dove A' è un simbolo non terminale nuovo.

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

Esempio Data la grammatica G:

$$E \rightarrow T + E \mid T - E \mid T$$
$$T \rightarrow a \mid b$$

si può notare che le produzioni aventi E come parte sinistra richiedono la seguente fattorizzazione:

$$E \rightarrow TE'$$
$$E' \rightarrow + E \mid - E \mid \varepsilon$$
$$T \rightarrow a \mid b$$

questa grammatica, ovviamente, è equivalente a quella data.

□

2.7.3 Parser Top-Down

In questo paragrafo introduciamo le idee di base per il parsing top-down e mostriamo come costruire un tipo efficiente di parser top-down: le grammatiche LL(1).

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 S della grammatica. In una generica configurazione il parser sceglie il nodo etichettato con un non terminale che si trovi più a sinistra nell'albero parziale. Una volta selezionato un nodo il parser si deve basare sulla stringa di input per decidere come espanderlo. Per espansione si intende la selezione 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. Il processo continua fino a quando non si ottiene un albero di derivazione per la stringa data oppure si fallisce. 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 esiste un albero di derivazione 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.

Ciò appare evidente 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 proprie 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 effettuando una scelta diversa (ove possibile) è in genere denominata "backtracking".

È evidente che un parser che fa backtracking può essere molto inefficiente (nel caso peggiore ha bisogno di operare tutte le scelte possibili per ogni nodo non terminale).

Un parser molto efficiente, d'altra parte, riesce ad "indovinare", ad ogni passo, 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 parsers top-down ci soffermeremo ancora sull'aspetto generale, per soffermarci, quindi, sul 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 generale, 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* (avanscoperta).

Consideriamo come **esempio** la grammatica contenente le sole regole S → *a*S*b*, S → *ab*, la quale genera il linguaggio $a^n b^n$, per $n > 0$, e procediamo all'analisi top-down della stringa *aabb*.

Per fare questo utilizziamo un automa a pila non deterministico con un solo stato, con alfabeto di pila {*a*,*b*,S}, con S simbolo iniziale di pila, e ammatiamo che l'accettazione della stringa sia per pila vuota. La funzione di transizione è data dalle seguenti regole:

$$\delta(q,\varepsilon,S) = \{(q,aSb),(q,ab)\}$$
$$\delta(q,a,a) = \{(q,\varepsilon)\}$$
$$\delta(q,b,b) = \{(q,\varepsilon)\}$$

(le situazioni non indicate sono di errore e causano l'arresto della macchina).

Una computazione che accetti la stringa *aabb* procederà attraverso le seguenti configurazioni:
(q,*aabb*,S) → (q,*aabb*,*a*S*b*) → (q,*abb*,S*b*) → (q,*abb*,*abb*) → (q,*bb*,*bb*) → (q,*b*,*b*) → (q,ε,ε)

Si osservi come questa computazione corrisponda esattamente alla derivazione S → *a*S*b* → *aabb* nella grammatica, se si considerano i passi in cui S appare in cima alla pila.

Vi è, però, un problema connesso al fatto che non siamo in grado di stabilire ad ogni passo se applicare la transizione corrispondente alla coppia (q,*a*S*b*) o quella corrispondente alla coppia (q,*ab*). D'altra parte possiamo osservare che la conoscenza dei primi due simboli nella parte restante di input ci darebbe la certezza di quale sia la scelta corretta. Infatti, se la parte restante di input inizia con aa, sappiamo che deve essere stata prodotta almeno un'altra S, mentre se questa parte inizia con *ab*, sappiamo che non possono essere state prodotte altre S. Modifichiamo quindi il funzionamento del nostro automa a pila in modo da poter leggere, senza eliminarli, due simboli di *lookahead*.

Riscriviamo quindi il comportamento dell'automa a pila come segue:

$$\delta(q,\varepsilon,S,\{aa\}) = (q,aSb)$$
$$\delta(q,\varepsilon,S,\{ab\}) = (q,ab)$$
$$\delta(q,a,a) = \{(q,\varepsilon)\}$$
$$\delta(q,b,b) = \{(q,\varepsilon)\}$$

dove gli elementi fra parentesi graffe indicano la stringa che deve essere letta in *lookahead* nella parte restante di input perché la transizione sia effettuata. Per le transizioni dove il *lookahead* non è necessario, si sottintende che esse avvengono con qualsiasi sequenza in *lookahead* (ovviamente compatibile con il fatto che il simbolo iniziale di questa sequenza viene eliminato nella transizione). Si osservi che in questo caso la presenza di ε non induce al non determinismo [1,3,4,7,8,14,15].

2.7.4 Grammatiche LL(k)

Intuitivamente, possiamo dire che le grammatiche LL(k) hanno le

seguenti caratteristiche [13]:

1. consentono un riconoscimento deterministico top-down ripercorrendo, in pratica, la derivazione canonica sinistra mediante una scansione della stringa da riconoscere una sola volta da sinistra a destra;
2. la produzione corretta da applicare si può scegliere guardando, ad ogni passo, k simboli della stringa da analizzare a destra di quelli già derivati e utilizzando l'informazione acquisita nei passi precedenti.

Formuliamo ora una notazione più precisa di grammatica LL(k).

Siano G = (V, Σ, P, S) una grammatica non ambigua e *w* una qualsiasi stringa di L(G).

Dovrà allora esistere un'unica derivazione canonica sinistra:

$$S = \alpha_0 \rightarrow \alpha_1 \rightarrow \ldots \rightarrow \alpha_n = w$$

L'analizzatore dovrebbe ricostruire i passi di questa derivazione partendo da S e per fare ciò dovrebbe decidere, ad ogni passo, quale produzione applicare, cioè come espandere il non terminale più a sinistra.

Si supponga allora di essere arrivati all'i-esimo passo:

$$S = \alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_{i-1} \rightarrow \alpha_i = x A \beta$$

dove:

x è la parte già scandita della stringa *w*;

A è il non terminale più a sinistra della forma di frase α_i;

β è il resto della forma di frase α_i.

Diremo che la grammatica G è LL(k) se è possibile decidere come espandere A (cioè quale produzione $A \rightarrow \alpha$ applicare) guardando, al più, k

simboli a sinistra della stringa ancora da esaminare e sfruttando le informazioni costituite da A e β (vedi figura 6) [13].

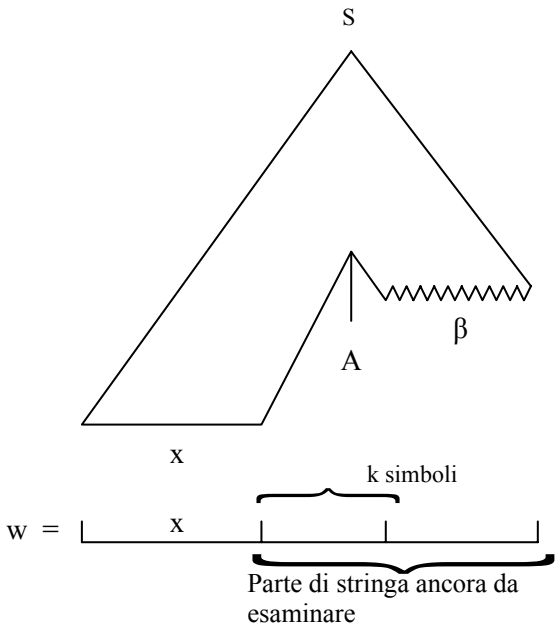


Figura 6 Analisi LL(k)

Definizione (Grammatica LL(k) [3,4] Una grammatica context free G è LL(k), con $k \geq 1$, se esistono le due derivazioni:

$$S \xrightarrow{*} xA\alpha \rightarrow x\beta\alpha \xrightarrow{*} xy$$

$$S \xrightarrow{*} xA\alpha \rightarrow x\gamma\alpha \xrightarrow{*} xz$$

e se $y/k = z/k$; allora si conclude che $\beta = \gamma$.

Diremo che una grammatica è LL se è LL(k) per qualche valore di k. I linguaggi sono quelli generati da grammatiche LL [13].

Se è vero che i linguaggi generati dalle grammatiche LL(k) sono riconoscibili in modo deterministico, è anche vero che, da un punto di vista

pratico, la necessità di memorizzare k simboli, l'esigenza di una scelta (fatta per confronto di stringhe lunghe k) delle produzioni da applicare e la dipendenza dalla parte di analisi già effettuata vanno ad aggravare il tempo di analisi e sull'occupazione di memoria, rendendo così di scarso interesse le grammatiche LL(k) con k > 1. È stato comunque appurato che per definire la sintassi dei linguaggi è sufficiente ricorrere a grammatiche LL(1) che consentono un'analisi molto più efficiente oltre che più semplice: è questa la ragione che ci spinge a dedicarci ad esse in modo più dettagliato [4,13].

2.7.5 Grammatiche LL(1)

Il termine LL(1) ha il seguente significato: la prima L indica che l'input è analizzato da sinistra verso destra; la seconda L sta a significare che il parser effettua una derivazione leftmost per la stringa di input; il numero 1 indica che si legge soltanto un solo simbolo della stringa per risolvere le scelte del parser. Per definizione, una grammatica LL(1) non è ambigua.

Nelle grammatiche LL(1) siamo quindi interessati a derivazioni che hanno la forma

$$S \rightarrow_{lm} X_1 X_2 \ldots X_n \rightarrow^*_{lm} uA\gamma \rightarrow_{lm} u\alpha\gamma \rightarrow^*_{lm} uavx = w$$

in cui, cioè, il primo simbolo incontrato dopo la stringa u è il terminale a.

Tale simbolo, nel caso la grammatica sia LL(1), deriva da α se α non è annullabile, mentre deriva da γ se α viene trasformato in ε.

Introduciamo ora due nuove funzioni che ci permetteranno di svolgere l'analisi delle grammatiche LL(1) [4,5].

2.7.6 Funzione First

La funzione First:$(\Sigma \cup V)^* \rightarrow P(\Sigma \cup \{\varepsilon\})$ [1,3,10,13,14,15] permette di definire quale sia il primo simbolo terminale prodotto a partire da una stringa X. Essa include nell'insieme prodotto anche il valore ε nel caso in cui X sia esattamente la stringa ε, ovvero sia composta da tutti simboli non terminali annullabili, cioè i simboli Y_i che possono dare origine ad una derivazione per $Y_i \xrightarrow{*} \varepsilon$.

In altre parole, 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 inoltre $\alpha \xrightarrow{*} \varepsilon$, allora anche $\varepsilon \in \text{First}(\alpha)$.

Per calcolare First(X) per ogni simbolo grammaticale X (terminale o non terminale) bisogna applicare le seguenti quattro regole per ogni terminale fino a quando nessun altro simbolo o ε può essere aggiunto ad uno qualsiasi degli insiemi First:

1. $\text{First}(a) = \{a\} \quad \forall a \in \Sigma$;
2. $\text{Firs}(\varepsilon) = \{\varepsilon\}$
3. $\text{First}(X) = \bigcup_{\{X \rightarrow \alpha \in P\}} \text{First}(\alpha) \quad \forall X \in V$;
4. $\text{First}(X_1 \ldots X_n) = \left[\bigcup_{\{i<k\}} \left(\text{First}(X_i) - \{\varepsilon\} \right) \right] \cup \text{First}(X_k)$

   per k massimo intero tale che: $X_1 \ldots X_{k-1} \xrightarrow{*} \varepsilon$ oppure k = n

2.7.7 Funzione Follow

Per un non terminale X l'insieme Follow(X) (definito come la funzione Follow:$V \rightarrow P(\Sigma)$) [1,3,10,13,14,15] contiene i simboli terminali *w*

che possono apparire immediatamente alla destra di X in una forma sentenziale, cioè l'insieme dei *w* tali che esiste una derivazione $S \xrightarrow{*} \alpha X w \beta$ per qualche α e β. Si noti che, ad un certo punto della derivazione, potrebbero esserci dei simboli non terminali tra X e *w* che poi vengono riscritti in ε se da essi deriva ε. Inoltre, se X è il simbolo più a destra in una forma sentenziale, allora anche il simbolo speciale $ appartiene a Follow(X).

Data una grammatica G = (V, Σ, P, S), la funzione Follow per ogni $X \in V$ è definita ricorsivamente come segue:

1. se S è il simbolo distinto allora $\$ \in \text{Follow}(S)$;
2. $$\text{Follow}(X) = \left\{ \bigcup_{\{Y \to \alpha X \beta\}} (\text{First}(\beta) - \{\varepsilon\}) \right\} \cup \left\{ \bigcup_{\{Y \to \alpha X \beta \mid \beta \to^* \varepsilon\}} \text{Follow}(Y) \right\} \cup \left\{ \bigcup_{\{Y \to \alpha X\}} (\text{Follow}(Y) \right\} \quad \forall X \in V$$

Alla luce di ciò si possono fare alcune osservazioni. Quando il simbolo iniziale non compare a destra di nessuna produzione, $ è l'unico simbolo nel suo insieme Follow. In generale l'insieme Follow, oltre ad essere definito per i soli non terminali, non contiene mai ε.

2.7.8 Costruzione della tabella per le grammatiche LL(1)

La proprietà LL(1) può quindi essere riformulata come segue: per ogni coppia di produzioni $A \to \alpha$, $A \to \beta \in P$, deve valere:

1. $\text{First}(\alpha) \cap \text{First}(\beta) = \{\ \}$
2. $\text{First}(\alpha) \cap \text{Follow}(A) = \{\ \}$ se $\varepsilon \in \text{First}(\beta)$

Si noterà, infatti, che questa proprietà considera sia il caso che né α né β siano annullabili, sia il caso che uno dei due lo sia. Ovviamente perché la grammatica sia LL(1) occorre che non siano entrambi annullabili.

Nel caso che una grammatica sia LL(1), si può procedere alla costruzione di un parser top-down deterministico che accetti stringhe utilizzando gli insiemi First e Follow ed una tabella che indichi al programma di parsing il lato destro della regola la cui applicazione va simulata nel corso di una derivazione sinistra. Sostanzialmente la tabella per una grammatica LL(1) è data dal prodotto cartesiano tra i non terminali (righe) e i terminali (colonne) della grammatica presa in considerazione. La sua costruzione è data dalle seguenti regole:

1. per ogni regola $X \rightarrow \gamma$ della grammatica G si inserisce nella casella *(X,a)*, con *a* simbolo terminale, la regola[4] $X \rightarrow \gamma$ per ogni *a* tale che $a \in \text{First}(\gamma) \setminus \{\varepsilon\}$;
2. per ogni regola $X \rightarrow \gamma$ della grammatica G, per cui $\varepsilon \in \text{First}(\gamma)$, si inserisce nella casella *(X,a)* la regola $X \rightarrow \gamma$ per ogni *a* tale che $a \in \text{Follow}(\gamma)$.

Una grammatica che genera una tabella con più di un elemento in corrispondenza di qualche *(X,a)* è ambigua e di conseguenza non è LL(1).

Le grammatiche ambigue corrispondenti sono dette LL(k). Le caselle della tabella lasciate vuote sono quelle in corrispondenza di situazioni di errore [1,14,15,16].

Esempio Consideriamo la grammatica

$$S \rightarrow xSy \mid FS \mid \varepsilon$$
$$F \rightarrow Fv \mid u$$

[4] Le regole grammaticali, in questo caso, vengono numerate.

Si può notare che tale grammatica non è LL(1) in quanto esiste una produzione in cui c'è la ricorsione sinistra, che pertanto va eliminata. Quindi la produzione $F \rightarrow Fv \mid u$ viene rimpiazzata con le produzioni:

$$F \rightarrow uF'$$
$$F' \rightarrow vF' \mid \varepsilon$$

La nuova grammatica è:

$S \rightarrow xSy \mid FS \mid \varepsilon$	0 | 1 | 2
$F \rightarrow uF'$	3
$F' \rightarrow vF' \mid \varepsilon$	4 | 5

Per verificare se la grammatica è LL(1) dobbiamo vedere se vengono soddisfatte le proprietà LL(1) e pertanto calcolare, come prima cosa, gli insiemi First e Follow:

First	**Follow**
$Fisrt(xSy) = \{x\}$	$Follow(S) = \{\$, y\}$
$Fisrt(FS) = \{u\}$	$Follow(F) = \{\$\}$
$Fisrt(\varepsilon) = \{\varepsilon\}$	$Follow(F') = \{\$\}$
$Fisrt(uF') = \{u\}$	
$Fisrt(vF') = \{v\}$	

Ora verifichiamo se valgono le proprietà LL(1):

Proprietà 1	**Proprietà 2**
$First(xSy) \cap First(FS) = \{\ \}$	$First(xSy) \cap Follow(S) = \{\ \}$
$First(FS) \cap First(\varepsilon) = \{\ \}$	$First(FS) \cap Follow(S) = \{\}$
$First(xSy) \cap First(\varepsilon) = \{\ \}$	$First(vF') \cap Follow(F') = \{\}$
$First(vF') \cap First(\varepsilon) = \{\ \}$	

Visto che tutte le intersezioni hanno dato, come risultato, un insieme vuoto, allora possiamo dedurre che il linguaggio ottenuto dalla grammatica ammette un riconoscitore LL(1).

Vediamo, quindi, come risulta la tabella 3, relativa alla grammatica in esame.

	X	**y**	**u**	**v**	**$**
S	0		1		2
F			3		
F'				4	5

Tabella 3. Grammatica LL(1)

2.8 Analisi ascendente deterministica: introduzione

Gli analizzatori ascendenti che considereremo sono una speciale classe di automi deterministici, detti a spostamento e a riduzione. Supporremo, nel seguito, che un metodo di analisi sintattica ascendente sia in grado di ottenere l'albero sintattico ricostruendo la stringa con l'uso della derivazione destra. Se $S = \beta_0 \rightarrow \beta_1 \rightarrow \ldots \rightarrow \beta_n = \delta$ è la derivazione destra della stringa δ, l'analisi sintattica ascendente consiste nel determinare la produzione che permette di passare da β_{i+1} a β_i per $i = n-1, \ldots, 0$.

Se $\beta_i = \alpha A \mu$ e $\beta_{i+1} = \alpha\gamma\mu$, si dice che γ viene ridotto ad A e che γ è la parte riducibile di β_{i+1}. Si noti che, essendo la derivazione destra, μ non contiene alcun non terminale. Per determinare ad ogni passo della derivazione la parte riducibile, si utilizza un metodo che scandisce i caratteri della stringa da analizzare da sinistra a destra e li sposta in cima alla pila,

finché sulla pila non compare la parte riducibile, cui viene allora sostituita la parte sinistra. Ciò significa che si ha un passo di riduzione se c'è una sottostringa che fa *match* con la parte destra di una produzione grammaticale; tale sottostringa viene sostituita dal simbolo a sinistra di tale produzione producendo così una derivazione rightmost rovesciata. Questo metodo si ripete finché non è più possibile procedere o perché la stringa non appartiene al linguaggio considerato (si è, cioè, rilevato un errore) o perché tutta la stringa è stata visitata e sulla pila è presente solo il simbolo iniziale. In tal caso la stringa è accettata o riconosciuta. Il procedimento risulta deterministico purché non vi sia più di una parte riducibile sulla pila, condizione che sarà garantita per le grammatiche LR(k) [5].

2.8.1 Parser Bottom-up

Una tecnica di analisi sintattica bottom-up, usata per fare il parsing di gran parte delle grammatiche libere dal contesto, è LR(k): L indica che l'input è analizzato da sinistra verso destra; R indica che nel processo di parsing viene prodotta una derivazione rightmost; il numero k indica quanti simboli di *lookahead*[5] vengono letti. I parsers LR sono usati per varie ragioni, ad esempio possono essere utili per riconoscere tutti i linguaggi costruiti a partire da grammatiche libere dal contesto.

[5] Il *lookahead* indica la quantità di simboli che bisogna leggere per distinguere una produzione grammaticale da tutte le altre. Di conseguenza, viene usato dal parser per decidere quale produzione applicare.

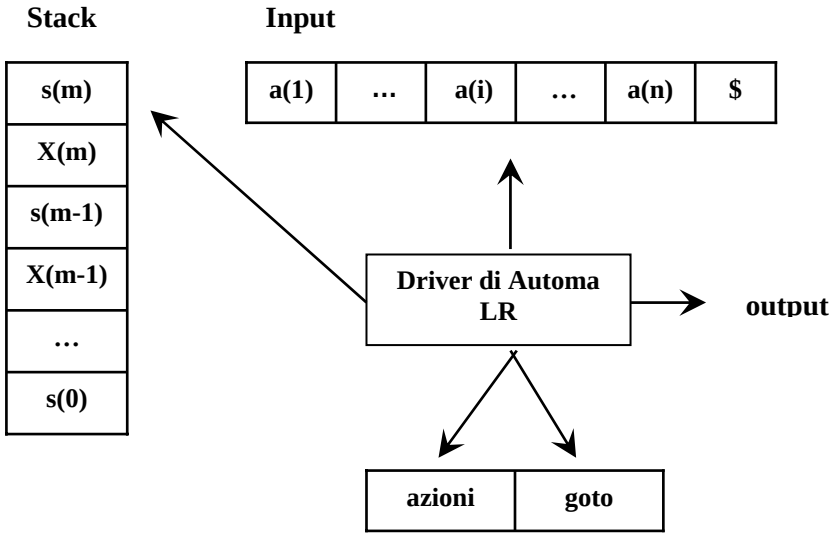


Figura 7. Modello per i parsers LR

La struttura di un parser LR è quella rappresentata nella figura 7: il parser è costituito da un input (insieme di caratteri), un output (risultato del parsing), uno stack, un driver di automa e una tabella di parsing divisa in due parti (*azioni* e *goto*). Il driver di automa è lo stesso per ogni parsing LR; ciò che cambia tra i vari parsing è solo la tabella di parsing. Il driver di automa legge, uno per volta, dei caratteri dal buffer di input ed usa lo stack per memorizzare stringhe della forma $s_0X_1s_1X_2s_2\ldots X_ms_m$, dove s_0 è lo stato iniziale, l'unico presente all'inizio della computazione sullo stack, mentre s_m è lo stato in cima allo stack. Ogni X_i è un simbolo grammaticale mentre ogni s_i è un simbolo che indica uno stato. Ogni stato descrive l'informazione contenuta nello stack prima di esso; la combinazione tra lo stato in cima allo stack e il simbolo in input corrente viene usata per indicizzare la tabella di parsing e per determinare le decisioni di *shift* e *reduce* del parser.

Come già detto, la tabella è composta da due parti.

Per la parte relativa alle *azioni* (*action*) all'interno della tabella, le azioni possibili sono quattro:

1. l'azione di *shift*, dove il successivo simbolo di input viene spostato in cima allo stack;
2. l'azione di *reduce*, dove il parser sceglie la h-esima[6] regola grammaticale $X \rightarrow \alpha_1 \ldots \alpha_k$, estrae ed elimina (operazione equivalente ad un'azione di pop) la sequenza $\alpha_1 s_{h+1} \ldots \alpha_k s_{h+k}$ dallo stack, calcola, dal simbolo X della regola grammaticale e dai k simboli (*lookahead*) all'inizio della stringa di input, il prossimo stato s_m a partire dallo stato s_h in testa alla pila, e inserisce sullo stack Xs_m;
3. l'azione di *accept*, dove il parsing segnala di aver terminato con successo;
4. l'azione di *error*, dove il parsing segnala di aver terminato con un fallimento.

La funzione *goto*, invece, prende uno stato e un simbolo grammaticale come argomenti al fine di produrre un nuovo stato. È la funzione che viene utilizzata nell'azione di *reduce* al fine di calcolare il nuovo stato da inserire sullo stack.

Una *configurazione* di un parser LR è una coppia in cui il primo componente è il contenuto dello stack mentre il secondo componente è l'input restante da analizzare:

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m , a_i a_{a+1} \ldots a_n \$)$$

Il simbolo \$ indica sempre la fine della stringa. Questa configurazione rappresenta la stringa

[6] Le regole grammaticali, in questo caso, vengono numerate.

$$X_1X_2\ldots X_m a_i a_{a+1}\ldots a_n \$$$

Per calcolare la tabella LR bisogna, a questo punto, chiarire alcuni concetti.

L'*insieme dei contesti sinistri* per un simbolo $X \in V$ è definito come l'insieme $LC(X) = \{\alpha \mid \exists w \in L(G),\ S \rightarrow^*_{rm} \alpha X y \rightarrow_{rm} \alpha X_1\ldots X_m y \rightarrow^*_{rm} w\}$, cioè l'insieme dei prefissi delle forme sentenziali per le derivazioni destre delle stringhe L(G) che possono precedere un'occorrenza di X nel momento in cui a tale occorrenza viene applicata una regola (e quindi tale occorrenza costituisce il simbolo non terminale più a destra).

Definiamo per ogni produzione $X \rightarrow \alpha\beta \in P$ l'insieme di elementi (*item*) $I = \{[X \rightarrow \alpha\bullet\beta]\}$ mettendo, quindi, un punto nella parte destra della produzione.

Ad **esempio**, per la produzione $T \rightarrow Sb$ si avrebbero gli item $[T \rightarrow \bullet Sb]$, $[T \rightarrow S\bullet b]$, $[T \rightarrow Sb\bullet]$.

Si può notare che ogni item corrisponde a una produzione della grammatica G. Un item della forma $[X \rightarrow \alpha\bullet Y\beta]$ con $Y \in V$ indica che si sta avanzando verso la costruzione di un contesto sinistro per la produzione $X \rightarrow \alpha Y\beta$ e che per poter arrivare a completare questa costruzione, e quindi a raggiungere l'item $[X \rightarrow \alpha Y\beta\bullet]$, occorre riuscire ad ottenere un item della forma $[Y \rightarrow \gamma\bullet]$, con $Y \rightarrow \gamma \in P$. Tale item è detto item completo in quanto permetterà di costruire prima, ed eliminare poi, γ in pila, mettendo al suo posto il simbolo Y. Si avrà quindi un prefisso del contesto sinistro della

produzione $X \rightarrow \alpha Y \beta$, rappresentato da un item della forma $X \rightarrow \alpha Y \bullet \beta$, e si potrà proseguire nella costruzione in pila della parte restante, β [1,2,13,14,15,16].

2.8.2 Automa di controllo per il parsing LR

Diamo quindi la tecnica (algoritmo) per la costruzione dell'automa di controllo nel caso in cui il parser sia in grado di leggere al più un simbolo di *lookahead*, cioè per $k \leq 1$. Questa tecnica richiede che siano numerate le regole della grammatica a partire da 1 e che siano introdotti un nuovo simbolo non terminale fittizio S' e una regola $S' \rightarrow S$, che viene numerata con 0. Questo accorgimento permette di far coincidere l'accettazione con la simulazione dell'applicazione della regola $S' \rightarrow S$, in modo da evitare possibili ambiguità connesse all'eventuale presenza di S nel lato destro di qualche produzione.

Gli stati dell'automa sono tutti gli elementi creabili a partire dalla grammatica G.

Definizione (Transizione) [1,14] Una transizione dell'automa è definita come segue: dato l'elemento $A \rightarrow \alpha \bullet X \beta$, allora esiste una transizione, etichettata X, dallo stato $A \rightarrow \alpha \bullet X \beta$ allo stato $A \rightarrow \alpha X \bullet \beta$.

Si osservi che:

- se X è un simbolo terminale, questa transizione consiste nel fare uno *shift* del simbolo sull'input in cima allo stack;
- se X è un simbolo non terminale, questa transizione consiste nel mettere X in testa alla pila. X, essendo non terminale, non è presente

nell'input ma sicuramente sull'input vi sarà presente una stringa derivabile da X stesso. Per riconoscere questa stringa occorre riconoscere un elemento del tipo $X \rightarrow \bullet\gamma$. In questo caso si è di fronte ad una *reduce*.

Abbiamo, inoltre, bisogno di definire due operazioni: *chiusura* e *goto*.

Definizione (Chiusura/closure) [1,14] Se I è un insieme di elementi per una grammatica G, allora *closure(I)* è l'insieme di elementi costruiti a partire da I attraverso le seguenti due regole:

1. inizialmente tutti gli elementi in I sono aggiunti a *closure(I)*;
2. se $A \rightarrow \alpha\bullet X\beta$ è in *closure(I)* e $X \rightarrow \gamma$ è una produzione, allora si aggiunge in *closure(I)* l'elemento $X \rightarrow \bullet\gamma$, se questo non è già presente. Questa regola si applica finché nessun nuovo elemento viene aggiunto da *closure(I)*.

Definizione (Goto) [1,14] L'operazione di *goto*, che prende come argomenti un insieme I di elementi e un simbolo grammaticale X, è definita come segue:

$$goto(I,X) = closure(\{A \rightarrow \alpha X\bullet\beta \mid \forall\ A \rightarrow \alpha\bullet X\beta \in I\})$$

2.8.3 Costruzione della tabella per le grammatiche LR(0)

Date tutte queste informazioni, possiamo ora costruire due tabelle, dette rispettivamente tabella *action* e tabella *goto*, nel seguente modo:

1. per ogni stato *i* si costruisce una riga etichettata con *i* per entrambe le tabelle;
2. per ogni simbolo terminale della grammatica, incluso il simbolo $ di fine stringa, si costruisce una colonna della tabella *action*;
3. per ogni simbolo non terminale della grammatica, escluso il simbolo fittizio S', si costruisce una colonna nella tabella *goto*;
4. per ogni terminale *a*, se lo stato *i* contiene un item [A → α•*a*β] allora l'azione da compiere è quella di inserire, in posizione (*i*,*a*), la scritta *shift*/j nella tabella action, dove j indica lo stato di arrivo dopo aver letto il simbolo *a*;
5. per ogni non terminale X, se lo stato *i* contiene un item [A → α•Xβ], allora l'azione da compiere è quella di inserire, in posizione (*i*,X), la scritta *goto*(j) nelle tabella *goto*; dove j indica lo stato di arrivo dopo aver letto il simbolo X;
6. ogni stato contenente un elemento LR(0) del tipo [X → γ•] deve avere un'operazione di riduzione (*reduce*), indicata con *ri*, dove *i* identifica la regola X → γ presa in considerazione (tutte le regole grammaticali devono essere numerate). Poiché il parser è LR(0), la riduzione deve essere fatta per ogni terminale della grammatica;
7. l'operazione di *reduce* [S' → α•, $], quando in testa alla pila c'è $, equivale ad accettazione. Di conseguenza, supponendo che l'elemento sia nello stato k, non bisogna effettuare la *reduce* ma inserire, in posizione (*i*,$), un *acc*;
8. tutte le celle vuote della tabella sottintendono una situazione di errore.

Esempio Consideriamo la grammatica G:

$$S \rightarrow aT$$
$$T \rightarrow Sb$$
$$T \rightarrow b$$

che genera il linguaggio $\{a^n b^n \mid n > 0\}$ e procediamo alla costruzione dell'automa di parsing.

- Stato0 = closure[S' → **•**S] = {[S' → **•**S], [S → **•***a*T]}
- Stato0 $\xrightarrow{S}$ Stato1 = {[S' → S**•**]}
- Stato0 $\xrightarrow{a}$ Stato2 = {[S → *a***•**T], [T → **•**S*b*], [T → **•***b*], [S → **•***a*T]}
- Stato2 $\xrightarrow{T}$ Stato3 = {[S → *a*T**•**]}
- Stato2 $\xrightarrow{S}$ Stato4 = {[T → S**•***b*]}
- Stato2 $\xrightarrow{b}$ Stato5 = {[T → *b***•**]}
- Stato2 $\xrightarrow{a}$ Stato2
- Stato4 $\xrightarrow{b}$ Stato6 = {[T → S*b***•**]}

Possiamo osservare che gli stati di riduzione, cioè quelli che contengono item completi, sono lo Stato3, lo Stato5 e lo Stato6. In tali stati l'item completo è anche l'unico item presente, quindi l'unica azione possibile è quella di riduzione, indipendentemente dal simbolo in lettura del *lookahead*. Si osservi, inoltre, che ognuno degli stati prodotti corrisponde a una viable-prefixes per un certo insieme di linguaggi di contesti sinistri, mentre gli stati di riduzione corrispondono a linguaggi di contesti sinistri per le corrispondenti produzioni della grammatica [1,2,13,14,15,16].

Definizione (Viable-prefixes) [8,14] È l'insieme dei prefissi di una forma sentenziale destra che possono comparire sullo stack di un parsing *shift-reduce*.

Proprietà (Conflitti LR) Nella tabella degli *action*, ogni casella ha al più un'indicazione di azione (*shift* o *reduce*) se e solo se tutti gli insiemi di item che contengono item completi, dello stesso stato, sono dei singleton, cioè se compare un unico item e l'item è completo. In caso contrario si avranno dei conflitti:

- *shift-reduce*, se sono presenti item della forma $[X \rightarrow \alpha_{\bullet}\beta]$, $[Y \rightarrow \alpha_{\bullet}] \in Z_k$ e $First(\beta) \cap Follow(Y) \neq \{\}$;
- *reduce/reduce*, se sono presenti item della forma $[X \rightarrow \gamma_{\bullet}]$, $[Y \rightarrow \beta_{\bullet}] \in Z_k$ e $Follow(X) \cap Follow(Y) \neq \{\}$.

Z_k è l'insieme degli stati.

Se la costruzione della tabella non dà origine a conflitti, diremo che la grammatica è LR(0).

Vediamo allora come risultano le tabelle *action* e *goto* per la grammatica considerata.

Useremo qui la convenzione di presentare in un'unica tabella entrambe le tabelle *action* e *goto*, separate mediante una colonna nera, di abbreviare la parola *shift* con s e la parola *reduce* con r, e di mettere nelle celle della tabella *goto* solo il numero dello stato di arrivo.

	A	**B**	**$**		**S**	**T**
0	s/2				1	
1			acc			
2	s/2	s/5			4	3
3	r1	r1	r1			
4		s/6				
5	r3	r3	r3			
6	r2	r2	r2			

Tabella 4. Grammatica LR(0)

2.8.4 Costruzione della tabella per le grammatiche SLR(1)

Gli elementi utilizzati dal parser SLR(1) sono quelli LR(0). Per la costruzione delle tabelle si usano gli stessi metodi definiti in LR(0), l'unica cosa che cambia è la costruzione della tabella in relazione alla sola azione *reduce*. Pertanto l'unica regola, relativa alla costruzione della tabella, che cambia rispetto a LR(0) è la numero 7:

- ogni stato contenente un elemento LR(0) del tipo $[X \rightarrow \gamma\bullet]$ deve avere un'operazione di riduzione (*reduce*), indicata con *ri*, dove *i* identifica la regola $X \rightarrow \gamma$ presa in considerazione (tutte le regole grammaticali devono essere numerate). L'operazione di riduzione, per uno stato k, deve essere fatta in ogni cella (k,t) $\forall\, t \in \text{Follow}(X)$, dove t è un simbolo terminale.

Come per le grammatiche LR(0), le grammatiche per cui la tabella

di analisi prodotta dal parser SLR(1) non contiene conflitti in nessuna cella (non ha ambiguità) sono dette grammatiche SLR(1).

Consideriamo la grammatica:

$$S \rightarrow aA$$
$$A \rightarrow aA \mid T$$
$$T \rightarrow aTb \mid \varepsilon$$

che genera il linguaggio $\{a^m b^n \mid m>n\geq 0\}$ e procediamo alla costruzione dell'automa di parsing.

Stato0 = closure[S' → •S] = {[S' → •S], [S → •*a*A]}

Stato0 $\xrightarrow{S}$ Stato1 = {[S' → S•]}

Stato0 $\xrightarrow{a}$ Stato2 = {[S → *a*•A], [A → •*a*A], [A → •T], [T → •*a*T*b*], [T → •]}

Stato2 $\xrightarrow{A}$ Stato3 = {[S → *a*A•]}

Stato2 $\xrightarrow{a}$ Stato4 = {[A → *a*•A], [T → *a*•Tb], [A → •*a*A], [A → •T], [T → •*a*Tb], [T → •]}

Stato2 $\xrightarrow{T}$ Stato5 = {[A → T•]}

Stato4 $\xrightarrow{A}$ Stato6 = {[A → *a*A•]}

Stato4 $\xrightarrow{T}$ Stato7 = {[T → *a*T•b], [A → T•]}

Stato4 $\xrightarrow{a}$ Stato4

Stato7 $\xrightarrow{b}$ Stato8 = {[T → *a*T*b*•]}

Osserviamo che lo Stato2, lo Stato4 e lo Stato7 contengono item completi ma non rappresentano dei singleton. Se applicassimo il criterio di costruzione adottato per la tabella LR(0), nello Stato2 si produrrebbe un conflitto *shift/reduce* sul simbolo *a*, nello Stato4 un conflitto *shift/reduce* sul simbolo *a* e nello Stato7 un conflitto *shift/reduce* sul simbolo *b*. La grammatica non è perciò LR(0).

Pertanto, tenendo presente la regola indicata all'inizio del paragrafo e i conflitti riscontrati, cerchiamo di costruire una tabella senza conflitti a partire dalla grammatica data. A questo fine osserviamo ora la natura di questi tre conflitti.

Il conflitto nello Stato2 si origina sul simbolo *a*. Questo simbolo non può, però, comparire alla destra di T in nessuna derivazione destra e, in realtà, in nessuna derivazione della grammatica. Infatti, se consideriamo l'insieme Follow per T che, essendo calcolato sui lati destri delle produzioni, risulta indipendente dal tipo di derivazione che si vuole considerare, abbiamo Follow(T) = {*b*,$}. Lo stesso si può dire per la situazione di conflitto nello Stato4. Per lo Stato7, in cui il conflitto è sul simbolo *b*, possiamo osservare che Follow(A) = {$}.

Vediamo, quindi, come risulta la tabella Simple LR(1), abbreviato SLR(1), per la grammatica in esame. A tal fine completiamo la costruzione dell'insieme Follow con Follow(S) = {$}.

Come previsto, la tabella così costruita non presenta conflitti. Vale la pena di rilevare che nella tabella LR i conflitti *shift/reduce* non possono prodursi nella colonna per $, in quanto questo simbolo non verrà mai messo nella pila [1,2,13,14,15,16].

	a	**b**	**$**	**S**	**A**	**T**
0	s/2			1		
1			acc			
2	s/4	r5	r5		3	5
3			r1			
4	s/4	r5	r5		6	7
5			r3			
6			r2			
7		s/8	r3			
8		r4	r4			

Tabella 5. Grammatica SLR(1)

2.8.5 Costruzione della tabella per le grammatiche LR(1)

Vi sono situazioni in cui esistono automi a pila deterministici capaci di accettare un linguaggio e che non sono, però, producibili mediante la costruzione della tabella SLR(1).

Esempio Consideriamo la grammatica

$$S \rightarrow Aa \mid bAc \mid dc \mid bda$$
$$A \rightarrow d$$

che produce il linguaggio finito $\{dc\} \cup \{da\} \cup \{bda\} \cup \{bdc\}$, e costruiamo gli stati del parser SLR(1).

Stato0 = closure[S' → **.**S] = {[S' → **.**S], [S → **.**A*a*], [S → **.***b*A*c*],

$[S \rightarrow \bullet dc], [A \rightarrow \bullet d], [S \rightarrow \bullet bda]\}$

$\text{Stato0} \xrightarrow{S} \text{Stato1} = \{[S' \rightarrow S\bullet]\}$

$\text{Stato0} \xrightarrow{A} \text{Stato2} = \{[S \rightarrow A\bullet a]\}$

$\text{Stato0} \xrightarrow{b} \text{Stato3} = \{[S \rightarrow b\bullet Ac], [S \rightarrow b\bullet da], [A \rightarrow \bullet d]\}$

$\text{Stato0} \xrightarrow{d} \text{Stato4} = \{[S \rightarrow d\bullet c], [A \rightarrow d\bullet]\}$

$\text{Stato2} \xrightarrow{a} \text{Stato5} = \{[S \rightarrow Aa\bullet]\}$

$\text{Stato3} \xrightarrow{A} \text{Stato6} = \{[S \rightarrow bA\bullet c]\}$

$\text{Stato3} \xrightarrow{d} \text{Stato7} = \{[S \rightarrow bd\bullet a], [A \rightarrow d\bullet]\}$

$\text{Stato4} \xrightarrow{c} \text{Stato8} = \{[S \rightarrow dc\bullet]\}$

$\text{Stato6} \xrightarrow{c} \text{Stato9} = \{[S \rightarrow bAc\bullet]\}$

$\text{Stato7} \xrightarrow{a} \text{Stato10} = \{[S \rightarrow bda\bullet]\}$

Possiamo rilevare la presenza di conflitti *shift/reduce* nello Stato4 e nello Stato7. Nel primo caso il conflitto si produce sul simbolo *c* che può essere portato in pila nel tentativo di completare l'item $[S \rightarrow dc\bullet]$, ma che fa anche parte dell'insieme Follow per A. Nel secondo caso il conflitto si produce sul simbolo *a* che può essere portato in pila nel tentativo di completare l'item $[S \rightarrow bda\bullet]$, ma che fa anche parte dell'insieme Follow per A, quindi Follow(A) = {*a*,*c*}.

Possiamo però domandarci se l'insieme Follow, che dà indicazioni sui simboli che possono seguire la parte di stringa prodotta da un certo non terminale in una qualsiasi derivazione, sia adeguato alle scelte delle azioni in una derivazione destra. In particolare consideriamo lo Stato4. Esso

contiene l'item [A → *d*•] in quanto lo Stato0 conteneva l'item [A → •*d*]. In altre parole, la scelta di ridurre la produzione A → *d* in questo stato corrisponde a replicare una derivazione S → A*a* → *da*. Ha senso quindi effettuare tale riduzione solo se il simbolo di *lookahead* è il simbolo *a*, mentre la presenza del simbolo *c* indicherebbe che stiamo distinguendo una derivazione S → *dc*. Al contrario, nello Stato7, l'item [A → *d*•] è presente in quanto lo Stato3 conteneva [A → •*d*] per effetto del calcolo della chiusura dell'item [S → *b*•A*c*]. In questo stato stiamo quindi tentando di simulare una derivazione S → *b*A*c* → *bdc*. Ciò è possibile solo se il simbolo in *lookahead* è *c*, mentre la presenza, nello stesso stato, di un simbolo di *lookahead a* indicherebbe che stiamo ricostruendo una derivazione S → *bda*. Si tratta, quindi, di avere un'indicazione più forte di quella fornita dagli elementi dell'insieme Follow, che sia strettamente connessa a cosa può comparire a destra della parte di stringa prodotta da un non terminale in una derivazione destra. Possiamo ragionare in questo modo: a destra di una stringa prodotta dal simbolo S' ci deve essere il simbolo di fine stringa $ e indichiamo ciò modificando la struttura degli item in modo che essi mantengano un'indicazione dell'insieme di simboli di *lookahead* accettabili. Abbiamo, quindi, l'item [S' → •S,{$}]. Ammettiamo ora di avere un item della forma [A → α•Bβ,{σ}], con $\alpha,\beta \in (V \cup \Sigma)^*$, $B \in V$, $\sigma \in P(\Sigma \cup \{\$\})$, e cerchiamo di capire quali solo i terminali che possono seguire la parte di stringa *w* prodotta in una derivazione destra a partire da B, considerato, in questo stato, come il non terminale più a destra nella forma sentenziale. Se β può, a sua volta, produrre dei terminali, questi seguiranno *w*. Se β può produrre la stringa λ, allora *w* potrà essere seguita da qualsiasi terminale in σ. In definitiva costruiremo la chiusura dell'item considerando gli item [B → •γ, {π}], dove π = First(First(β)σ), per ogni regola B → •γ ∈ P.

Ovviamente dovremmo ora costruire la tabella immettendo indicazioni di riduzione secondo il seguente criterio (l'unica regola che cambia rispetto alla costruzione della tabella della grammatica LR(0) è la numero 7):

- ogni stato contenente un elemento LR(1) del tipo [X → γ•, t] deve avere un'operazione di riduzione (*reduce*) indicata con *ri*, dove *i* identifica la regola X → γ presa in considerazione (tutte le regole grammaticali devono essere numerate). Poiché il parser è LR(1), la riduzione dovrà essere fatta soltanto quando il simbolo in testa all'input sarà t.

Si noterà che lo spostamento dell'indicatore di avanzamento • fa semplicemente ereditare al nuovo item l'insieme di *lookahead* dell'item originario. In questo caso, infatti, si sta avanzando verso il completamento di un item, in modo da poter effettuare una riduzione.

L'insieme di *lookahead* serve in questo caso a ricordarci che tale riduzione potrà essere effettuata solo se, al momento del completamento della costruzione in pila del lato destro, avremo in lettura un simbolo corretto, cosa che non può essere modificata durante l'avanzamento.

La chiusura, invece, deve tener conto del fatto che il non terminale su cui si vuole effettuare la riduzione è stato inserito nel contesto di un lato destro.

Per la grammatica data otteniamo quindi la nuova specifica per le grammatiche LR(1).

Stato0 = closure[S' → •S] = {[S' → •S, {$}], [S → •A*a*, {$}],

[S → •*bAc*, {$}], [S → •*bda*, {$}], [S → •*dc*, {$}], [A → •*d*, {*a*}]}

Stato0 $\xrightarrow{S}$ Stato1 = {[S' → S•, {$}]}

Stato0 $\xrightarrow{A}$ Stato2 = {[S → A•*a*, {$}]}

Stato0 $\xrightarrow{b}$ Stato3 = {[S → *b*•A*c*, {$}], [S → *b*•*da*, {$}], [A → •*d*, {*c*}]}

Stato0 $\xrightarrow{d}$ Stato4 = {[S → *d*•*c*, {$}], [A → *d*•, {*a*}]}

Stato2 $\xrightarrow{a}$ Stato5 = {[S → A*a*•, {$}]}

Stato3 $\xrightarrow{A}$ Stato6 = {[S → *b*A•*c*, {$}]}

Stato3 $\xrightarrow{d}$ Stato7 = {[S → *bd*•*a*, {$}], [A → *d*•, {*c*}]}

Stato4 $\xrightarrow{c}$ Stato8 = {[S → *dc*•, {$}]}

Stato6 $\xrightarrow{c}$ Stato9 = {[S → *b*A*c*•, {$}]}

Stato7 $\xrightarrow{a}$ Stato10 = {[S → *bda*•, {$}]}

Possiamo osservare che il conflitto è stato risolto secondo i criteri visti in precedenza, cosicché la tabella risulta la seguente:

	a	**b**	**C**	**d**	**$**		**S**	**A**
0		s/3		s/4			1	2
1					acc			
2	s/5							
3				s/7				6
4	r5		s/8					
5					r1			
6				s/9				

7	s/10		r5					
8					r3			
9					r2			
10					r4			

Tabella 6 Grammatica LR(1)

Abbiamo quindi mostrato come si possono costruire parser deterministici guidati da tabelle LR(1). Per questi parsers vale l'importante proprietà per cui se $[A \rightarrow \alpha\bullet\beta, \sigma]$ è un item LR(1) valido per una stringa γ che appare nella pila e costituisce un prefisso possibile per una qualche forma sentenziale destra, allora si ha che α è suffisso di γ, $\sigma \subset \text{Follow}(A)$, $\sigma \subset \text{Follow}(\gamma\beta)$, avendo considerato l'estensione a stringhe di terminali e non terminali della funzione Follow.

Questa proprietà può essere estesa ai parsers LR(k), per qualsiasi k, considerando σ come formata da stringhe di k simboli, e considerando anziché l'insieme Follow, l'insieme Follow_k [1,2,13,14,15,16].

2.8.6 Costruzione della tabella per le grammatiche LALR(1)

Notiamo che il parser LR(1) ha all'incirca lo stesso numero di stati e, in generale, ne può avere di più del parser SLR(1). Infatti il numero di stati LR(0) per una grammatica in cui ci siano regole con lati destri lunghi al massimo r simboli sarà al massimo $2^{m \times (r+1)}$, mentre il numero di stati LR(1), se la grammatica possiede un alfabeto di t simboli, sarà $2^{m \times (r+1) \times 2^{t+1}}$.

Nei casi reali ovviamente non si raggiungeranno queste grandezze. Resta però il fatto che gli stati LR(1) costruiti sono il più delle volte

caratterizzati da poche colonne occupate, cioè prendono in considerazione sottoinsiemi limitati di $V \cup \Sigma$. In particolare, può accadere che stati diversi possiedano gli stessi nuclei di item, distinguendoli solo per gli insiemi di *lookahead* associati (ovvero, gli item senza considerare il *lookahead* di quello stato).

In questo caso un parser LALR(1) è ottenuto facendo due stati *i* e *j* e costruendo un unico stato *l* in modo tale che, se gli item in *i* sono del tipo $[A \rightarrow \alpha_{\bullet}\beta, \sigma_1]$ e quelli in *j* sono del tipo $[A \rightarrow \alpha_{\bullet}\beta, \sigma_2]$, con esattamente gli stessi A, α e β, *l* conterrà gli item del tipo $[A \rightarrow \alpha_{\bullet}\beta, \sigma_1 \cup \sigma_2]$. Ovviamente questo è possibile solo a patto di non introdurre nuovi conflitti.

In altri termini, il parser LALR(1) identifica gli stati uguali senza tener presente i *lookahead* creati con l'algoritmo del parser LR(1) e unisce fra loro questi stati fondendone i *lookahead*.

Gli item LALR(1) hanno la forma

$$A \rightarrow \alpha_{\bullet}\beta, \{t_1,\ldots, t_n\}$$

dove $A \rightarrow \alpha_{\bullet}\beta$ è un item di LR(0) e $\{t_1,\ldots, t_n\}$ è un insieme di tokens (o un insieme di *lookahead*).

Esempio Consideriamo la grammatica

$$S \rightarrow aAa \mid bAb \mid aBb \mid bBa$$
$$A \rightarrow c$$
$$B \rightarrow c$$

e costruiamo gli stati LR(1):

Stato0 = closure[S' $\rightarrow$ $_{\bullet}$S] = {[S' $\rightarrow$ $_{\bullet}$S, {\$}], [S $\rightarrow$ $_{\bullet}$*a*A*a*, {\$}],

$[S \rightarrow \bullet bAb, \{\$\}], [S \rightarrow \bullet aBb, \{\$\}], [S \rightarrow \bullet bBa, \{\$\}]\}$

$Stato0 \xrightarrow{S} Stato1 = \{[S' \rightarrow S\bullet, \{\$\}]\}$

$Stato0 \xrightarrow{a} Stato2 = \{[S \rightarrow a\bullet Aa, \{\$\}], [S \rightarrow a\bullet Bb, \{\$\}], [A \rightarrow \bullet c, \{a\}],$
$[B \rightarrow \bullet c, \{b\}]\}$

$Stato0 \xrightarrow{b} Stato3 = \{[S \rightarrow b\bullet Ab, \{\$\}], [S \rightarrow b\bullet Ba, \{\$\}], [A \rightarrow \bullet c, \{b\}],$
$[B \rightarrow \bullet c, \{a\}]\}$

$Stato2 \xrightarrow{c} Stato4 = \{[A \rightarrow c\bullet, \{a\}], [B \rightarrow c\bullet, \{b\}]\}$

$Stato3 \xrightarrow{c} Stato5 = \{[A \rightarrow c\bullet, \{b\}], [B \rightarrow c\bullet, \{a\}]\}$

Possiamo osservare che lo Stato4 e lo Stato5 contengono gli stessi nuclei degli item e li unifichiamo per ottenere uno Stato4/5:

$Stato4/5 = \{[A \rightarrow c\bullet, \{a,b\}], [B \rightarrow c\bullet, \{a,b\}]\}$

In conclusione, detti LR(0), SLR(1), LALR(1), LR(1) gli insiemi delle grammatiche che godono delle proprietà corrispondenti, si avrà $LR(0) \subset SLR(1) \subset LALR(1) \subset LR(1)$.

Tale sequenza di inclusioni deriva dalla sequenza di inclusioni, vera per qualsiasi grammatica $G = (\Sigma, V, P, S) \in LR(1)$, con $\sigma \subset (\sigma \cup \tau) \subset Follow(A) \subset T$, per qualsiasi σ, τ insiemi di *lookahead* per una qualche produzione in P, e per qualsiasi $A \in V$.

Infine, si osservi che la concatenazione del contenuto dello stack, letto dal basso verso l'alto, con la parte di ingresso ancora da consumare, costituisce ad ogni passo una forma sentenziale di una derivazione destra [1,2,13,14,15,16].

Capitolo 3

Parser generator YACC

3.1 Parser generator Yacc: introduzione

In questo capitolo vedremo come un parser generator può essere usato per facilitare la costruzione dell'interfaccia di un compilatore. Useremo Yacc, il parser generator LALR(1) con regole non ambigue, come base della nostra discussione, poiché implementa molti concetti discussi nei due capitoli precedenti ed è ampiamente disponibile. YACC (Yet Another Compiler-Compiler), creato nel 1970 da S. C. Johnson, è disponibile come comando sul sistema UNIX ed è stato usato per aiutare ad implementare centinaia di compilatori [1].

Yacc genera una funzione per controllare il trattamento d'ingresso. Questa funzione, chiamata parser (o analizzatore sintattico), chiama la routine a basso livello fornita dall'utente (analizzatore lessicale) per identificare gli elementi di base (tokens) nella stringa d'ingresso. I tokens sono organizzati secondo le regole di struttura d'ingresso, chiamate regole grammaticali; quando una di queste è riconosciuta, allora viene invocato il codice utente relativo alla regola, cioè un'azione; le azioni hanno la possibilità di restituire valori e di far uso dei valori di altre azioni. Le

specifiche di ingresso, in Yacc, corrispondono ad un insieme di regole grammaticali, ognuna delle quali descrive una possibile struttura e le assegna un nome [17]. Per **esempio**, una regola grammaticale è:

$$A \rightarrow B;$$

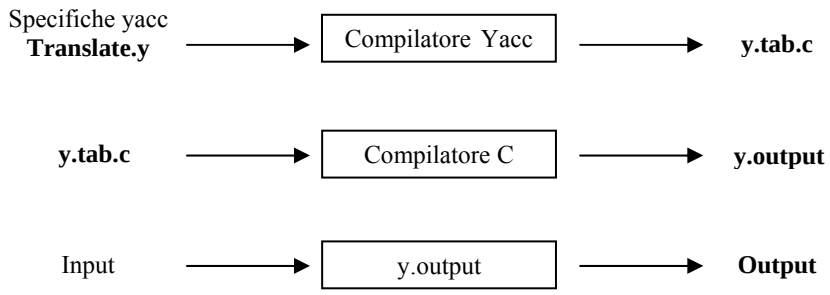
dove A e B rappresentano strutture nel trattamento d'ingresso; si presume B definito altrove. La freccia e il punto e virgola servono semplicemente come punteggiatura nella regola e non hanno significato nel controllo dell'ingresso.

Una parte importante del processo di analisi è svolta dall'analisi lessicale, che legge la sequenza in ingresso, riconosce le strutture a livello più basso e comunica i tokens identificati all'analizzatore sintattico. Una struttura riconosciuta dall'analizzatore lessicale è chiamata simbolo terminale (token), mentre una struttura riconosciuta dal parser è chiamata simbolo non terminale.

L'ingresso da leggere potrebbe non essere conforme alle specifiche; in questo caso si avrebbero degli errori che potrebbero essere identificati in tempo, quando possibile, con una scansione da sinistra a destra. La manipolazione degli errori permette il reinserimento dei dati errati o la continuazione del trattamento d'ingresso saltando i dati non validi [1,17].

3.2 File generati dal parser Yacc

Un traduttore può essere costruito usando Yacc com'è illustrato nella figura seguente:



Per prima cosa, il file, chiamato **translate.yacc**, contiene le specifiche di Yacc per la preparazione del traduttore. Nel sistema UNIX il comando **yacc translate.c** trasforma il file **translate.yacc** nel programma in C, chiamato **y.tab.c**, usando il metodo LALR. Il programma **y.tab.c** è la rappresentazione del parser LALR scritto nel linguaggio di programmazione C, con in aggiunta altre procedure preparate dall'utente. La tabella del parser LALR viene composta nel modo descritto nel capitolo 2. Compilando il file **y.tab.c** con la libreria **ly**, che contiene il programma di parsing LR, usando il comando **cc y.tab.c –ly** otteniamo il programma **y.output**, che realizza le relazioni specificate nel programma di origine di Yacc [1].

3.3 Struttura di un parser generator Yacc

3.3.1 Sezione del file di specifica

Ogni file di specifica di Yacc consta di tre sezioni: dichiarazioni, regole e programmi.

Le sezioni sono separate da un doppio segno di percento (%%). In altre parole, una specifica completa è quella del tipo:

dichiarazioni
%%
regole
%%
programmi

La sezione dichiarazioni può anche essere vuota. Inoltre, se la sezione programmi è omessa anche il secondo segno %% può essere omesso; quindi, la più piccola specifica di Yacc valida è:

%%
regole

I commenti sono racchiusi tra /*…*/.

La sezione regole è costituita da una o più regole grammaticali. Una regola grammaticale ha la forma:

$$A \rightarrow BODY\ ;$$

A rappresenta il nome di un non terminale, *BODY* una sequenza di zero o più nomi e lettere. La freccia ed il punto e virgola sono la punteggiatura utilizzata nelle regole per separare il corpo di una regola dal simbolo non terminale e le regole fra loro.

Se ci sono diverse regole con la stessa parte sinistra, può essere usata la barra verticale | per evitare di riscrivere la parte sinistra. Inoltre, si può omettere il punto e virgola prima della barra verticale.

Quindi le regole della grammatica

A → B C D ;
A → E F ;
A → G ;

possono essere presentate a Yacc come:

```
A : B C D
  | E F
  | G
  ;
```

Non è necessario che tutte le regole grammaticali con la stessa parte sinistra appaiano insieme nella sezione regole, sebbene ciò renda l'ingresso più facile da leggere e da cambiare.

Se un simbolo non terminale indica una stringa vuota, lo si può indicare nel modo più ovvio:

```
Empty : ;
```

I nomi che rappresentano tokens devono essere dichiarati nella sezione dichiarazioni:

```
%token a b ...
```

Ogni nome non definito nella sezione dichiarazioni è assunto a rappresentare un simbolo non terminale. Ogni simbolo non terminale deve apparire nella parte sinistra di almeno una regola.

Di tutti i simboli non terminali, uno, chiamato *start symbol* (simbolo iniziale, S), ha importanza particolare. Il parser è progettato per riconoscere il simbolo di partenza, che rappresenta la struttura più ampia e più generale

descritta dalle regole grammaticali. È preferibile dichiarare esplicitamente il simbolo di partenza nella sezione dichiarazioni:

%start S

La fine dell'ingresso è segnalata da un token speciale, chiamato *endmarker*. Se tale token è visto in un qualsiasi altro contesto viene segnalato come errore [1,17].

3.3.2 Sezione azioni

Ad ogni regola grammaticale l'utente può associare azioni da eseguire ogni volta che la regola è applicata nel trattamento d'ingresso; quindi ogni regola è costituita da una produzione grammaticale e da un'azione semantica associata alla regola stessa. Questa azione può restituire valori ed ottenere a sua volta valori restituiti da azioni precedenti.

Una serie di produzioni come

$$A \rightarrow B \mid C \mid \ \ldots \mid Y$$

può essere scritta in Yacc:

```
A : B   { azione semantica 1}
  | C   { azione semantica 2}
. . .
  | Y   { azione semantica n}
  ;
```

Per permettere una facile comunicazione tra le azioni ed il parser si utilizza il simbolo dollaro ($). Per restituire un valore, l'azione normalmente

assegna alla pseudo-variabile $$ un qualche valore. Per esempio, un'azione che restituisce il valore 1 è { $$ = 1; }, definita azione di default.

In un'azione semantica, il simbolo $$ è riferito al valore associato al simbolo non terminale a sinistra della regola, mentre $i è il valore riferito all'i-esimo simbolo a destra di una regola (terminale o non terminale), leggendo da sinistra a desta.

Quindi, se abbiamo una regola della forma E → E + T | T la sua azione semantica associata è:

```
E : E '+' T   { $$ = $1 + $3; }
  | T
  ;
```

Notiamo che il simbolo non terminale T a destra della prima produzione è il terzo simbolo grammaticale e + è il secondo. L'azione semantica associata alla prima produzione consiste nell'aggiungere al valore di E di destra della produzione il valore di T e nell'assegnare il valore del risultato al simbolo non terminale E di sinistra. L'azione semantica della seconda produzione è di default essendo una regola grammaticale della forma A → B.

Si possono definire, inoltre, altre variabili da usare nelle azioni. Dichiarazioni e definizioni possono apparire nella sezione dichiarazioni racchiuse tra i segni %{ e %}. Queste dichiarazioni e definizioni sono globali e perciò conosciute nelle dichiarazioni delle azioni e nell'analizzatore lessicale [1,17].

3.3.3 Sezione programmi

La sezione programmi può includere la definizione dell'analizzatore lessicale *yylex()* e tutte le altre funzioni, per esempio quelle usate nelle azioni specificate dalle regole grammaticali. Non è specificato se la sezione programmi precede o segue le azioni semantiche del file output di Yacc; perciò se l'applicazione contiene definizioni e dichiarazioni progettate per l'applicazione del codice nelle azioni semantiche, definizioni e dichiarazioni verranno scritte nella sezione dichiarazioni, racchiuse tra i segni %{ e %} [1].

3.4 Interfaccia con l'analizzatore lessicale

In questo paragrafo tratteremo un analizzatore lessicale con cui leggere la sequenza d'ingresso e comunicare i tokens (con i valori, se si desidera) al parser. L'analizzatore lessicale è una funzione di tipo intero chiamata *yylex()*. Tale funzione restituisce un intero che indica il valore associato al token letto. Se questo token deve essere restituito allora deve essere definito nella sezione dichiarazioni nelle specifiche Yacc. Se vi è un valore da associare a questo token, esso è comunicato al parser attraverso la variabile esterna *yylval.*

Il parser e l'analizzatore lessicale devono accordarsi sui numeri per rappresentare i tokens al fine di comunicare tra loro per funzionare. I numeri possono essere scelti da Yacc o dall'utente. Nella situazione di default, i numeri sono scelti da Yacc. Per assegnare un numero ad un token (letterali inclusi), se il nome del token viene scritto per la prima volta nella sezione delle dichiarazioni, allora tale nome può essere immediatamente seguito da

un intero non negativo. Questo intero è assunto per essere il numero di token del nome o del letterale. I tokens non definiti da questo meccanismo mantengono la loro definizione di default, l'importante è che tutti i numeri di token siano distinti. Il token *endmarker* avrà un numero di token 0 o negativo. Questo numero di token non può essere ridefinito dall'utente; quindi tutti gli analizzatori lessicali devono essere preparati per restituire uno 0 o un numero di token negativo alla fine dell'ingresso del token *endmarker*. Uno strumento molto utile per costruire analizzatori lessicali è il programma LEX (Lexical Analyzer Generator, sviluppato da Mike Lesk[7]), progettato proprio per lavorare in accordo con i parsers generati da Yacc [1,17].

3.5 Come lavora il parser prodotto da Yacc

Yacc trasforma il file di specifica in un programma C che analizza l'ingresso secondo le specifiche date. L'algoritmo usato per arrivare al parser partendo dalle specifiche è complesso: verranno, in seguito, discusse solo alcune parti del file *y.tab.c*.

Il parser prodotto da Yacc consiste in un automa a stati finiti con una pila ed è capace di leggere e memorizzare il token successivo (chiamato token di *lookahead*). Lo stato corrente è sempre quello in cima alla pila. Gli stati dell'automa sono etichettati con numeri di interi; all'inizio l'automa è nello stato 0, la pila contiene solo lo stato 0 e nessun token di *lookahead* è stato letto.

L'automa ha solo quattro azioni possibili, chiamate spostamento

[7] Cfr; per maggiori dettagli: "LEX – A Lexical Analyzer Generator" di M. E. Lesk, Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, October 1975.

(*shift*), riduzione (*reduce*), accettazione ed errore. Una mossa del parser è eseguita come segue:

1. partendo dallo stato corrente, il parser controlla se ha bisogno di un token di *lookahead* per vedere quale azione può compiere e, se ne ha bisogno e non ce l'ha, chiama la funzione *yylex()* per ottenere il token successivo;
2. usando lo stato corrente e, se occorre, il token di *lookahead*, il parser decide sulla sua prossima azione e la esegue. Questo avviene quando si effettuano operazioni di inserimento o di astrazione di stati nella o dalla pila.

L'azione più comune che il parser può compiere è quella di *spostamento*. Quando si effettua un'azione di spostamento, c'è sempre un token di *lookahead*. Per esempio, nello stato 56 ci potrebbe essere un' azione:

IF shift 34

Questa indica che, nello stato 56, se il token di *lookahead* è IF, tale stato viene messo più in basso nella pila e lo stato 34 diventa lo stato corrente (in cima alla pila). Il token di *lookahead* è cancellato.

L'azione di *riduzione*, invece, impedisce che la pila possa aumentare all'infinito. Le azioni di riduzione sono appropriate quando il parser ha visto la parte destra di una regola grammaticale e la rimpiazza con la parte sinistra. Un'azione di riduzione di default è rappresentata da un punto (.).

Le azioni di riduzione sono associate a singole regole grammaticali, alle quali sono associati dei numeri. Ad esempio, l'azione

. reduce 18

si riferisce alla regola grammaticale 18, invece l'azione

IF shift 34

si riferisce allo stato 34.

Ammettiamo che la regola da ridurre sia

$$A \rightarrow x\ y\ z\ ;$$

L'azione di riduzione dipende dal simbolo della parte sinistra (A in questo caso) e dal numero di simboli nella parte destra (tre in questo caso). Per ridurre, prima si estraggono i tre stati in cima alla pila, cioè *x*, *y* e *z* (in generale, il numero degli stati estratti eguaglia il numero di simboli nella parte destra della regola). Dopo l'estrazione di questi stati, resta scoperto lo stato in cui il parser si trova prima dell'inizio del trattamento della regola. Usando questo stato scoperto ed il simbolo della parte sinistra della regola, si esegue quello che in effetti è lo spostamento di A. Si ottiene, così, un nuovo stato che viene posto in cima alla pila, e l'analisi continua. Questa azione viene definita come azione di *goto*. In generale, lo stato scoperto contiene un elemento del tipo

A *goto 20*

che causa l'inserimento nella pila dello *stato 20* che diviene lo stato corrente.

Se la parte destra della regola è vuota, dalla pila non viene estratto nessuno stato: lo stato scoperto è infatti lo stato corrente.

L'azione di riduzione è importante anche nel trattamento delle azioni e dei valori dati dall'utente.

In aggiunta alla pila che tiene lo stato, un'altra pila, che lavora in parallelo alla prima, mantiene i valori restituiti dall'analizzatore e dalle

azioni. Quando arriva un token ottenuto mediante uno spostamento, la variabile esterna *yylval* è copiata sulla pila dei valori. Una volta terminata l'esecuzione del codice utente, la riduzione è completata. Quando si svolge l'azione di *goto*, la variabile esterna *yylval* è copiata nella pila dei valori. Le pseudo-variabili $1, $2,…, puntano alla pila dei valori.

L'azione di *accettazione* indica che l'intero ingresso si attiene alla specifica. Questa azione viene eseguita solo quando il token di *lookahead* è l'*endmarker* ed indica che il parser ha svolto il suo lavoro con successo.

L'azione di *errore* rappresenta un punto in cui il parser non può proseguire la scansione secondo le specifiche. Il token d'ingresso visto insieme con il token di *lookahead* non può essere seguito da nessun token, a differenza di quanto avverrebbe in un ingresso regolare, pertanto il parser tenta di risolvere la situazione e riprende la scansione [1].

3.6 Ambiguità e conflitti usando Yacc

Un insieme di regole grammaticali è ambiguo se c'è qualche stringa in ingresso che può essere strutturata in due o più modi differenti. Per esempio, la regola grammaticale

expr : expr - expr

è un modo naturale per esprimere il fatto che un modo di formare un'espressione aritmetica è quello di mettere altre due espressioni insieme separate da un meno. Purtroppo questa regola grammaticale non specifica del tutto il modo in cui tutti gli ingressi complessi potrebbero essere strutturati. Per esempio, se l'ingresso è

expr - expr - expr

la regola ammette questo ingresso strutturato come

(expr - expr) - expr

oppure

expr - (expr - expr)

(la prima è chiamata *associazione a sinistra*, la seconda *associazione a destra*). Yacc trova questa ambiguità quando tenta di costruire il parser.

Il parser prodotto da una grammatica in input può contenere stati dove occorrono dei conflitti. Di norma questi conflitti occorrono quando la grammatica non è LALR(1). Una grammatica ambigua contiene sempre, al più, un conflitto LALR(1). Yacc, comunque, cerca di risolvere questi conflitti utilizzando regole di default e regole di precedenza.

Nel caso in cui si presentino conflitti *shift/reduce* o *reduce*/*reduce*, Yacc produce comunque un parser, e lo fa selezionando uno dei passi validi ogni volta che ha una scelta. Una regola che descrive quale scelta fare in una data situazione è chiamata *regola di disambiguazione*.

Yacc di default si attiene a due regole di disambiguazione:

1. in un conflitto *shift/reduce* effettua l'operazione di spostamento (*shift*), e pertanto, in questo caso, le riduzioni vengono ignorate;
2. in un conflitto *reduce/reduce* effettua l'operazione di ridurre la regola grammaticale che viene prima in rapporto alla specifica di Yacc (per quanto è possibile è meglio evitare tali conflitti).

I conflitti potrebbero sorgere a causa di un errore di ingresso o di logica, o perché le regole grammaticali richiedono un parser più complesso

di quanto Yacc sia in grado di produrre. L'uso delle azioni interne alle regole può anche causare conflitti se l'azione deve essere fatta prima che il parser possa essere sicuro su quale regola sia da riconoscere. In questi casi l'applicazione delle regole di disambiguazione è inappropriata e porta ad un parser scorretto. Per questo motivo Yacc riporta sempre il numero di conflitti *shift/reduce* e *reduce/reduce* risolti dalle due regole di default. In generale, ogni volta che è possibile applicare una regola di disambiguazione per produrre un parser corretto è anche possibile riscrivere le regole grammaticali in modo che lo stesso ingresso sia letto senza che ci siano conflitti. Molti dei precedenti parser generators consideravano i conflitti come errori fatali, mentre Yacc produce parsers, dunque, anche in presenza di conflitti.

C'è una situazione comune in cui, per risolvere i conflitti, le regole date sopra non sono sufficienti; questo si verifica nel parsing delle espressioni aritmetiche. Molte delle costruzioni usate per le espressioni aritmetiche possono essere descritte in modo naturale con la nozione di livelli di precedenza per gli operatori, insieme alle informazioni di associatività destra e sinistra. Risulta che grammatiche ambigue con regole di disambiguazione appropriate possano essere usate per creare parsers più veloci e più facili da scrivere rispetto a parsers costruiti da grammatiche non ambigue. La nozione di base è scrivere regole grammaticali della forma

$$E \rightarrow E \; op \; E \mid unary \; E$$

per tutti gli operatori binari ed unari. Ciò crea una grammatica molto ambigua, con molti conflitti di parser. Come regola di disambiguazione, l'utente specifica la precedenza, o forzatura obbligata, di tutti gli operatori, e l'associatività degli operatori binari. Queste informazioni sono sufficienti per permettere a Yacc di risolvere i conflitti di parsing in sintonia con queste

regole e di costruire un parser che realizzi le precedenze e le associatività desiderate, le quali sono associate a token nella sezione dichiarazione.

Tale sezione è costituita da una serie di linee che iniziano con una parola chiave di Yacc (*%left, %right* o *%nonassoc*) seguita da una lista di tokens. Tutti i tokens che si trovano sulla stessa linea hanno lo stesso livello di precedenza e associatività; le linee sono elencate in ordine crescente di precedenza o di forzatura obbligata. Quindi

%left ‘ + ’ ‘ - ’
%left ‘ * ’ ‘ / ’

descrive la precedenza e l’associatività dei quattro operatori aritmetici.

Gli operatori + e - hanno associatività a sinistra e minore precedenza rispetto agli operatori * e /, anche essi associativi a sinistra. La parola *%nonassoc* è usata per descrivere operatori che non si possono associare a se stessi. A volte un operatore unario ed un operatore binario hanno la stessa rappresentazione simbolica ma differenti precedenze. Un esempio di ciò è l’operatore - unario e binario; al - unario si potrebbe dare lo stesso peso della moltiplicazione. La parola chiave *%prec* cambia il livello di precedenza associato ad una particolare regola grammaticale, appare immediatamente dopo il corpo di una regola grammaticale (prima di un’azione o di un punto e virgola di chiusura) ed è seguita da un nome di token o da un letterale. In Yacc tale operazione viene scritta nel modo seguente:

E : ‘-’ E *%prec* ‘*’ ;

Un token dichiarato da *%left*, *%right* e *%nonassoc* non ha bisogno di essere, ma potrebbe esserlo, dichiarato da *%token*.

Precedenza e associatività sono usate da Yacc per risolvere i conflitti di parsing e danno origine a nuove regole di disambiguazione.

Formalmente, le regole lavorano come segue:

1. l'associatività e la precedenza sono riservate a quei tokens e a quei letterali cui sono state assegnate;
2. una precedenza ed un'associatività sono associate ad ogni regola grammaticale. Se è usata la costruzione *%prec*, questa avrà prevalenza sulla costruzione di default. Alcune regole grammaticali possono non avere né precedenza né associatività ad esse associate;
3. quando c'è un conflitto *reduce/reduce*, o ci sono un conflitto *shift/reduce* ed il simbolo in ingresso, o quando la regola grammaticale non ha precedenza ed associatività, allora vengono usate le due regole di disambiguazione date all'inizio della sezione e sono riportati i conflitti;
4. se c'è un conflitto *shift/reduce* e allo stesso tempo la regola grammaticale ed il carattere in ingresso hanno precedenza ed associatività, allora il conflitto è risolto a favore delle azioni (*shift* o *reduce*) associate con la precedenza più alta. Se le precedenze sono le stesse, allora è usata l'associatività; l'associatività a sinistra implica la riduzione, quella a destra implica lo spostamento, la non associazione implica un errore.

I conflitti risolti con la precedenza non sono calcolati nel numero dei conflitti *shift/reduce* e *reduce/reduce* riportati da Yacc. Questo significa che errori nelle specifiche delle precedenze possono mascherare errori nella grammatica in ingresso [1,17,18,19].

3.7 Gestione degli errori in Yacc

La gestione degli errori è un campo estremamente difficile e molti dei problemi che si incontrano sono semantici. Quando si trova un errore, potrebbe essere necessario, per esempio, richiedere il contenuto dell'albero sintattico, cancellare o alterare simboli in entrata nella tabella e, di norma, regolare i cambiamenti per evitare di generare qualche altro errore in uscita.

Quando si trova un errore raramente si interrompe l'elaborazione: è più utile, infatti, continuare la scansione dell'ingresso per trovare successivi errori di sintassi. Questo conduce al problema di far ripartire il parser dopo un errore richiedendo (attraverso algoritmi specifici) di scartare diversi tokens dalla stringa di ingresso e di tentare di aggiustare il parser così da poter continuare con l'ingresso.

In Yacc esiste un token di nome *error* riservato proprio alla gestione degli errori. Questo nome può essere usato anche nelle regole grammaticali. Definiamo quindi, in aggiunta ad un insieme di regole della grammatica, una produzione di errore definita come segue:

$$A \rightarrow error\ \alpha$$

dove A è un simbolo non terminale e α è una stringa di simboli grammaticali oppure è una stringa vuota. Yacc, quindi, genererà un parser dalle specifiche, trattando le produzioni di errore come produzioni ordinarie.

Tuttavia un parser generator Yacc, quando incontra un errore, tratta in modo speciale gli stati contenenti item che producono errori. Quando si incontra un errore, Yacc estrae simboli dalla pila finché entra in uno stato dove è ammesso il token error della forma $A \rightarrow .error\ \alpha$, quindi si comporta come se il token error fosse il token di *lookahead* corrente (token che ha causato l'errore) ed esegue l'azione incontrata. Quindi il parser

sposta un fittizio token error sulla pila, come se vedesse il token error sul relativo input. Quando $\alpha \in \varepsilon$, occorre immediatamente una riduzione di A ed è invocata l'azione semantica associata alla produzione A → *error* ; il parser, allora, scarta i simboli in input finché trova un simbolo in input sul quale può effettuare la sua analisi. Se $\alpha \notin \varepsilon$, Yacc salta la sottostringa in input che può essere ridotta da α. Se α è composto da simboli terminali, allora Yacc guarda la stringa di terminali in input e la riduce effettuando l'operazione di *shift* della stringa sulla pila. A questo punto il parser avrà in cima alla pila *error α*. Il parser ridurrà *error α* ad A e riprenderà la normale analisi. Se non sono state specificate speciali regole per gli errori, allora l'elaborazione termina in uno stato di errore.

Per **esempio**, una regola generica di errore della forma

stmt → *error*

significherebbe, in effetti, che su un errore di sintassi il parser potrebbe tentare di saltare oltre le dichiarazioni in cui l'errore è stato visto. Più precisamente, il parser scansionerà oltre, cercando i tokens successivi che possono seguire un'istruzione, e farà ripartire il processo col primo di questi tokens. Se le parti iniziali delle istruzioni non sono sufficientemente distinte, si può produrre una falsa partenza a metà di un'istruzione e in conclusionen questo caso viene riportato un secondo errore, di fatto inesistente. Pertanto, per prevenire una cascata di messaggi di errore, il parser, una volta localizzato l'errore, rimane in uno stato di errore finché non legge un certo numero di tokens successivi e li sposta in cima alla pila. Se un errore è localizzato quando il parser è già in uno stato di errore, non è dato nessun messaggio e il token in ingresso è tranquillamente cancellato.

Per gli errori si potrebbero usare, tuttavia, delle azioni con alcune speciali regole, più facili da trattare rispetto ad una regola generica di errore, che è difficilmente controllabile. Una regola più semplice e facile da controllare è, ad esempio, quella dalla forma:

stmt → error ' ; '

dove, in presenza di un errore, il parser tenta di saltare oltre l'istruzione, ma fa ciò soltanto fino al ';' successivo. Tutti i tokens dopo l'errore e prima del ';' successivo non possono essere spostati e vengono ignorati. Quando il ';' viene visto, questa regola permetterà la riduzione e verrà eseguita ogni azione di pulizia associata ad essa.

Un altro tipo di regola di errore è utile nelle applicazioni interattive, dove sarebbe desiderabile poter reinserire una linea dopo un errore. Una possibile regola di errore potrebbe essere

```
lines   :   error ' \n'   { printf ( " Reenter last line:   " ) ;  } input
                          {  $$ = $4;  }
```

che causa la sospensione di una normale analisi quando si verifica un errore sintattico in una linea di input. Quando si incontra l'errore, il parser estrae simboli dalla sua pila finché non si trova in uno stato che effettui l'azione di *shift* sul token *error*; tale stato, quindi, deve includere l'item della forma:

lines : . error ' \n'

Quindi il parser sposta il token *error* sulla pila e salta avanti nella stringa di input finché non trova il carattere che permette di inserire una nuova linea. A questo punto il parser sposta il carattere in testa alla pila, riduce *error '\n'* in *lines* e scrive il messaggio diagnosticato. In Yacc esiste una procedura *yyerrok()* che fa tornare il parser nella modalità normale (cioè

nel suo modo normale di operare). Pertanto l'ultimo esempio può essere scritto in un modo migliore:

```
lines   :   error ' \n'
                    {       yyerrok;
                            printf ( " Reenter last line:   " ) ;    }
            input
                        { $$ = $4; }
            ;
```

Come abbiamo visto, il token posto immediatamente dopo il simbolo *error* è il token d'ingresso in cui l'errore è stato scoperto. A volte è inappropriato; per esempio, un'azione di recupero di un errore può incaricarsi di ritrovare il posto giusto per far ripartire il processo di analisi. In questo caso il precedente token di *lookahead* deve essere eliminato, e ciò viene fatto attraverso la dichiarazione

```
yyclearin ;
```

Questi meccanismi sono alquanto rigidi ma permettono al parser di funzionare anche in presenza di molti errori [1,17,18,19].

3.8 Ambiente di sviluppo Yacc

Quando l'utente fornisce in ingresso a Yacc un file di specifica, l'uscita è un sorgente C, chiamato *y.tab.c* nella maggior parte dei sistemi (a seconda delle convenzioni dei file system locali, i nomi possono essere diversi da installazione a installazione).

La funzione prodotta da Yacc è chiamata *yyparse()* ed è una funzione che restituisce un intero. Quando è invocata, essa chiama ripetutamente *yylex()*, l'analizzatore fornito dall'utente per ottenere i tokens d'ingresso. Eventualmente, se si è localizzato un errore (a meno di recupero degli errori), *yyparse()* restituisce il valore 1 oppure l'analizzatore lessicale restituisce il token di *endmarker* e fa accettare il parser. In questo caso, *yyparse()* restituisce il valore 0.

L'utente deve fornire un ambiente appropriato a questo parser per ottenere un programma operativo. Per esempio, come in ogni programma C, deve essere definito un programma chiamato *main()*, che ad un certo punto chiami *yyparse()*. In aggiunta, si invoca una procedura chiamata *yyerror()* per stampare un messaggio di errore quando lo si incontra. Tale messaggio indica, di norma, la stringa "syntax error" e l'indicazione del luogo in cui l'errore sintattico è stato localizzato.

La variabile intera esterna *yychar* contiene il numero di token di *lookahead* al momento della localizzazione.

La variabile intera esterna *yydebug* è normalmente posta a 0. Se ha un valore diverso da 0, il parser produrrà una descrizione a parole delle proprie azioni, ed anche una descrizione dei simboli in ingresso che sono stati letti. A seconda dell'ambiente operativo, potrebbe essere possibile settare questa variabile per l'uso di un sistema di debug [17].

3.9 Ricorsione sinistra

L'algoritmo usato dal parser Yacc fa uso di regole grammaticali con ricorsione sinistra della forma:

name : name rest_of_rules ;

Queste regole capitano spesso nella scrittura di specifiche di liste e sequenze:

```
list  :  item
      |  list  ';'  item
      ;
```

e

```
seq  :  item
     |  seq  item
     ;
```

In ognuno di questi casi, la prima regola sarà ridotta solo per il primo item, la seconda regola sarà ridotta per il secondo item e per quelli successivi.

Con le regole che usano la ricorsione destra come

```
seq  :  item
     |  item  seq
     ;
```

il parser potrebbe diventare più grande e gli item sarebbero visti e ridotti da destra a sinistra; addirittura, poi, una pila interna al parser potrebbe essere in pericolo di overflow se viene letta una sequenza molto lunga. L'utente, quindi, dovrebbe usare le ricorsioni sinistre dove è ragionevole che queste vengano usate.

Va considerato, inoltre, che se una sequenza di lunghezza zero è significativa allora la specifica è la seguente:

```
seq  :  /* empty */
     |  item  seq
     ;
```

Anche in questo caso la prima regola si potrebbe ridurre una sola volta mentre la seconda regola viene ridotta una volta per ogni item letto. L'ammissione si sequenze vuote porta spesso a guadagnare in generalità. Comunque possono sorgere conflitti se si chiede a Yacc di decidere quale sequenza vuota ha visto, quando non ha visto abbastanza di essa per decidere [17,18].

3.10 Limiti interni di Yacc

Yacc, al suo interno, può avere parecchie tabelle. I limiti minimi e massimi per queste tabelle sono indicati nella tabella seguente. L'esatto significato di questi valori è definito nell'esecuzione che implementa la relazione tra questi valori e tra loro e tutti i messaggi di errore che Yacc potrebbe generare [20].

	Minimo	
Limiti	**Massimo**	**Descrizione**
{NTERMS}	126	Numero di tokens.
{NNONTERM}	200	Numero di non terminale.
{NPROD}	300	Numero di regole.
{NSTATES}	600	Numero di stati.
{MEMSIZE}	5200	Lunghezza delle regole; lunghezza totale, nel nome (tokens e non terminali) di tutte le regole della grammatica. La parte sinistra viene contata per ogni regola, anche se non è esplicitamente ripetuta, come

		specificato nelle regole grammaticali in Yacc.
{ACTSIZE}	4000	Numero delle azioni. L'azione qui (e nella descrizione del file) si riferisce alle azioni del parser (*shift*, *reduce*, e così via) e non all'azione semantica definita nelle regole grammaticali di Yacc.

3.11 Descrizione dei file generati da Yacc

Come detto nei precedenti paragrafi, data una grammatica, Yacc genera due file: il file *y.output*, che rappresenta la tabella del parser LALR, e il file *y.tab.c*, che consiste nel programma C della grammatica ed usa il metodo LALR.

Quello che ci proponiamo di fare in questa sede è di descrivere tali file, anche se il file *y.tab.c* non è stato discusso in modo approfondito in ogni sua parte.

Definiamo, in un file chiamato *a.yacc*, la seguente specifica:

```
%start S
%token x y z

%%
S : A B
  ;
A : x y
```

```
  ;
B : z
  ;
%%
```

Quando Yacc è chiamato con l'opzione –v, viene prodotto un file chiamato *y.output* contenente una descrizione del parser. Il file *y.output* corrispondente alla precedente grammatica è:

```
0   $accept : S  $end
1   S : A B
2   A : x y
3   B : z

state 0
        $accept : . S   $end
        x   shift 3
        .   error
        S   goto 1
        A  goto 2
state 1
        $accept : S .   $end
        $end    accept
        .   error
state 2
        S : A . B
        z   shift 5
        .   error
```

```
        B  goto 4
state 3
        A : x . y
        y   shift 6
        .   error
state 4
        S : A B .       (1)
        .   reduce 1
state 5
        B : z .         (3)
        .   reduce 3
state 6
        A :  x y .      (2)
        .   reduce 2

3 terminals, 3 nonterminals
4 grammar rules, 7 states
```

Viene usato il simbolo **.** per indicare quale parte della regola è stata vista e quale è ancora da vedere.

Si supponga che la stringa di input sia: *x y z*

All'inizio lo stato corrente è lo stato 0. Il parser ha bisogno di riferirsi alla stringa di input in modo da decidere tra le azioni disponibili nello stato 0; il primo token, *x*, viene dunque letto, diventando il token di *lookahead*. L'azione nello stato 0 su *x* è "shift 3": così lo stato 3 viene messo in cima alla pila diventando lo stato corrente ed il token di *lookahead* torna vuoto. Il token successivo, *y*, viene letto e diventa il token di

lookahead. L'azione nello stato 3 per il token *y* è "shift 6", così lo stato 6 viene messo nella pila ed il *lookahead* è di nuovo vuoto. La pila ora contiene 0, 3 e 6; nello stato 6, questa volta senza consultare il *lookahead*, il parser riduce la regola 2:

A : x y

Questa regola ha due simboli nella parte destra, quindi due stati, 6 e 3, sono estratti dalla pila, scoprendo lo stato 0. Consultando la descrizione dello stato 0, che riguarda l'operazione di goto sul simbolo non terminale A, si ottiene:

A : goto 2

Quindi lo stato 2 viene messo sulla pila, diventando lo stato corrente.

Nello stato 2, il token successivo, *z*, viene letto. L'azione è "shift 5", così lo stato 5 è inserito nella pila, che ora ha 0, 2 e 5 ed il *lookahead* vuoto. Nello stato 5 l'unica azione possibile è ridurre con la regola 3. Questa ha un simbolo nella parte destra e quindi lo stato 5 viene eliminato dalla pila lasciando in cima ad essa lo stato 2. Il *goto* nello stato 2 con B, considerando la parte sinistra della regola 3, è lo stato 4. Ora la pila contiene 0, 2 e 4. Nello stato 4 la sola azione è ridurre secondo la regola 1. Questa presenta due simboli sulla destra, e quindi sono estratti dalla pila due stati, lasciando ancora lo stato 0. Nello stato 0 c'è un *goto* con S che fa entrare il parser nello stato 1. In esso l'input è stato letto; si raggiunge l'*endmarker*, indicato con "$end" nel file *y.output*. L'azione nello stato 1, quando l'*endmarker* è letto, è di accettazione e fa terminare l'analisi al parser.

Nel file *y.tab.c*, come in ogni altro programma, si hanno la parte delle dichiarazioni e la parte del codice generato.

Nelle dichiarazioni vengono riportati la versione e il tipo di generatore di cui si fa uso, sono dichiarate e inizializzate le variabili riguardanti i tokens e il flag degli errori, vengono assegnati ai tokens un numero di token a partire dal 257 e alla variabile di errore YYERRCODE il valore di default 25, viene riportata la variabile YYMAXTOKEN, che indica il valore massimo assegnato ai tokens, ed inoltre vengono dichiarati gli array.

Prendiamo in esame la grammatica data precedentemente; quello che cercheremo di fare è descrivere due array che vengono dichiarati all'interno del file *y.tab.c*. Innanzitutto è bene ricordare che le produzioni sono numerate, pertanto:

-1 S' : S

0 S : A B

1 A : x y

2 B : z

const short yylhs[] = { -1 , 0 , 1 , 2 }

dove: **-1** è di default ed è riferito alla regola S' : S;

0 è riferito alla produzione S : A B e vengono messi tanti zero quante sono le regole della produzione S;

1 è riferito alla produzione A : x y e vengono messi tanti uno quante sono le regole della produzione A;

2 è riferito alla produzione B : z e vengono messi tanti due quante sono le regole della produzione B;

e così via nel caso ci fossero altre produzioni.

const short yylen[] = { 2 , 2 , 2 , 1 } indica la lunghezza della parte destra della regola,

dove: **2** è di default ed è riferito alla regola S' : S;

2 è riferito alla produzione S : A B;

2 è riferito alla produzione A : x y;

1 è riferito alla produzione B : z;

Nella parte del codice, invece, troviamo i seguenti metodi:

yygrowstack(): permette di inizializzare le pile con i vari puntatori;

yyparse(): assegna i numeri ai tokens così da far funzionare e comunicare in modo corretto il parser e l'analizzatore lessicale. I numeri possono essere scelti da Yacc o dal programmatore;

yyloop: permette di leggere ogni simbolo della stringa di input e di verificare se bisogna effettuare un'operazione di *shift* o di *reduce*;

yynewerror: permette di scrivere il messaggio di errore "syntax error";

yyinrecovery: permette di effettuare il recupero degli errori;

yyreduce: permette di effettuare l'operazione di *reduce*;

yyoverflow: controlla se la pila è piena ed invia il seguente messaggio di errore: “yacc stack overflow”;

yyabort: invia il valore 1 nel caso in cui l’analisi termini con insuccesso;

yyaccept: invia il valore 0 nel caso in cui la stringa di input sia stata accettata.

Capitolo 4

Parser generator Bison

4.1 Parser generator Bison: introduzione

Bison [21] è un generatore per tutti i tipi di parsers e converte una grammatica context-free in un parser per le grammatiche LALR(1) o GLR. Bison è compatibile con Yacc: tutte le grammatiche scritte per Yacc possono essere usate anche per Bison.

Bison entra in esecuzione con una linea di comando:

bison namegramm.y

dove **namegramm.y** è il nome del file della grammatica da usare per creare il parser (il suffisso .y è usato tradizionalmente in UNIX per indicare una grammatica per parser).

Bison crea un file C, **namegramm.tab.c**, che può poi essere compilato e collegato con il resto del programma. Bison accetta due argomenti opzionali che dovrebbero essere posti prima del nome della grammatica:

-d : genera l'header file **namegramm.tab.h** con le informazioni sui tokens;

-v : genera un file di output utile per il controllo della tavola del parser.

4.2 Struttura di un parser generator Bison

Il file di input per Bison è una grammatica dalla forma:

```
%{
        Prologue
%}
Dichiarazioni di Bison
%%
        Regole grammaticali
%%
Epilogue
```

I simboli %%, %{, %} appaiono in ogni file di input per Bison e servono a separare le sezioni.

I commenti vengono racchiusi tra /*…*/ e possono comparire in tutte le sezioni.

4.2.1 Sezione prologue

La sezione prologue [21] definisce i tipi e le variabili usati nelle azioni. Inoltre vengono dichiarati la funzione dell'analizzatore lessicale *yylex()*, la funzione per la gestione del messaggio di errore *yyerror()* e altri

identificatori globali usati nelle regole grammaticali. Tale sezione termina con la prima occorrenza %} posta fuori da ogni commento. Si può avere più di una sezione prologue, mescolata con la sezione delle dichiarazioni di Bison e ciò è possibile solo nel caso si usi il linguaggio di programmazione C. Per esempio, la dichiarazione *%union* può usare tipi definiti nel file header e funzioni che prendono argomenti di tipo YYSTYPE. Questo tipo può essere denotato con due blocchi di prologue, uno prima e l'altro dopo la dichiarazione di *%union*.

4.2.2 Sezione dichiarazioni

La sezione dichiarazioni [21] definisce i nomi dei simboli terminali e non terminali e può anche descrivere la precedenza degli operatori e i tipi di dati dei valori semantici dei vari simboli. Alcune grammatiche che hanno una struttura semplice non hanno bisogno della sezione dichiarazioni. Tutti i tokens di tipo *name* (esclusi i tokens + e *) possono essere dichiarati. I simboli non terminali devono essere dichiarati se si specifica il tipo di dato da usare per il valore semantico. La prima regola del file di input specifica il simbolo iniziale (di partenza), dato come default. Se si desidera dare un simbolo iniziale differente da quello di default, basta semplicemente dichiararlo.

4.2.3 Sezione regole grammaticali

La sezione regole grammaticali [21] definisce la struttura di ogni simbolo non terminale e contiene, quindi, una o più regole della grammatica

di Bison. In questa sezione devono comparire sempre almeno una regola della grammatica e il primo %% (che precede le regole della grammatica).

Un grammatica per Bison è divisa in tre parti distinte: una sezione *dichiarazioni*, una sezione *grammatica* e una sezione *programma*. La sezione *dichiarazioni* è usata per dichiarare informazioni necessarie sia a Bison che al file C da questi generato. Essa comprende l'include di header file, definizioni di tipo e dichiarazioni di funzioni esterne per le azioni. La sezione *grammatica* elenca le regole della grammatica e le azioni da eseguire per ciascuna di queste regole. La sezione *programma* è opzionale e può contenere qualsiasi programma C valido che verrà incluso assieme al parser nel file C che sarà generato. Comunque è preferibile lasciare vuota questa sezione e codificare queste routine come moduli separati.

4.2.4 Sezione epilogue

La sezione epilogue [21] è posta alla fine del file del parser, mentre il prologue è messo all'inizio del file. La sezione epilogue può contenere tutto il codice che si usa.

Spesso le funzioni, come *yylex()* e *yyerror()*, vengono definite in questa sezione. Se l'ultima sezione è vuota, si può omettere %% che separa la sezione delle regole grammaticali.

Il parser Bison, in sé, contiene molte macro e identificatori i cui nomi iniziano con yy o YY'

4.3 Lista di dichiarazioni usate in Bison

Qui riportiamo le dichiarazioni usate per definire una grammatica [21]:

%union

dichiara l'insieme di tipi data che possono essere assunti dai valori semantici;

%token

dichiara un simbolo terminale senza averne specificato l'associatività o la precedenza;

%right

dichiara un simbolo terminale con l'associatività a destra;

%left

dichiara un simbolo terminale con l'associatività a sinistra;

%nonassoc

dichiara un simbolo terminale che non è associativo. Se fosse usato in un contesto in cui potrebbe essere associativo si avrebbe un errore sintattico;

%type

dichiara il tipo di valori semantici per un simbolo terminale;

%start

indica il simbolo iniziale della grammatica, quindi riferisce a Bison quale dei non terminali debba essere considerato come la condizione di arrivo e il parser si fermerà quando raggiungerà questo non terminale; se %start manca, Bison userà il primo non terminale che incontra nella sezione grammatica;

%expect

dichiara il previsto numero di conflitti *shift-reduce*;

Per cambiare il comportamento di Bison, si usano le seguenti direttive:

%debug

nel file del parser, definisce la macro YYDEBUG, posta a 1 se non è stata definita;

%defines

scrive un file header (intestazione) contenente definizioni macro per i tokens di tipo name definiti nella grammatica. Se il file di output del parser è chiamato **name.c** allora questo file di header è chiamato **name.h**. Se YYSTYPE è stato già definito come macro allora viene dichiarato YYSTYPE nell'header di output. Quindi, se si usa %union con componenti che richiedono altre definizioni, o se definiamo YYSTYPE come macro, bisogna che queste definizioni vengano dichiarate in tutti i moduli mettendoli, per esempio, in un header che è incluso sia nel parser che in qualsiasi altro modulo che abbia bisogno di YYSTYPE. A meno che anche il parser sia

nell'header di output, la variabile *yylval* è dichiarata come variabile esterna. Se vengono usate delle locazioni, l'header di output dichiara YYLTYPE e *yylloc* usando un protocollo simile a YYSTYPE e *yylval*. Il file di output è essenziale se viene messa la definizione di *yylex* in un codice diverso da questo file, e questo avviene perché *yylex* si riferisce alle dichiarazioni e ai tipi di tokens del suddetto file di output;

%desctructor

specifica come il parser dovrebbe riprendere la memoria associata ai simboli scartati;

%file-prefix = "prefix"

specifica il prefisso dei nomi da usare per tutti i file output di Bison. I nomi sono scelti come se il file di input si chiamasse **prefix.y**;

%location

genera le posizioni della generazione del codice. Questo avviene solo nel caso in cui la grammatica usi il token speciale @n, ma se la grammatica non usa tale token e utilizza *%locatio*n allora si ha un messaggio di errore sintattico;

%name-prefix = "prefix"

rinomina i simboli esterni usati dal parser in modo che inizino con il prefisso yy. I simboli rinominati dal parser in C sono *yypars*e, *yylex*, *yyerror*, *yynerrs*, *yylval*, *yychar*, *yydebug* e *yylloc* (se sono usate le locazioni). Per esempio, se si usa *%name-prefix="c"*, i nomi si trasformano in *c_parse*, *c_lex* e così via.

%no-parser

non include nessun codice C nel file parser e genera solo le tabelle. Il file parser contiene solo indirizzi *#define* e dichiarazioni di variabili statiche. Questa opzione, inoltre, permette a Bison di scrivere il codice C per le azioni della grammatica in un file chiamato **file.act**;

%no-lines

non genera nessun *#line* dei comandi del preprocessore nel file parser. Di norma, Bison scrive questi comandi nel file parser in modo che il compilatore C e i debuggers associno gli errori e il codice oggetto al file sorgente (la grammatica);

%output = "file"

specifica il *file* per il file parser;

%pure-parser

richiede un programma di parser puro;

%require "version"

richiede la versione *version* di Bison;

%token-table

genera un array dei nomi dei tokens nel file parser. Il nome dell'array è *yytname*; *yytname[i]* è il nome dei tokens il numero di codice dei quali all'interno di Bison è *i*. I primi tre elementi di *yytname* corrispondono ai tokens predefiniti *$end*, *error* e *$undefined*; seguono i simboli definiti nel file della grammatica. La tabella include tutti i caratteri necessari a rappresentare i tokens in

Bison (costanti e stringhe). Quando viene specificato questo tipo di dichiarazione, Bison genera delle definizioni macro per queste macro: YYNTOKENS, YYNNTS, YYNRULES e YYNSTATES:

YYNTOKENS : il numero più alto di token più uno
YYNNTS : il numero dei simboli non terminali
YYNRULES : il numero di regole grammaticali
YYNSTATES : il numero degli stati del parser;

%verbose

scrive un file di output extra che contiene le descrizioni *verbose* degli stati dei parsers;

%yacc

richiede l'opzione --*yacc* se è richiesta tale dichiarazione; imita, cioè, Yacc con tutte le sue relative chiamate.

4.4 Valori semantici

I simboli terminali e i non terminali hanno sia valori sintattici che semantici. Ad esempio se il token NUM in una grammatica rappresenta un intero, il valore semantico di quel token è il valore effettivo di quel numero.

I valori semantici dei tokens vengono assegnati dallo scanner. Il valore semantico di un non terminale è gestito dal parser ed è generalmente il valore di ritorno di una routine di azione. Per accedere a questi valori si usano simboli speciali [21].

Consideriamo la regola:

```
expr  :  NUM       { $$ = $1 ; }
```

Quando il parser vede il token NUM lo riduce a expr e chiama la routine di azione $$ = $1; . Il simbolo $$ rappresenta il valore semantico del non terminale che viene definito, in questo caso, expr; il simbolo $1, invece, si riferisce al valore semantico del primo simbolo posto sul lato destro della regola, in questo caso NUM. Se traduciamo la regola in linguaggio corrente, questa potrebbe risultare come "assegna il valore semantico di NUM al valore semantico expr". In altre parole il valore del numero stabilito dallo scanner è accessibile ad ogni regola che usa expr. I valori semantici dei tokens vengono passati al parser dallo scanner usando la variabile globale *yylval*.

4.5 Come usare il parser Bison

Il file C che Bison genera consiste in una sola routine chiamata *yyparse()*. Questa routine dovrebbe essere chiamata da *main()* dopo aver concluso l'inizializzazione. Il controllo sarà restituito da *yyparse()* a *main()* quando il parser avrà completato le operazioni.

Lo scanner è una routine chiamata *yylex()*, codificata dall'utente o generata automaticamente da un analizzatore lessicale. La routine *yyparse()* chiama *yylex()* al fine di ricavare il token successivo.

I valori del tokens che *yylex()* restituisce sono definiti nell'header file generato con l'opzione -d. La routine *yylex()* deve anche inserire il valore semantico del token nella variabile globale *yylval*. Inoltre, *yylex()* deve restituire uno 0 quando l'input è finito (si noti che il parser è del tutto

indifferente ai luoghi da cui *yylex()* ricava i tokens, cioè a un file o alla tastiera). Qualsiasi routine di azione usata nella grammatica deve essere codificata separatamente dall'utente [21].

4.6 Gestione degli errori

Un errore sintattico si presenta quando il parser non può trovare una regola nella grammatica che soddisfi il token che *yylex()* gli ha restituito. Bison ha due metodi di gestione degli errori sintattici.

Il primo e più semplice è concludere il processo; *yyparse()* restituirà il valore 1 invece di 0, che indica il successo.

Il secondo metodo implica un tentativo di riparazione dell'errore. Bison salterà alcuni dei tokens nel tentativo di trovare un sequenza che possa continuare ad analizzare. L'errore non terminale verrà usato nella grammatica per dire a Bison in quale situazione può tentare la riparazione. Nella grammatica, ad esempio, la regola:

stmp : error SEM { *yyerrok;* }

dice a Bison di saltare i tokens finché non raggiunge un punto e virgola (token SEM) e di continuare come se fosse stato incontrato uno stmp.

L'azione da eseguire consiste in una macro speciale, *yyerrok()*, che dice a Bison di ignorare l'errore. L'azione potrebbe benissimo contenere altre istruzioni per la gestione degli errori.

Indipendentemente dalla possibilità o meno di far fronte a un errore, per prima cosa Bison passa a *yyerror()* una stringa da usare per la stampa di un messaggio di errore. Anche *yyerror()* è una funzione scritta dall'utente [21].

4.7 Generalizzazione del parser LR (GLR)

Bison produce parser deterministici che scelgono unicamente quando ridurre la regola e che riduzione applicare basandosi sulla stringa di input di un token extra di *lookahead*. Di conseguenza, Bison lavora su linguaggi context-free. Le grammatiche ambigue, poiché hanno stringhe con più di una possibile sequenza di riduzioni, non possono avere parser deterministici. Per concludere, ci sono linguaggi dove particolari scelte di Bison producono la perdita di informazioni necessarie all'analisi dell'input [21].

Quando viene usata la dichiarazione *%glr-parser* nel file in cui è posta la grammatica, Bison genera un parser che usa un algoritmo differente chiamato Generalizzazione LR (o GLR). Un parser Bison GLR usa lo stesso algoritmo base del parser Bison, ma si comporta diversamente nei casi in cui ci siano conflitti *shift-reduce* che non sono stati risolti dalle regole di precedenza o da un conflitto *reduce-reduce*. Quando un parser GLR incontra tale situazione, effettua dei tagli, uno per ogni possibile azione di *shift* o di *reduce*. Alcune pile possono incontrare conflitti ed effettuare tagli ulteriori con il risultato che, anziché avere una sequenza di stati, la pila di un parser Bison GLR è un albero di stati [21].

In effetti ogni pila rappresenta una stima rispetto a quanto risulta da un'analisi approfondita. L'input può indicare, anche, che una stima sia errata e in tal caso la pila utilizzata viene eliminata, in caso contrario, le azioni semantiche generate sono conservate in ogni pila, piuttosto che essere eseguite immediatamente. Quando una pila viene eliminata, le azioni semantiche che sono state salvate non vengono mai eseguite. Quando una riduzione induce due pile a diventare equivalenti, il loro insieme di azioni semantiche viene salvato con lo stato che risulta da tale riduzione. Diciamo che due pile sono equivalenti quando entrambe sono rappresentate dalla

stessa sequenza di stati e ogni coppia di stati rappresenta un simbolo grammaticale che produce lo stesso segmento di flusso di tokens in input [21].

Ogni volta che il parser fa una transizione da molti stati ad uno stato, il parser ritorna al normale algoritmo di parsing LALR(1) dopo aver risolto ed eseguito le azioni salvate precedentemente. Dopo questa transizione, alcuni degli stati sulla pila saranno valori semantici che sono insiemi (realmente dei multi insiemi) di possibili azioni. Il parser prova a selezionare una delle azioni la cui la regola ha la più alta precedenza, data dalla dichiarazione *%dprec*. Altrimenti, se le azioni alternative non hanno un ordine di precedenza, ma la stessa funzione di fusione è dichiarata per le regole utilizzate con la dichiarazione *%merg*, Bison risolve e valuta tali regole chiamando la funzione di fusione sul risultato. In caso contrario, Bison segnala un'ambiguità nella grammatica [21].

È possibile usare una struttura dati ad albero per il parser GLR che permette l'elaborazione di qualsiasi grammatica LALR(1) in tempo lineare O(n), dove n è la dimensione di input, nel caso peggiore per qualsiasi grammatica non ambigua non necessariamente LALR(1) in tempo $O(n^2)$, e nel caso peggiore per qualsiasi grammatica context-free in tempo $O(n^3)$.

Tuttavia Bison attualmente usa una struttura dati semplice che richiede tempo proporzionale alla lunghezza dell'input e al numero massimo di pile richieste per qualsiasi prefisso dell'input. Quindi, grammatiche ambigue e non deterministiche possono richiedere tempo e spazio esponenziale. Solitamente il non determinismo, in una grammatica, è locale e pertanto il parser "è in dubbio" solo su alcuni tokens. Perciò questo tipo di struttura dati usato da Bison dovrebbe essere, generalmente, sufficiente per l'analisi di una grammatica. Sulle grammatiche di tipo LALR(1), Bison utilizzando la struttura di default è un po' più lento [21].

4.8 Altri parser generators per grammatiche LALR(1)

APaGeD (Attribute Parser Generator for D) [22,23]: genera parser LL e parser LALR(1) basati su GLR, permettendo di avere grammatiche flessibili e maggiori prestazioni. Fornisce, inoltre, un buon trattamento degli errori con espliciti messaggi di errore automaticamente generati, del tipo "file(line): found A, expected B, C or D".

Beavor [24]: prende una grammatica context-free e la converte in una classe Java che implementa un parser per il linguaggio della grammatica. Beavor accetta grammatiche espresse in Extended Backus-Naur form (EBNF)

BYACC (Berkeley Yacc) [25]: è la migliore variante di Yacc disponibile. Contrariamente a Bison, è scritto per evitare le dipendenze su un particolare compilatore.

BYACC/J [26]: è un'estensione del generatore Yacc ottenuta aggiungendo il flag -J e genera il codice sorgente in java, cioè è Yacc nel linguaggio java. È molto veloce e, inoltre, non richiede librerie di routine, poiché il codice generato è l'intero parser.

CSTools [27]: è un compilatore scritto con gli stessi strumenti di Yacc, usando C# come linguaggio di implementazione. Tali strumenti sono scritti usando tecniche ad oggetti e servono per una migliore comprensione nelle procedure standard.

CUP (Constructor of Useful Parsers) [28]: viene usato allo stesso modo di Yacc dal momento che mette a disposizione la maggior parte delle caratteristiche di Yacc. CUP, tuttavia, è scritto in java, nelle sue specifiche include il codice java e produce parsers che sono implementati in java.

GOLD [29]: è un parser generator che adotta l'approccio LALR per l'analisi dei linguaggi e un automa a stati finiti deterministici (DFA) per l'analisi lessicale delle diverse classi di token. A differenza di molti altri strumenti, GOLD parser non richiede di incorporare le azioni del parser nella grammatica (come annotazioni): GOLD infatti analizza la grammatica e salva le tabelle di parsing in un file separato, che può poi essere caricato e usato dal parser engine. Al momento GOLD supporta Java e tutti i linguaggi su piattaforma .NET e su ActiveX, Delphi e Visual C++; i sorgenti sono disponibili per Java, C#, Delphi, Visual C++, Visual Basic (5 e .NET).

Happy [30]: è un sistema di parser generator per Haskell, simile a Yacc per C. Happy prende un file che contiene come specifica una grammatica BNF e produce un modulo Haskell che contiene un parser per la grammatica. Il parser in questione è flessibile perché si possono avere molti parser Happy nello stesso programma e molti punti di entrata nella singola grammatica. Happy, infine, è usato anche per le grammatiche GLR.

Jacc (Just Another Compiler Compiler) [31]: è un parser generator per java, compatibile con il parser Yacc; genera parser bottom-up *shift-reduce* per grammatiche non ambigue, funziona su molte piattaforme di sviluppo java e dispone di un meccanismo per la gestione dei messaggi di errore.

LEMON [32]: è un parser generator per C e C++. Lavora allo stesso modo di Bison e Yacc ma è differente da loro in quanto usa una diversa sintassi grammaticale per ridurre il numero di errori di codifica. Lemon usa anche un sistema di parsing più sofisticato e più veloce di Yacc e Bison, che è sia rientrante che thread-safe; implementa, inoltre, delle features che possono essere usate per limitare la perdita di risorse, e questo fa sì che Lemon sia adatto all'uso in programmi a lunga esecuzione, quali interfacce grafiche e controllori embedded.

LPG [33]: conosciuto meglio come JikesPG, è un generatore di analizzatori lessicali e di parser LALR nel linguaggio Java. Le grammatiche di input sono scritte in regole BNF. LPG supporta il backtracking per risolvere l'ambiguità e genera il codice per la rappresentazione astratta dell'albero sintattico. Un punto forte di tale parser è il metodo di recupero degli errori.

Capitolo 5

Parser generator Essence

5.1 Parser generator Essence: introduzione

Essence [36] è un parser generator per le grammatiche LR(k) e SLR(k), scritto in Scheme48. Tale generatore effettua il recupero degli errori, è altamente efficiente e deriva da un parser generale che prende in input una grammatica come parametro. Il modo di usare Essence è un po' differente da quello degli altri tipi di parsers. La parte principale, come in ogni parsers, è una grammatica context-free. Essence fornisce una nuova sintassi, della forma *define-grammar*, che include un linguaggio per le grammatiche attributate context-free in Scheme48.

Data una grammatica, l'analisi può essere fatta in due modi [36]:

- La procedura di un generico parser è quella di accettare una grammatica, un metodo di analisi (SLR o LR), la dimensione del *lookahead* e un input; il parser, quindi, fornisce il risultato dell'analisi effettuata. Questo metodo è molto lento, è buono per incrementare lo sviluppo del parser, ma poco pratico per la produzione di altri parsers.

- Un parser generator produce un parser specializzato da una grammatica, un metodo di analisi e un *lookahead*. Il parser specializzato accetta solo un input come argomento e funziona in maniera opposta al parser generale; è altamente efficiente e quindi buono per la produzione di parsers.

5.2 Struttura di una grammatica per Essence

Le grammatiche context-free sono la parte essenziale per la generazione di parser. Essence specifica le cosiddette grammatiche S-attributate con la valutazione delle regole per gli attributi sintetizzati. Si assume che ogni nodo dell'albero di analisi trasporti un'istanza di un attributo sintetizzato, e che una grammatica di Essence fornisca un'espressione descrivendo come stima un attributo con ogni produzione.

La struttura *grammar* definisce la struttura delle grammatiche.

Una grammatica di Essence consiste in due oggetti: una rappresentazione di tale grammatica e una enumerazione necessaria a dare un codice simbolico alle parti di input da analizzare [36].

L'istruzione *define-grammar* definisce entrambi gli oggetti nel modo seguente:

(define-grammar <variabile$_1$> <variabile$_2$> <NonTerminali> <Terminali> <simbolo iniziale> <regole>)

5.2.1 Struttura sintattica

<NonTerminali> ha la forma

(<nonterminale> …)

dove ogni <nonterminale> è un <identificatore>.

<Terminali> è allo stesso modo della forma

(<terminale> …)

con <terminale> un <identificatore>.

<Terminali> e <NonTerminali> possono essere disgiunti.

<simbolo iniziale> può essere un <nonterminale>.

<regole> ha la forma:

(<regola> …)

dove ogni <regola> ha la forma

((<nonterminale> <simbolo grammaticale> …) <attributi>)

dove ogni <simbolo grammaticale> è ancora un <nonterminale>, un <terminale> o $error; mentre <attributi> è un'espressione nel linguaggio Scheme [36].

5.2.2 Struttura semantica

Define-grammar definisce una grammatica context-free con la numerazione dei suoi simboli. Prende come primo argomento la rappresentazione degli oggetti della grammatica, come secondo argomento

il tipo di enumerazione per i suoi simboli, come terzo argomento una lista di simboli non terminali, come quarto argomento una lista di terminali e come quinto argomento il simbolo iniziale (uno dei non terminali) da cui è derivata la lista delle regole grammaticali [36].

Le regole grammaticali consistono in una produzione – una lista con simboli non terminali a sinistra della produzione e una lista di simboli terminali e non terminali a destra della produzione – e in un'attribuzione.

Nel linguaggio Scheme48 l'attribuzione è un'espressione che può avere variabili libere *$i*, dove *i* varia da 1 al numero di simboli della parte destra della produzione. Durante la fase di analisi, quando viene valutata l'attribuzione, il parser Essence lega *$i* all'istanza dell'attributo dell'*i*-esimo simbolo della parte destra della produzione [36].

Qui riportiamo un semplice **esempio** di grammatica per le espressioni aritmetiche:

```
( define-grammar g10 g10-symbol
        ( E  T  P )
        ( + - * / l r n )
        E
        ( ( ( E  T )  $1)
          ( (E  T + E)    ( +  $1  $3 )
          ( ( E  T  - E )  ( -  $1  $3 )
          ( ( T  P )  $1 )
          ( ( T  P * T )   ( *  $1  $3 )
          ( ( T  P / T )   ( /  $1  $3 )
          ( ( T  n )  $1 )
          ( ( P  l  E  r )  $2 ) ) )
```

Questa definizione stabilisce una numerazione di tipo g10-symbol con i componenti (in questo ordine) [36]:

- $Start (per il simbolo iniziale generato da *define-grammar*);
- i non terminali nell'ordine in cui sono stati definiti in *define-grammar*;
- $error;
- i terminali nell'ordine in cui sono stati definiti in *define-grammar*.

5.3 Esecuzione del parser Essence

Essence fornisce dei parsers generali che accettano una grammatica come input e un'analisi "right away" permettendo l'incremento dello sviluppo degli attributi grammaticali. Essence funziona con implementazioni differenti di analizzatori LR. Il file di configurazione *packages.scm* contiene le definizioni per le strutture dell'interfaccia definita da *parser-interface*. Il file *packages.scm*, inteso per l'uso delle produzioni, è nella struttura *cps-lr* dove l'implementazione è in *cps/cps-lr.scm*; *parser-interface*, invece, descrive un legame chiamato *parse* composto dai seguenti componenti [36]:

- *Grammar*: è una grammatica definita in *define-grammar*;
- *Lookahead*: è un numero intero non negativo che denota il *lookahead* del parser;
- *Method*: è un simbolo, *lr* o *slr*, che specifica quale metodo di parsing viene usato (uno per il parser LR e l'altro per il parser SLR);
- *Input*: è un flusso (secondo la struttura del flusso – stream) costituito dagli enumerandi dei terminali di *grammar* in relazione con il loro valore. Questo è l'*input* reale al parser. Il flusso contiene i numeri

ordinali dei valori enumerati. Questi sono ottenuti facilmente attraverso la struttura Scheme48. In assenza di un'implementazione della grammatica e dell'enumerazione, la lista dei componenti sopra elencati descrive il tracciato tra i simboli grammaticali e gli enumerandi; gli enumerandi sono in base 0.

La procedura *parse* restituisce il risultato della valutazione dell'albero di analisi derivato da una stringa di input.

Se una grammatica contiene produzioni contenenti il simbolo $error, il parser cercherà di recuperare l'errore, quando è possibile [36].

5.4 Implementazione dello stream di input

Gli analizzatori Essence si appoggiano sugli streams per implementare le sequenze dei terminali di input. Gli analizzatori seguono solo tre procedure per l'implementazione. Una possibile implementazione è l'uso delle liste [36].

5.4.1 Interfaccia dello stream

La sezione dell'interfaccia dello stream relativo agli analizzatori Essence contiene questo tipo di procedure [36]:

- *stream-car stream*: restituisce il primo elemento di uno stream non vuoto;
- *stream-cdr stream*: una volta applicato ad uno strema non vuoto dello *stream*, restituisce un nuovo stream con gli stessi elementi

dello *stream* precedente;

- *stream-empty? stream*: restituisce #t quando vengono applicati uno stream vuoto e uno stream non vuoto.

5.4.2 Struttura dello stream

Un esempio di implementazione degli streams nella distribuzione di Essence è **common/stream.scm**, definito in **packages.scm**, che è anche l'implementazione della struttura *stream* di Scheme48. Oltre alle procedure descritte precedentemente fornisce una procedura di creazione e una di trasformazione da e nelle liste [36]:

make-stream state-transformer state

dove *make-stream* genera uno stream con *state*, lo stato iniziale, e con *transformer*, lo stato di trasformazione. Uno stato dello stream rappresenta uno stato nello svolgimento dello stream medesimo e, come tale, produce gli elementi dello stream. *State-transformer* accetta uno stato come argomento e restituisce una coppia che ha come primo elemento lo *stream* da *car* e come secondo elemento un nuovo stato da *cdr* [36].

Tali procedure invocano *state-transformer* solo in modalità single-thread; esse lo eseguiranno solo in stato di accesso e lo eseguiranno una sola volta. Dopodiché, esse lo eseguiranno una volta allo stato restituito dallo *state-transformer* e così via.

(*list* → *stream list*): produce uno stream esattamente con gli elementi di *list*;

(*list* → *list stream*): produce una lista esattamente con gli elementi di *stream*. *Stream* può essere infinito.

5.5 Recupero degli errori

I parsers Essence possono effettuare il recupero degli errori allo stesso modo dei parsers Yacc e Bison. L'idea base è che l'utente che ha creato una grammatica possa specificare le produzioni speciali di errore nei posti critici della grammatica per "interferire" con l'analisi degli errori.

Questo permette di stampare messaggi di errore adatti alla valutazione corretta di un attributo [36].

Le produzioni di errore contengono $error, un simbolo grammaticale speciale, nella parte destra della produzione. ($error non può essere esplicitamente dichiarato come un simbolo terminale o non terminale in *define-grammar*).

Quando un errore si presenta durante la fase di analisi, il parser Essence finge di aver trovato nella stringa di input il simbolo $error; pertanto andrà all'ultimo stato LR capace di accettare $error come successivo simbolo nella stringa di input. Essence, inoltre, scarta dall'input i simboli terminali fino al successivo simbolo terminale accettato dopo aver trovato il simbolo $error; dopo ciò riprende la fase di analisi nella sua normalità. Per impedire l'invio di numerosi messaggi, il parser valuta i successivi tre simboli terminali nella stringa di input cercando sempre di recuperare l'errore precedentemente trovato [36].

Riportiamo un **esempio** di espressioni grammaticali in cui sono presenti errori:

```
( define-grammar  g10-error g10-error-symbol
        ( E T P )
        ( + - * / l r n )
        E
        ( ( ( E  T ) $1 )
          ( ( E  $error ) 0 )
          ( ( E  T  +  E ) ( +  $1  $3 ) )
          ( ( E  T  -  E )  ( -  $1  $3 ) )
          ( ( T  P ) $1 )
          ( ( T  P  *  T ) ( * $1 $3 ) )
          ( ( T  P  /  T )  ( / $1 $3 ) )
          ( ( P  n ) $1 )
          ( ( P  l  E  r ) $2 )
          ( ( P  l  $error r )  0 ) ) )
```

Dopo la prima regola che contiene il simbolo $error, il parser, quando incontra un errore all'interno di un'espressione con parentesi, salta tutti i simboli finché non trova la successiva parentesi chiusa, da cui riprende la sua normale fase di analisi [36].

5.6 Altri parser generators per grammatiche LR o GLR

Elkhound [34]: è un parser generator simile a Bison. I parsers generati usano l'algoritmo di parsing GLR, che funziona con grammatiche context-free, mentre i parsers LR (come Bison) richiedono che le grammatiche siano LALR(1). Effettuando l'analisi con le grammatiche

context-free, Elkhound presenta due vantaggi: ha *lookahead* illimitati e supporta grammatiche ambigue.

Menhir [35]: progettato da François Pottier e da Yann Régis-Gianas, è un parser generator LR(1) per il linguaggio di programmazione ad oggetti Caml. Menhir, dunque, compila le specifiche della grammatica LR(1) con il codice OCaml.

APaGaD, Happy: vedere paragrafo 4.7.

Capitolo 6

Parser generator Coco/R

6.1 Parser generator Coco/R: introduzione

Coco/R (Compiler Compiler Generating Recursive descent parsers) è un generatore di compilatori che prende come ingresso una grammatica attributata di un linguaggio sorgente e genera uno scanner o un parser per questo linguaggio. Lo scanner lavora come un automa a stati finiti deterministici mentre il parser usa ricorsioni discendenti. L'utente deve dichiarare una classe *main()* che chiama il parser e le classi semantiche (per esempio i simboli delle tabelle o un generatore di codice) che vengono usati tramite le azioni semantiche nel parser (figura 8) [41].

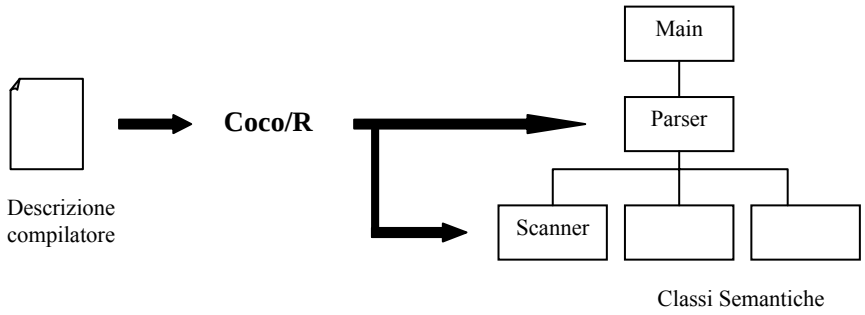


Figura 8. Input ed output di Coco/R

6.2 Caratteristiche di Coco/R

Il parser è specificato da un insieme di produzioni con attributi e azioni semantiche. Tali produzioni tengono conto delle alternative, delle ripetizioni e delle parti facoltative. Coco/R traduce le produzioni in un parser discendente ricorsivo efficiente [41].

Gli attributi <*out name*> specificano i parametri dei simboli. Ci sono attributi di input <*x, y*> e attributi di output <*out z*> o <*ref z*>.

I simboli non terminali possono avere tutti i numeri degli attributi di input e di output (java permette solo un attributo di output, che può, tuttavia, essere un oggetto della stessa classe). I simboli terminali non hanno attributi espliciti, ma i tokens restituiti dallo scanner contengono informazioni che possono essere viste come attributi. Tutti gli attributi sono valutati durante la fase di analisi.

Il simbolo speciale ANY può essere usato per denotare una serie di tokens complementari.

Le azioni semantiche possono essere disposte, all'interno della grammatica, alla fine di ogni produzione, e vengono eseguite dal parser generator. Possono, inoltre, contenere dichiarazioni scritte nel linguaggio del parser generator (Java, C#, C++).

Principalmente, Coco/R può operare su grammatiche LL(1) ma è in grado di svolgere il suo lavoro anche su grammatiche non LL(1) usando i cosiddetti *resolvers* che prendono decisioni di analisi basandosi su multi-simboli di *lookahead* e su informazioni semantiche.

Ogni produzione può avere proprie variabili locali ma, oltre a queste, può dichiarare variabili globali o metodi, che sono usati nei campi e nei metodi del parser. Le azioni semantiche possono anche accedere ad altri oggetti o metodi scritti dall'utente.

Coco/R controlla la grammatica per vedere se c'è totalità, consistenza e non ridondanza. Inoltre segnala i conflitti LL(1).

I conflitti nelle grammatiche LL(1) possono essere risolti attraverso i simboli di *lookahead* o attraverso controlli semantici.

Il messaggio di errore stampato dal parser generator può essere configurato secondo le esigenze dell'utente.

Il parser e lo scanner generators possono essere specificati per appartenere ad un determinato *namespace* [41, 42].

6.3 Struttura generica di un parser generator Coco/R

Gli elementi basilari di Coco/R [41] sono identificatori, numeri, stringhe e costanti, definiti come segue:

Ident = *letter { letter | digit } .*
number = *digit { digit } .*
string = *' " '{ anyButQuote } ' " ' .*
char = *'\' ' anyButApostrophe '\' ' .*

Le lettere maiuscole sono distinte dalle lettere minuscole. Le stringhe non devono estendersi attraverso linee multiple. Sia le stringhe che le costanti possono contenere le seguenti sequenze:

\\ backslash	\r carriage return	\f form feed
\" quote	\n new line	\a bell
\0 null character	\t horizontal tab	\b backspace
\' apostrophe	\v vertical tab	\uxxxx hex char value

Le seguenti parole sono parole chiave per Coco/R:

ANY	CONTEXT	IGNORE	PRAGMAS
CHARACTERS	END	IGNORECASE	PRODUCTIONS
COMMENTS	FROM	Out	SYNC
COMPILER	IF	WEAK	
TO	TOKENS		

I commenti sono racchiusi tra /* e */ se sono distribuiti su più linee altrimenti si può mettere il solo simbolo // all'inizio del commento, se questo è distribuito su un'unica linea.

6.3.1 Struttura del compilatore Coco1/R

Questa sezione si occupa del compilatore Coco1/R, usato come linguaggio di input per Coco/R. La descrizione del compilatore consiste in un insieme di regole grammaticali che descrivono la struttura lessicale e sintattica di un linguaggio.

La struttura di un compilatore Coco1/R è data dalla forma seguente:

Coco1 = [Imports]
"COMPILER" ident
[GlobalFieldsAndMethods]
ScannerSpecification
ParserSpecification
"END" ident '.'
.

dove il nome dopo la parola chiave *COMPILER* è il nome della grammatica e allo stesso modo viene riscritto dopo la parola chiave *END*. Il nome della

grammatica denota anche il simbolo non terminale iniziale (start symbol).

La specifica del parser, parte principale per la descrizione del compilatore, può contenere una produzione per questo simbolo, le produzioni e la grammatica attributata, che specificano la sintassi del linguaggio per l'analisi e la relativa traduzione [41,42].

6.3.2 Le produzioni

Le produzioni specificano la struttura sintattica dei simboli non terminali. Esse consistono in una parte sinistra e una parte destra separate da uno stesso simbolo. La parte sinistra specifica l'insieme dei non terminali con i suoi attributi formali e le variabili locali della produzione, la parte destra consiste in espressioni che specificano la struttura dei non terminali così come la sua traduzione nella forma di attributi e azioni semantiche [41,42].

Le produzioni possono essere date in qualsiasi ordine e, al loro interno, si possono fare riferimenti espliciti ai non terminali. Per ogni non terminale ci deve essere esattamente una produzione. In particolare, ci deve essere una produzione per il nome della grammatica, che è il simbolo di partenza della grammatica medesima [41].

6.3.3 Le azioni semantiche

Un'azione semantica è una parte del codice scritta nel linguaggio di Coco/R. È eseguita dal parser generator nella posizione in cui è stata specificata nella grammatica. Le azioni semantiche sono semplicemente

copiate dal parser generator non controllato da Coco/R. Inoltre, un'azione semantica può anche contenere dichiarazioni di variabili locali. Ogni produzione ha il proprio insieme di variabili locali, che è mantenuto nelle produzioni ricorsive. L'azione semantica facoltativa sulla parte sinistra della produzione (LocalDecl) è usata per le dichiarazioni, ma le variabili possono anche essere dichiarate in ogni altra azione semantica. Le azioni semantiche, inoltre, possono non solo accedere alle variabili locali ma anche ai metodi dichiarati all'inizio della grammatica attributata [41,42].

6.3.4 Gli attributi

Le produzioni sono considerate come metodi di parsing. L'occorrenza di un non terminale sulla parte destra di una produzione può essere vista come una chiamata al metodo per l'analisi di questo non terminale. I non terminali possono avere attributi che corrispondono come parametri ai metodi di analisi per i non terminali. Ci sono attributi di input, che sono usati per passare valori alle produzioni di un non terminale, e attributi di output, che sono usati per far tornare il valore dalle produzioni ad un non terminale visitato (cioè al posto in cui il non terminale occorre in una certa altra produzione) [41,42].

Come con i parametri, distinguiamo tra attributi formali, che sono specificati nelle dichiarazioni dei non terminali nella parte sinistra della produzione, e attributi attuali, che sono specificati nelle occorrenze dei non terminali nella parte destra della produzione.

Gli attributi sono inclusi tra angoli (cioè <…>). Se gli attributi contengono gli operatori < oppure > o tipi generici come List<T> allora l'attributo può essere scritto come <. … .>.

Coco/R controlla che i non terminali con attributi siano sempre usati con attributi e che i non terminali senza attributi siano sempre usati senza attributi. Tuttavia, Coco/R non controlla la corrispondenza fra gli attributi formali e quelli attuali, operazione lasciata al compilatore del linguaggio di uscita [41,42].

6.3.5 Attributi dei simboli terminali

I simboli terminali non hanno attributi in Coco1/R. Ad ogni modo lo scanner restituisce il valore del token (cioè la stringa di tokens) come pure il numero di linea e la colonna dei tokens. Queste informazioni possono essere viste come attributi di output di quel token [41].

6.3.6 Il simbolo ANY

Nelle produzioni della grammatica il simbolo ANY denota ogni token che non è un'alternativa al simbolo ANY nella produzione corrente. La seguente produzione, per esempio, produce una lista di attributi in Coco1/R e restituisce il numero di caratteri tra simboli ad angolo:

```
Attributes  < out int len >  =
            '<'          (.  int beg = t.pos + 1;  .)
            {ANY}
            '>'          (. len = t.pos – beg;  .)  .
```

In questo esempio il simbolo > è un'alternativa implicita al simbolo ANY. Il significato della produzione è che ANY è in relazione con tutti i tokens ad

eccezione del simbolo >. L'attributo t.pos indica la posizione, nel testo, del token riconosciuto più di recente [41].

6.4 Conflitti LL(1)

Il parser discendente ricorsivo richiede che la grammatica sia LL(1) perché, ad un certo punto della grammatica, il parser possa decidere in base al simbolo di *lookahead* quale alternativa selezionare.

Prendiamo in esame un produzione non LL(1):

```
Statement = ident '=' Expression ';'
          | ident '(' [ActualParameters] ')' ';'
          | ...
          .
```

Le alternative iniziano con il simbolo *ident*. Quando il parser inizia con il simbolo *Statement* e *ident* è il successivo token di input, il parser non può distinguere tra le due alternative. Tuttavia, questa produzione può essere trasformata nel modo seguente:

```
Statement = ident ( '=' Expression ';'
                  | '(' [ActualParameters] ')' ';'
                  )
          | ...
          .
```

dove tutte le alternative iniziano con un simbolo distinto e i conflitti LL(1) sono stati risolti [41,42].

I conflitti LL(1) possono risultare non solo dalle alternative esplicite, come nell'esempio sopra riportato, ma anche dalle alternative implicite, attraverso delle espressioni.

Controllare manualmente tutte queste alternative sarebbe non conveniente e faticoso, ma fortunatamente Coco/R lavora automaticamente.

Per **esempio**, la grammatica:

A = *(a* | B C *d)* .
B = *[b] a* .
C = *c {d}* .

stamperà il seguente avvertimento:

LL1 warning in A : a is start of neveral alternatives
LL1 warning in C : d is start & successor of deletable structure

Il primo conflitto è presente perché B può iniziare con *a*. Il secondo conflitto viene dal fatto che *c* può essere seguito da *d* e così il parser non sa se effettuare un'altra iterazione di {*d*} in *c* o terminare con *c* e continuare con *d*.

Si noti che i conflitti riscontrati da Coco/R non vengono segnalati come errori ma come avvertimenti. Il parser ogni volta che vede due o più alternative che possono iniziare sempre con lo stesso token sceglie sempre il primo [41,42,43].

6.5 Risolvere i conflitti LL(1) con la trasformazione della grammatica

Se Coco/R segnala un conflitto LL(1) l'utente dovrebbe provare ad

eliminarlo trasformando la grammatica come indicato nei seguenti esempi [41,42,43].

- **Fattorizzazione**

La maggior parte dei conflitti LL(1) può essere risolta attraverso la fattorizzazione, cioè unendo le parti comuni delle alternative in conflitto e spostandole nella parte anteriore della produzione. Ad esempio, la produzione

$$A = abc \mid abd$$

può essere trasformata in

$$A = ab \{c \mid d\}$$

- **Ricorsione a sinistra**

La ricorsione a sinistra presenta sempre un conflitto LL(1). Ad esempio, nella produzione

$$A = Ab \mid c$$

entrambe le alternative iniziano con *c* (perché first(A) = {*c*}). Tuttavia la ricorsione a sinistra può essere trasformata sempre in un'iterazione, e così, per esempio, la produzione precedente diventa

$$A = c\{b\}.$$

- **Conflitti duri**

Qualche conflitto LL(1) non può essere risolto con la semplice

trasformazione della grammatica. Consideriamo le seguenti produzioni della grammatica:

```
Expr    =  Factor { '+' Factor}
Factor  =  '(' ident ')' Factor   /* type cast*/
        |  '(' Expr ')'           /* nested expression */
        |  ident
        |  number
        .
```

Il conflitto si presenta perché due alternative di *Factor* iniziano con (e inoltre perché *Expr* può anche essere derivato da un *ident*. Tale conflitto non può essere risolto attraverso una trasformazione della grammatica ma l'unico modo è quello di guardare *ident* seguito da (: se denota un tipo, il parser deve selezionare la prima alternativa, altrimenti una seconda alternativa. La soluzione di questo conflitto verrà discussa nella sezione successiva.

- **Scrittura leggibile**

Alcune trasformazioni della grammatica possono diminuire la leggibilità della grammatica. Consideriamo il seguente esempio:

```
UsingClause  =  "using"  [ ident '=' ]  Qualident  ';' .
Qualident    =  ident { '.' ident } .
```

Il conflitto è su *UsingClause* dove *[ident '=']* e *Qualident* iniziano con *ident.* Anche se questo conflitto può essere eliminato trasformando la produzione in

```
UsingClause = "using" ident ( { '.' Ident }
                            | '=' Qualident
                          ) ';' .
```

la leggibilità viene meno. La soluzione di questo conflitto verrà discussa nella sezione successiva.

- **Scrittura semantica**

La fattorizzazione non è adatta perché il processo semantico è differente per i vari conflitti, per esempio:

```
A = ident ( . x = 1; . ) { ',' ident ( . n++; .) } ':'
  | ident ( . Foo(); . ) { ',' ident ( . Bar(); .) } ';' .
```

Le parti comuni a queste due alternative non possono essere fattorizzate, perché ogni alternativa ha un suo processo semantico. La struttura semantica può essere risolta con il metodo trattato nel successivo paragrafo.

6.6 Come risolvere i conflitti LL(1)

Un conflitto *resolver* è un'espressione booleana inserita, all'interno della grammatica, all'inizio della prima alternativa, e il parser decide, usando un multi simbolo di *lookahead* o un controllo semantico, se questa è in relazione con l'attuale input. Se il *resolver* restituisce true, allora l'alternativa prefissata dal *resolver* è selezionata, altrimenti la successiva alternativa sarà quella controllata.

Un conflitto *resolver* sarà scritto nel modo seguente:

Resolvers = "IF" '(' ... any expression ... ')' .

dove *any* è un'espressione booleana che può essere scritta tra parentesi. In molti casi è una funzione che restituisce true o false.

I conflitti trattati nella sezione precedente si possono pertanto risolvere nel modo seguente:

UsingClause = "using" [IF(IsAlias()) ident '='] Qualident ';' .

dove *IsAlias()* è un metodo, prestabilito dall'utente, che legge i due tokens successivi. *IsAlias()* restituisce true se *ident* è seguito da '=' altrimenti restituisce false [41,42].

6.6.1 Conflitti risolti da un multi simbolo di lookahead

Il parser generato memorizza con due variabili globali il token riconosciuto più recentemente e il token corrente di *lookahead*:

```
Token  t ;      // il token riconosciuto più recentemente
Token  la ;     // token di lookahead
```

Lo scanner generato implementa il metodo *Peek()* che può essere usato per leggere, spostandosi in avanti, il token di *lookahead* senza rimuovere i tokens dalla stringa di input. Il parser quando richiama lo scanner restituisce ancora questi tokens [41,42].

Con il metodo *Peek()* possiamo implementare *IsAlias()* nel modo seguente:

```
bool IsAlias()  {
        Token next  =  scanner.Peek() ;
        Return la.kind  = = _ident  &&  next.kind  = =  _eql ;
        }
```

Il conflitto discusso nel paragrafo precedente può essere risolto con questa produzione:

```
A  =  IF  ( FollowedByColon() )
        ident ( . x = 1; . ) {',' ident ( . x++; .) } ':'
    | ident ( . Foo(); . ) { ','  ident ( . Bar(); . ) } ';' .
```

Qui di seguito presentiamo l'implementazione della funzione *FollowedByColon()*:

```
bool   FollowedByColon()  {
                Token x = la;
                while (x.kind = = _comma || x.kind = = _ident)
                        x = scanner.Peek();
                return x.kind  = =  _colon;
        }
```

6.6.2 Nomi dei tokens

È conveniente riferirsi ai numeri di tokens con nomi quali *_ident* o *_comma*. Coco/R genera tali nomi per tutti i tokens dichiarati, nella sezione *TOKENS*, all'interno delle specifiche dello scanner [41]. Per esempio, se i tokens sono dichiarati in questo modo:

```
TOKENS
    Ident    =  letter { letter | digit } .
    number   =  digit { digit } .
    eql      =  '=' ;
    comma    =  ',' .
    colon    =  ':' .
```

Coco/R genererà nel parser le seguenti dichiarazioni:

```
const int _EOF  =  0 ;
const int _ident  =  1 ;
const int _number  =  2 ;
const int _eql  =  3 ;
const int _comma = 4 ;
const int _colon = 5 ;
```

I nomi dei tokens sono preceduti da un *underscore* per evitare i conflitti con parole chiave riservate ad altri identificatori. Normalmente la sezione *TOKENS* contiene solo dichiarazioni per le classi di tokens come *ident* o *number*. Tuttavia, se il nome del token letterale è necessario, tale token deve essere dichiarato nella sezione *TOKENS*. Nelle produzioni della grammatica questi tokens possono riferirsi ad un nome (*_comma*) o ad un valore (‘,’).

6.6.3 Traslazione dei conflitti resolvers

Coco/R tratta i *resolvers* come azioni semantiche e li copia

semplicemente nel parser generator, nella posizione in cui compaiono nella grammatica [41,42].

Per **esempio**, la produzione:

```
UsingClause  =  "using" [ IF ( IsAlias() )  ident '=' ]  Qualident ';'.
```

è traslata nel seguente metodo di parsing:

```
void UsingClause() {
        Expect( _using );
        if (IsAlias()) {
                Expect( _ident );
                Expect( _eql );
        }
        Qualident();
        Expect( _semicolon );
}
```

6.6.4 Conflitti risolti attraverso informazioni semantiche

Il parser quando si presenta un conflitto *resolver* può basare la propria decisione non solo sui tokens di *lookahead* ma anche su altre informazioni. Per esempio potrebbe accedere alla tabella dei simboli e scoprire una proprietà semantica circa quel token [41,42]. Consideriamo il seguente conflitto LL(1) tra tipi ed espressioni:

```
Expr    =  Factor { '+' Factor } .
Factor  = '(' ident ')'  Factor        /* type cast */
        |  '(' Expr ')'                /* nested expression */
```

```
        | ident | number .
```

Poiché *Expr* può iniziare con un *ident*, anche il conflitto può essere risolto controllando se questo *ident* denota un tipo o un altro oggetto:

```
Factor = IF ( IsCast() )
                '(' ident ')' Factor          /* type cast */
        | '(' Expr ')'                        /* nested expression */
        | ident | number .
```

IsCast() trova *ident* nella tabella dei simboli e ritorna true, se è di tipo nome:

```
bool IsCast() {
    Token x = scanner.Peek() ;
    if (la.kind = = _lpar && x.kind = = _ident) {
        object obj = symTab.Find(x.val) ;
        return obj != null  &&  obj.kind = = Type;
    } else return false;
}
```

6.6.5 Disporre i resolvers in modo corretto

Coco/R controlla se i *resolvers* sono disposti correttamente attraverso le segueti regole [41,42]:

1. se due alternative iniziano con lo stesso token, il *resolver* deve essere disposto nella parte anteriore a quello precedente, altrimenti non verrebbe mai eseguito in quanto il parser sceglierebbe sempre la

prima alternativa. Più precisamente, un *resolver* deve essere disposto prima del punto in cui avvengono i conflitti LL(1).

2. un *resolver* può solo essere disposto davanti ad un'alternativa che è in conflitto con qualche altra alternativa, altrimenti si ha un errore.

L'**esempio** seguente mostra come i *resolvers* siano disposti in modo scorretto:

```
A = ( a  ( IF (...) b )  c        // resolver posto in una posizione errata.
                                     Non esistono conflitti LL(1)
  | IF ( ... )  a  b              // resolver non valutato
  | IF  ( ... )  b                // resolver posto in una posizione errata.
                                     Non esistono conflitti LL(1)
  ) .
```

Qui, invece, l'esempio indica la disposizione corretta dei *resolvers*:

```
A = ( IF ( ... )  a  b   // risolve il conflitto tra le due prime alternative
  | a  c
  | b
  ) .
```

6.7 Trattamento degli errori sintattici

Se viene rilevato un errore sintattico durante la fase di analisi, il parser genera un messaggio di errore e prova a recuperare l'errore di input sincronizzandolo con la grammatica. I messaggi di errore sono generati automaticamente; l'utente, inoltre, può inserire determinati suggerimenti

nella grammatica per permettere al parser di rimediare agli errori precedentemente fatti [41,42].

6.7.1 Simboli terminali non validi

Se, in una stringa di input, ci si aspettava un certo simbolo, poi non trovato, il parser, comunque, segnala che si era in attesa di quel determinato simbolo [41]. Per esempio, data la produzione

$$A = a\ b\ c\ .$$

e un determinato input del tipo:

a x c

il parser invia il seguente messaggio:

-- line ... col ...: b expected

6.7.2 Liste alternative non valide

Se il simbolo di *lookahead* non è in relazione con nessuna delle alternative della lista previste per un non terminale A, allora il parser segnala con un messaggio che A è un errore [41]. Per esempio, data la produzione

$$A = a\ (\ b \mid c \mid d\ )\ e\ .$$

e dato il seguente input del tipo

a x e

il messaggio di errore sarà:

-- line ... col ...: invalid A

Questo messaggio di errore può essere migliorato se trasformiamo la lista delle alternative con un simbolo non terminale, per esempio:

A = *a*B*e* .
B = *b* | *c* | *d* .

In questo caso il messaggio di errore sarà della forma:

-- line ... col ...: invalid B

6.7.3 Sincronizzazione

Dopo aver segnalato l'errore il parser continua finché non arriva ad un certo *punto di sincronizzazione* dove prova a sincronizzare l'input con la grammatica. I punti di sincronizzazione sono specificati dalla parola chiave SYNC. Ci sono punti nella grammatica in cui è previsto il simbolo *safe*. Quando il parser trova un punto di sincronizzazione salta tutto l'input finché non trova un token che occorre a questo punto.

In molti linguaggi di programmazione i punti di sincronizzazione sono dichiarati all'inizio del comando (dove le parole chiave sono *if*, *while* o *for*) o all'inizio della sequenza delle dichiarazioni (dove le parole chiave sono *public*, *private* o *void*). Il segno ; è inoltre un buon punto di sincronizzazione in una sequenza di dichiarazioni [41,42].

La produzione seguente, per esempio, specifica che l'inizio delle dichiarazioni e il punto e virgola dopo un assegnamento sono punti di sincronizzazione:

```
Statement =
        SYNC
        ( Designator '=' Expression SYNC ';'
        | "if" '(' Expression ')' Statement ["else" Statement]
        | "while" '(' Expression ')' Statement
        | '{' {Statement} '}'
        | ...
        ) .
```

Per evitare un numero elevato di messaggi di errore durante la sincronizzazione, un errore è segnalato soltanto se almeno due tokens sono stati riconosciuti correttamente dall'ultimo errore riscontrato. Di norma ci sono pochi punti di sincronizzazone per una grammatica e ciò rende il recupero degli errori poco costoso in Coco/R [41,42].

6.7.4 Weak tokens (tokens deboli)

Il recupero degli errori può essere ulteriormente migliorato specificando i tokens che sono deboli in un certo contesto. Un *weak token* è un simbolo che viene spesso digitato male o mancante come, per esempio, un punto e virgola al posto di una virgola in una lista di parametri.

Un *weak token* è preceduto dalla parola chiave WEAK. Il parser, quando prevede un token debole che però non è presente nella stringa di input, memorizza o il token in input che segue il *weak token* o il token previsto per ogni punto di sincronizzazione [41].

I *weak tokens* sono usati spesso come separatori di simboli che occorrono all'inizio di un'iterazione. Per esempio, date le produzioni

ParameterList = '(' Parameter { WEAK ',' Parameter } ')' .
Parameter = ["ref"|"out"] Type ident .

in cui il parser non trova i simboli , o), il parser dopo il primo parametro segnala un errore e salta i simboli di input finché non trova un token debole (un inizio accettato da *Parameter*) o un'iterazione, come ad esempio), o un qualsiasi simbolo previsto da un punto di sincronizzazione (incluso il simbolo end-of-file, EOF). L'effetto è che il parsing della lista dei parametri non terminerebbe prematuramente ma otterrebbe la possibilità di sincronizzarsi con la partenza del parametro successivo dopo un separatore digitato male (erroneamente).

Per avere un buon recupero degli errori l'utente di Coco/R dovrebbe effettuare alcuni esperimenti con gli input errati e porre le chiavi private *SYNC* e *WEAK* in modo appropriato per recuperare gli errori più probabili [41].

6.8 Struttura del file

Il parser è generato dal file con il nome **Parser.frame**. Questo file contiene parti di codice fisso e indicatori testuali che denotano le posizioni che parti specifiche della grammatica hanno all'interno di Coco/R. In rare situazioni gli utenti più esperti possono modificare le parti fisse di un pezzo di codice influenzando, fino ad un certo punto, il comportamento dello scanner e del parser [41].

6.9 Altri parser generators per grammatiche LL

CppCC [37]: mira a rimpiazzare Lex & Yacc per coloro che vogliono scrivere in C++ il loro parser e genera parser LL(k) ricorsivi e discendenti.

JavaCC (Java Compiler Compiler) [36]: è uno dei più diffusi parser generators per Java; fornisce anche funzionalità quali la costruzione di alberi (tramite lo strumento JJTree), azioni, debugging, etc. JavaCC genera parsers top-down (mediante analisi ricorsiva discendente), al contrario di Yacc e delle varianti che producono parsers bottom-up.

JetPAG (Jet Parser Auto-Generator) [38]: è un parser ricorsivo discendente ed un generatore di analisi lessicale; genera riconoscitori ottimizzati ed efficienti. I programmi generati dalla compilazione si possono facilmente integrare con progetti più grandi, pertanto questo tipo di parser è molto flessibile.

PRECC [39]: fa parte del sistema UNIX ed è progettato per estendere le possibilità di Lex e Yacc. L'utilità di Precc è che esso prende *lookahead* illimitati e backtracking nella notazione estesa BNF e grammatiche parametrizzate. Il codice è generato nel linguaggio C e, a differenza del codice generato da Yacc, è modulare e veloce con altri tipi di linguaggi.

Spirit [40]: è un parser generator ad oggetti ricorsivo e discendente; è scritto nel linguaggio C++ ed è capace di analizzare grammatiche ambigue.

APaGed: vedere paragrafo 4.7.

Capitolo 7

Conclusioni

La presente tesi si è prefissata di sottolineare ed analizzare le differenze di gestione e di analisi dei vari tipi di parsers, in particolar modo prendendo in esame i parser generators Yacc, Bison, Essence e Coco/R.

La prima parte del lavoro, alla quale si è giunti dopo un'accurata raccolta di materiale, è consistita nel descrivere in modo dettagliato tutte le definizioni utilizzate in questa tesi; di esse si è cercato di spiegare in modo semplice e chiaro il significato e l'uso ai fini dell'analisi di una determinata grammatica.

Nella seconda parte ci siamo concentrati sul parser generator Yacc, del quale si è cercato di descrivere in maniera puntuale le proprietà e il funzionamento. L'idea iniziale era quella di studiare in ogni sua parte il codice Yacc e i codici da esso generati. Purtroppo, però, Yacc si è rivelato di struttura molto complessa e di non facile analisi; abbiamo preferito, quindi, concentrarci sullo studio dettagliato del file *y.output* e in forma generica del file *y.tab.c*. Per poter lavorare sul codice si è dovuto utilizzare il sistema UNIX e per la generazione dei file è stata usata la versione 1.28 di Yacc.

Nell'ultima parte della tesi abbiamo preso in esame gli altri tre parsers sopra nominati e si è cercato di vedere qual è la struttura dati che ciascun parser utilizza per verificare se una stringa data in ingresso appartiene o meno ad una grammatica accettata da tale parser. In questa fase si è potuto notare che la struttura dati dei parsers top-down è molto più complessa dei parsers bottom-up; ciò avviene perché i parsers top-down utilizzano funzioni e trasformazioni più complesse degli altri.

Dallo studio svolto è emerso che i parser generators rappresentano un utile strumento per la generazione di codici, codici che il progettista, se lo desidera, può modificare in base alle proprie esigenze. Per avere una buona gestione del parser di cui fa uso, il progettista può, ad esempio, inserire, all'interno del file in cui è scritta la grammatica, dei messaggi adatti a valutare in maniera corretta un determinato token.

Un'altra proprietà di tutti i parsers presi in esame è quella di risolvere il problema delle grammatiche ambigue: le ambiguità presenti vengono considerate errori sintattici e risolte dai parsers applicando i rispettivi metodi di risoluzione.

Come si è visto, esistono vari tipi di parsers a seconda del linguaggio di programmazione, e ciò perché si è avuta la necessità di trovare dei generatori di parsers a seconda del tipo di linguaggio adottato dall'utente. Questi, per interagire con tali generatori, usa un'interfaccia grafica che lo porta ad ottenere, in modo del tutto efficiente e corretto, un ottimo risultato senza scrivere o modificare il codice del parser medesimo.

Il presente lavoro ha preso in considerazione un parser generator per ogni tipo di grammatica; sarebbe interessante, in futuro, studiare i parsers che analizzano la stessa grammatica, per mettere in evidenza la loro struttura dati e il modo in cui i parsers medesimi analizzano e risolvono i vari errori sintattici.

Bibliografia

[1] Aho, Sethi, Ullman, “Compilers / Principles, Techniques, and Tools”, Addison-Wesley publishing company 1988

[2] Dino Mandrioli, “Introduzione al progetto di compilatori”, Franco Angeli 1985

[3] M. Aiello, A. Albano, G. Attardi, and U. Montanari, “Teoria della computabilità, logica, teoria dei linguaggi formali”, E.T.S., Pisa 1976

[4] Crespi Reghizzi, Della Vigna, Grezzi, “Linguaggi formali e compilatori”, ISEDI 1978

[5] Crespi Reghizzi, “Sintassi semantica e tecniche di compilazione”, Clup – Milano 1986

[6] Crespi Reghizzi, “Compilatori interpreti tecniche di traduzione”, Masson 1990

[7] P.D. Terry, Rhodes university, 1996, “Compilers and compiler generators”, International Thomson Computer Press 1997

[8] William M. Waite, Gerhard Goos, “Compiler construction”, 1995

[9] Keith Cooper, Linda Torczon, “Engineering a compiler”, Elsevier 2003

[10] O.G. Kakde, “Algorithms for compiler design”, Charles River Media 2002

[11] David L. Dill, “Introduction and lexical analysis”, Spring 1997

[12] G.Ausiello, F. D’Amore, G. Gambosi, “Linguaggi, Modelli, Complessità”, E.T.S., 2001

[13] G. Callegarin, L. Varagnolo, “Corso di informatica generale / Strutture dati linguaggi formali e compilatori vol. 2”, CEDAM 1984

[14] Robin Hunter, "Compilers / Their design and construction using pascal", John Wiley & Sons Ltd 1985

[15] A.J. Kfoury, M. A. Arbib, R.N. Moll, "An introduction to formal language theory", Springer 1988

[16] S. Sippu, E. Soisalon-Soininen, "Parsing theory Vol. II", Springer 1990

[17] Stephen C. Johnson, "Yacc: Yet Another Compiler-Compiler", Computing Science Technical Report No. 32, Bell Laboratories, Murray hill, New Jersey 1975

[18] Thomas Niemman, "A compact guide to: lex & yacc", ePaper Press

[19] George Hansper, 2000
http://www.luv.asn.au/overheads/lex_yacc/yacc.html

[20] http://www.opengroup.org/onlinepubs/009695399/utilities/yacc.html

[21] Charles Donnelly & Richard Stallman, "Bison – The Yacc-compatible Parser Generator", Free Software Foundation, version 2.3, 30 maggio 2006

[22] Jascha Wetzel, "APeGeD v0.3 Documentation", 24 agosto 2007

[23] http://apaged.mainia.de/

[24] http://beaver.sourceforge.net/index.html

[25] Robert Corbett, http://invisible-island.net/byacc/byacc.html

[26] Tomas Hurka, http://byaccj.sourceforge.net/

[27] M. K. Crowe, "Compiler Writing tools using C#", version 4.7i, luglio 2007; http://cis.paisley.ac.uk/crow-ci0/#Research

[27] Scott E. Hudson, "CUP: LALR parser generator for java", Graphics Visualization and Usability Center, Georgia Institute of Technology; http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html

[29] http://edenti.deis.unibo.it/Ling/2006-2007/Strumenti.html

[30] Simon Marlow, "Happy User Guide", Andy Gill, 1997; http://www.haskell.org/happy/

[31] Mark P. Jones, "Jacc: just another compiler compiler for java – A reference Manual and User Guide", Department of Computer Science & Engineering, OGI School of Science & Engineering at OHSU, 20000 NW Walker Road, Beaverton, OR 97006, USA, 16 febbraio 2004; http://web.cecs.pdx.edu/~mpj/jacc/index.html

[32] Ginger G. Wyrick, D. Richard Hipp http://www.hwaci.com/sw/lemon/

[33] http://sourceforge.net/projects/lpg/

[34] "Elkhound: A Fast, Practical GLR Parser Generator", Scott McPeak, George C. Necula, In Proceedings of Conference on Compiler Construct (CC04), aprile 2004
http://www.cs.berkeley.edu/~smcpeak/elkhound/

[35] François Pottier & Yann Régis-Gianas, "Menhir Reference Manual", INRIA, 20 maggio 2007, http://cristal.inria.fr/~fpottier/menhir/

[35] Mike Sperber, Peter Thiemann, "Essence - An LR Parser Generator for Scheme, version 1.0", 26 marzo 1999

[36] https://javacc.dev.java.net/

[37] Alec Panovici, "CppCC user's guide", 1 febbraio 2003

[38] http://jetpag.sourceforge.net/

[39] Peter T. Breuer & Jonathan P. Bowen, "The PRECC Compiler Compiler", Oxford University Computing Laboratory

[40] Joel de Guzman, http://www.boost.org/libs/spirit/ , settembre 2002

[41] Hanspeter Mössenböck, "The Compiler Generator Coco/R – User Manual", Johannes Kepler University Linz, Institute of System Software, 19 settembre 2006, http://ssw.jku.at/Coco/

[42] Albrecht Wöß, Markus Löberbauer, Hanspeter Mössenböck, "LL(1) Conflict Resolution in a Recursive Descent Compiler Generator", Johannes Kepler University Linz, Institute of Practical Computer Science

[43] Hanspeter Mössenböck, “Data Structures in Coco/R”, Johannes Kepler University Linz, Institute of System Software, aprile 2005