UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in Informatica

Dispensa di Linguaggi Formali e Compilatori

Dal materiale della professoressa Paola Quaglia

Squadra

Filippo Daniotti
Samuele Conti
Francesco Bozzo
Michele Yin
Giacomo Zanolli

Squadra

Federico Izzo
Simone Alghisi
Emanuele Beozzo
Samuele Bortolotti

Anno accademico 2020-2021

Indice

Prefazione

Il progetto

Questo testo è una dispensa di appunti scritta da studenti; lo scopo è quello di raccogliere i contenuti del corso di Linugaggi Formali e Compilatori e organizzarli secondo un'esposizione quanto più completa, efficace ed intuitiva possibile, tanto per lo studente desideroso di ottenere un'ottima padronanza degli argomenti, quanto anche per lo studente pigro in cerca di risorse per "portare a casa" l'esame.

Gli appunti sono stati presi durante il corso di Linguaggi Formali e Compilatori tenuto dalla professoressa Paola Quaglia per il Corso di Laurea in Informatica, DISI, Università degli studi di Trento, anno accademico 2020-2021. I contenuti provengono quindi primariamente dalle lezioni della professoressa, mentre invece ordine ed esposizione sono in gran parte originali. Allo stesso modo, la maggior parte degli assets (figure, grafi, tabelle, pseudocodici) sono contenutisticamente tratti dal materiale della professoressa, ma ricreati e molto spesso manipolati dagli autori; inoltre, per ottenere il risultato appena citato, è stato molto spesso necessario abbandonare quasi del tutto l'esposizione della professoressa e usarla, appunto, come canovaccio per svilupparne una originale.

Istruzioni per l'uso

L'elaborato finale è molto lungo (circa 300 pagine); questo potrebbe allontanare alcune categorie di utenti, ma ci sentiamo di sottolineare che una tale lunghezza è supportata da numerose ragioni. La più importante è che la narrazione alterna teoria ed esercizi, in quanto questi ultimi sono spesso fondamentali a un'efficace assimilazione degli stessi concetti teorici su cui poggiano le basi. Inoltre, i passaggi di questi esercizi, nonostante molto meccanici, sono descritti nei dettagli, tanto da risultare a volte quasi ridondanti; questo potrebbe risultare fastidioso agli utenti più *sgai*[1], per cui consigliamo di valutare di volta in volta se studiare nel dettaglio

[1] svegli

l'ennesimo esercizio su uno stesso algoritmo, ma nondimeno raccomandiamo di non saltare nessuno di questi a piè pari.

Preme sottolineare che la squadra non si assume alcuna responsabilità rispetto all'esito dell'esame di chi sceglie di usare questo testo come principale risorsa di studio. Nonostante noi abbiamo versato sangue per confezionare un prodotto completo, corretto e rifinito nei dettagli, non possiamo dimenticare di essere degli studenti, proni a commettere errori o malinterpretare dei passaggi come chiunque altro, e purtroppo per mancanza di tempo e mezzi non siamo riusciti a completare le 500 revisioni che ci eravamo proposti di fare. Per questo motivo dobbiamo essere intellettualmente onesti e consigliare ai lettori di fare sempre riferimento a quanto detto in classe dalla professoressa e al materiale da lei fornito e/o indicato. Per chi invece riponesse tanta fiducia in noi ad utilizzare questo testo come unica risorsa di studio, occhi aperti.

Segnalazione errori

Se durante la lettura doveste incorrere in errori di qualsiasi tipo, tra gli altri errori di battitura, errori concettuali o di impaginazione, vi chiediamo di fare una segnalazione; ve ne saremo riconoscenti e provvederemo a correggere quanto prima. Per fare una segnalazione potete scegliere uno dei seguenti modi:

- visitando la repository del progetto (la trovate qui: `https://github.com/filippodaniotti/Appunti-LFC`) potete trovare i nostri profili Github con relativi indirizzi email, scriveteci senza particolari pensieri;
- se avete le mani in pasta con LaTeX potete aprire una Github issue in cui segnalate l'errore, o anche clonare la stessa repository e proporre un vostro fix con una pull request;
- potete contattarci tramite l'indirizzo email istituzionale, che rispetta sempre questo schema: *nome.cognome@studenti.unitn.it*;
- se ci conoscete personalmente, non esitate a contattarci via canali informali come Telegram o WhatsApp;
- se ci conoscete e siete a Trento potete trovare alcuni di noi in biblioteca circa ogni giorno, da oggi fino alla fine dei tempi (o della sessione).

La squadra

Questo progetto è frutto della collaborazione di svariate personalità, ciascuna a suo modo fondamentale per il mantenimento e la riuscita dello stesso.

p!ps Filippo Daniotti: curatore del progetto, ha curato la revisione stilistica, la creazione e il mantenimento di tabelle e pseudocodici, e ha scritto le lezioni sempre con gli appunti degli altri.

sam4retas Samuele Conti: iniziatore e curatore del progetto, ha curato la scrittura e l'esposizione degli esercizi e la revisione e correttezza concettuale di tutto il progetto.

frab0zzo Francesco Bozzo: tecnico, ha curato il coordinamento tecnico la scelta di pacchetti per la creazione degli assets, ha implementato la CI e fornito consulenza in merito ai problemi tecnici di latex.

f1zzo Federico Izzo: tecnico, ha curato la creazione e il mantenimento di grafi e alberi.

t0k3n$ Simone Alghisi, Emanuele Beozzo, Samuele Bortolotti: scrittori, hanno curato la rielaborazione e la trascrizione delle lezioni, fornendo le indicazioni sulla scelta degli assets.

ch!n4 Michele Yin: revisore, l'asia, la precisione chirurgica, si è occupato di fare una revisione ad ampio respiro sui dettagli matematici e di scrivere alcune lezioni all'occorrenza, principalmente.

j4bb Giacomo Zanolli: revisore, si è occupato di una revisione di ampio respiro e di impostare il setup docker.

Nota del curatore

Dai è stato divertente adesso vi racconto una barzelletta allora ci sono tre amici

Parte I

Teoria dei linguaggi formali e degli automi a stati finiti

Capitolo 1

Introduzione

1.1 Il compilatore

1.1.1 Cos'è un compilatore

Il compilatore, generalizzando, è un meccanismo che trasforma il codice sorgente in codice eseguibile. Una caratteristica del codice eseguibile è l'essere machine dependent, difatti noi studieremo i principi generali di questa traduzione ma dobbiamo tenere bene a mente il fatto che ogni tipo di macchina ha il suo proprio linguaggio macchina che deve essere utilizzato per scrivere gli eseguibili da essa utilizzabili. Durante il corso, useremo spesso come punto di riferimento il compilatore gcc: tale progetto è in sviluppo da 20 anni e consta di ben più di due milioni di righe di codice!

A cosa serve L'esempio classico di utilizzo di un compilatore è la traduzione da codice `c` in assembly. Un codice in assembly, tuttavia non è ancora eseguibile. Manca il passaggio di conversione da assembly a codice binario; tuttavia, questo passaggio non è così complesso dato che assembly ha una corrispondenza quasi 1:1 con il binario. Ma quest'ultima traduzione non è l'unico passaggio mancante: rimane ancora da affrontare la fase di linking.

1.1.2 Le fasi del processo di compilazione

In questa sezione presentiamo le fasi del processo di compilazione come se fossero in pipeline, ma sappi che per ragioni di efficienza vengono spesso sovrapposte. Due punti fondamentali di questo processo sono i seguenti:

1. Se scriviamo il programma in un certo linguaggio, lo dovremo compilare con il compilatore dedicato esattamente a quel linguaggio

2. Quando scrivo codice per un certo compilatore devo rispettare la grammatica del linguaggio per cui il compilatore lavora

La grammatica di un linguaggio definisce la struttura legale delle varie operazioni possibili in quel linguaggio; ad esempio, può definire questa forma per un'operazione di assegnamento:

```
Identifier Assign Expression Semicolon
```

Dove `Identifier` è un segnaposto per un qualsiasi identificatore (stringa) che può essere utilizzato dal programmatore. In primis dunque abbiamo questi due elementi: codice sorgente e grammatica.

Avendo chiarito il punto di partenza, possiamo procedere con una descrizione delle fasi della compilazione.

Analisi lessicale

Il primo passaggio che compie il compilatore è l'analisi lessicale: questa traduce un flusso di caratteri in un flusso di *tokens*, ciò significa che risolve ogni "stringa" riconoscendo il suo ruolo. Per esempio, traduce:

```
pippo = 2*3
```

In una nuova stringa, che ha una forma simile a:

```
<ID, pippo> ASS <NUM,2 > MUL <NUM,3> SEMCOL
```

In questa fase i token che otteniamo contengono le informazioni sul tipo di dato o comando che ogni singola stringa identifica, "si tratta di capire chi è chi". Nota che non vengono perse informazioni, non si tratta di una traduzione in cui la stringa dell'ID viene eliminata: si tiene la stringa "pippo" e vi si assegna il ruolo di identificatore (ID).

Analisi sintattica

Dopo l'analisi lessicale arriva l'analisi sintattica: questa analisi verifica che la sequenza di token sia aderente alla grammatica del linguaggio che stiamo utilizzando, ed è la fase in cui compaiono i temuti syntax error.

Se rispettiamo le regole della grammatica il flusso di token viene tradotto in un parse tree (o meglio ancora, dipendentemente dal compilatore, in un albero di sintassi astratta). Ecco il parse tree che si può ricavare dal nostro esempio:

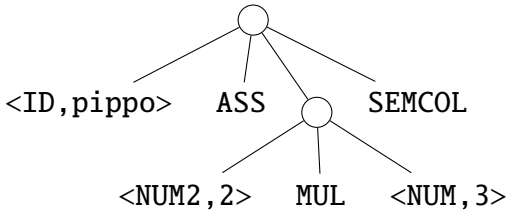


Figura 1.1: Esempio di parse tree

La struttura del parse tree è derivata dalla grammatica del linguaggio. Ad esempio, `<NUM,2> MUL <NUM,3>` sono sottoalberi di `ASS` poiché la grammatica del linguaggio prevede che l'assegnazione abbia questa forma.

Più la struttura del parse tree è minimale più le fasi successive sono efficienti; qui che entra in gioco l'AST, che semplifica ancora lo schema: l'albero di sintassi astratta (abstract syntax tree, AST) è ottenuto dal parse tree compattando delle parti che non saranno utili nelle fasi seguenti della compilazione.

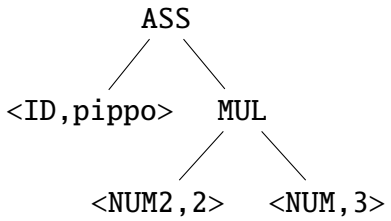


Figura 1.2: Esempio di abstract syntax tree

Il guadagno in complessità in questo caso è poco (abbiamo risparmiato tipo 3 nodi), ma in casi più complessi il risparmio è molto più rilevante.

Non è facile scrivere una grammatica che possa essere analizzata con efficienza da un compilatore e che, al tempo stesso, generi parse tree più semplici possibili; ma di questo discuteremo in futuro. È da notare che esistono svariate forme di compilatori, noi ne stiamo vedendo una presentazione generale, ma, in molti casi, fasi come creazione di parse tree e AST vengono spesso sovrapposte.

L'output dell'analisi sintattica è chiamato *tree*, ma non necessariamente è un albero: spesso può essere un grafo. Ad esempio in un'operazione del tipo: `pippo = pippo * 2` il nodo pippo sarebbe collegato sia alla radice che all'operatore `ASS`, creando quindi un grafo.

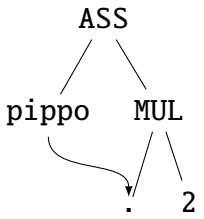


Figura 1.3: Abstract syntax tree che diventa un grafo quando la variabile a cui si assegna il valore fa parte dell'espressione alla destra dell'assegnamento.

Analisi semantica

Ora è la fase dell'analisi semantica. Un esempio di operazione di analisi semantica è quello di capire se stiamo utilizzando il giusto operatore di moltiplicazione per l'operazione che vogliamo effettuare. Stiamo chiaramente riferendoci a quei linguaggi in cui esistono diversi significati per uno stesso operatore, come nel caso in cui l'operatore `MUL` può significare moltiplicazione sia tra interi sia tra float. Quindi si può dire che in questo caso l'analisi semantica chiarisce quale significato ha l'operatore `MUL`: nell'esempio riportato in 1.2 `MUL` = moltiplicazione tra due variabili di tipo `NUM`.

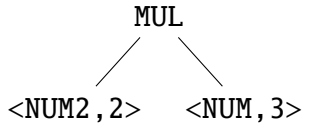


Figura 1.4: La semantica di `MUL` deve essere specificata

Generazione del codice intermedio

È il momento della generazione del codice intermedio: in questa fase viene creato un codice testuale creato traducendo il parse tree. Queste istruzioni sono leggibili dall'uomo ma non è né assembly né codice effettivo, d'altronde si chiama intermedio, no? Nell'esempio in 1.5 abbiamo sulla sinistra il parse tree e sulla destra abbiamo un esempio del codice intermedio generato.

Figura 1.5: Generazione del codice intermedio

Generazione del target code

Questa fase consiste nella traduzione del codice intermedio nell'assembly della specifica macchina. Il codice binario non è necessariamente il target code della compilazione: ad esempio, potremmo essere su una VM e quindi generare una sorta di bytecode, o magari potremmo avere come obiettivo quello di tradurre in c per poi usare il gcc che sappiamo essere super-efficiente.

1.2 Riassumendo le fasi della compilazione

Riassumendo i passaggi necessari per la compilazione sono:

1. Analisi lessicale
2. Analisi sintattica (generazione di AST)
3. Analisi semantica
4. Generazione codice intermedio
5. Generazione target code

Possiamo dividere questi passaggi in due gruppi:

- *Front-end* Dall'analisi lessicale fino al codice intermedio.
- *Back-end* Tutto il resto.

Ma perché passiamo dalla fase del codice intermedio? Per ragioni di modularità: Se abbiamo N linguaggi, abbiamo N front-end e se abbiamo K macchine, abbiamo K back-end ciò implica che senza un codice intermedio dovremo creare $N * K$ differenti compilatori, usiamo il codice intermedio come livello di astrazione per semplificare il tutto.

Capitolo 2

Le grammatiche generative

2.1 Definizione di grammatica generativa

Le grammatiche che tratteremo in questo corso sono chiamate grammatiche generative, così dette perché vengono usate per generare linguaggi.

Il vocabolario Il primo passo per definire una grammatica è definirne il vocabolario, questo consiste in un set di simboli. Alcuni di questi simboli sono chiamati terminali (terminals) e giocano il ruolo di token nell'analisi lessicale. Un simbolo particolare nella grammatica è lo start symbol: come suggerisce il nome, questo è il simbolo che permette di iniziare a generare il linguaggio derivato dalla grammatica (tale processo sarà chiarito in seguito).

Le produzioni Altra cosa da fissare in una grammatica è il set delle produzioni (productions); queste sono regole di riscrittura da stringhe ad altre stringhe e rispettano la seguente limitazione: la stringa di partenza deve contenere almeno un simbolo non-terminale. Un esempio di regole per una grammatica è il seguente:

$$\{S \to aSb, S \to ab\} \tag{2.1}$$

Dove S è lo starting point mentre a e b sono caratteri terminali (a volte detti anche parole).
Per quanto abbiamo visto fin ora una grammatica generativa è composta da:

- Un vocabolario, ovvero l'insieme dei simboli
- Alcuni simboli speciali, ovvero i terminali, i non terminali ed il simbolo di partenza

- Le produzioni, ovvero regole che permettono di trasformare stringhe contenenti non terminali in qualcos'altro

Questi sono gli ingredienti di una grammatica generativa.

Il linguaggio Il linguaggio generato da una grammatica è l'insieme di stringhe composte solo da terminali che può essere generato partendo dallo start symbol di quella stessa grammatica.

2.1.1 Derivare il linguaggio di una grammatica generativa

Per derivare un linguaggio dalla grammatica generativa dobbiamo applicare, **quante volte possiamo**, tutte le regole di riscrittura che appartengono alla grammatica stessa.

Le derivazioni Ogni riscrittura è chiamata derivation step (operazione denotata dal simbolo $\implies$, diverso dal simbolo usato nelle produzioni $\rightarrow$). L'operazione di derivazione serve per tradurre stringhe con caratteri non terminali in stringhe composte **solo** da caratteri terminali (seguendo le regole dettate dalle produzioni).

Un esempio pratico Partendo dalla grammatica espressa in 2.1 possiamo sviluppare la seguente derivazione:

$$S \implies ab \tag{2.2}$$

Questa è una derivazione da S con un singolo step; la parola ab appartiene al linguaggio della nostra grammatica d'esempio. Un'altra derivazione possibile è la seguente:

$$S \implies aSb \implies aaSbb \tag{2.3}$$

Diversamente da prima, $aaSbb$ non è una parola del linguaggio perché non è composta *solo* da caratteri terminali. In questo modo si può definire qual è il linguaggio determinato da una grammatica, ad esempio con la grammatica che abbiamo visto prima possiamo generare ogni sequenza di parole formate da n terminali a seguiti da n terminali b. Tale linguaggio può essere scritto formalmente in questo modo:

$$\{a^n b^n \mid n > 0\} \tag{2.4}$$

L'utilità del meccanismo di derivazione del linguaggio può esserci più chiara se pensiamo all'esempio appena visto come un metodo per verificare che tutte le parentesi aperte vengano chiuse, dove il terminale a sta per parentesi aperta ed il terminale b sta per parentesi chiusa. Una grammatica come 2.1 tramite il mezzo delle produzioni ci permette di eseguire questo tipo di controlli.

Regole di notazione Presentiamo ora alcune regole di notazione per lavorare con il processo di derivazione delle grammatiche generative.

- i simboli del vocabolario prescelto che non fungono da terminali verranno sempre indicati con lettere maiuscole
- il carattere ε denota la parola vuota
- la lunghezza di ε è zero
- $\varepsilon = \varepsilon\varepsilon$
- $\varepsilon = b^0$ dove b è un qualunque carattere terminale
- il carattere ε non viene scritto nelle parole

2.2 Esercizi di derivazione da una grammatica

Presentiamo ora una serie di eserciziett di derivazione del linguaggio da una certa grammatica, con breve spiegazione del processo.

Grammatica	Linguaggio derivato
$S \to aAb$ $aA \to aaAb$ $A \to \varepsilon$	$\{a^n b^n \mid n > 0\}$

Tabella 2.1: Esercizio 1

Spiegazione di Tab.2.1:

Ricordiamo che si parte sempre e solo dallo start symbol a derivare un linguaggio, quindi da S. L'unica derivazione che si può effettuare in questo caso è $S \implies aAb$. Da qui in poi non si può procedere che con una delle due productions $aA \to aaAb$ oppure $A \to \varepsilon$, in un caso si aumenterà la stringa di un a a sinistra e di un b a destra, nell'altro caso si terminerà la derivazione.

Diventa quindi semplice vedere che il linguaggio di questa grammatica è lo stesso generato da 2.1, ovvero: $\{a^n b^n \mid n > 0\}$.

Grammatica	Linguaggio derivato
$S \to AB$ $A \to aA$ $A \to a$ $B \to Bb$ $B \to b$	$\{a^n b^m \mid n, m \geq 1\}$

Tabella 2.2: Esercizio 2

Spiegazione di Tab.2.2:

Come prima si parte dal simbolo S che anche questa volta ci permette una sola produzione, quindi deriviamo senza ulteriori indugi: $S \implies AB$. In seguito ci rendiamo conto che le productions dell'esercizio altro non sono che delle regole per generare un numero a nostro piacimento di terminali a e b, quindi deriviamo tranquillamente il seguente linguaggio: $\{a^n b^m \mid n, m \geq 1\}$.

Grammatica	Linguaggio derivato
$S \to aSBc$ $S \to abc$ $cB \to Bc$ $bB \to bb$	$\{a^n b^n c^n \mid n > 0\}$

Tabella 2.3: Esercizio 3

Spiegazione di Tab.2.3:

In questo caso il simbolo di partenza S ci offre due possibili productions, quindi dobbiamo sperimentarle entrambe per generare il linguaggio completo.

- la seconda produzione ci porta subito ad una derivazione terminale $S \implies abc$;
- la prima produzione invece $S \to aSBc$ ci permette di ricorrere in un processo ricorsivo che continua ad aggiungere terminali a sulla sinistra, la coppia Bc sulla destra e termina solo quando decidiamo di sostituire S con abc; Questo implica che nella situazione terminale ci potremo trovare in una situazione in cui abbiamo una stringa con questa forma $aaa...abcBcBcBc...Bc$.

Da questa forma per eliminare tutti i non terminali B non abbiamo altra soluzione se non quella di utilizzare la produzione $cB \to Bc$ per spostare tutte le c in fondo.

Alla prima sostituzione otteniamo: $aaa...abBcBc...Bc...BcBcc$. Riutilizzando la stessa regola possiamo spostare tutte le B da destra verso il centro, ottenendo: $aaa...abBB...Bcc...ccc$ ed in seguito con $bB \rightarrow bb$ trasformare tutte le B in b. Il linguaggio che ricaviamo alla fine è $\{a^n b^n c^n \mid n > 0\}$.

Grammatica	Linguaggio derivato
$S \rightarrow AB$ $A \rightarrow a$	$\emptyset$

Tabella 2.4: Esercizio 4

Spiegazione di Tab.2.4:

La grammatica specificata nell'esercizio non permette di generare stringhe contenenti solo caratteri terminali, di conseguenza non genera alcuna parola valida. Essendo il linguaggio definito come un insieme di parole che la grammatica può creare, per male che vada tale insieme è vuoto, ma esiste sempre. Questo è proprio il nostro caso, il linguaggio di questa grammatica è l'insieme vuoto.

Grammatica	Linguaggio derivato
$S \rightarrow \varepsilon$	$\{\varepsilon\}$

Tabella 2.5: Esercizio 5

Spiegazione di Tab.2.5:

Non c'è molto da dire in questo caso se non che l'insieme contenente la parola vuota è diverso dall'insieme vuoto $\{\varepsilon\} \neq \emptyset$.

Grammatica	Linguaggio derivato
$S \rightarrow aSb$ $S \rightarrow \varepsilon$	$\{a^n b^n \mid n > 0\} \cup \{\varepsilon\} = \{a^n b^n \mid n \geq 0\}$

Tabella 2.6: Esercizio 6

Spiegazione di Tab.2.6:

La spiegazione è lasciata al lettore.

Grammatica	Linguaggio derivato
$S \to CD$	$\{ww \mid w \in \{a^*b^*\}^*\}$
$C \to aCA \vert bCB$	
$AD \to aD$	
$BD \to bD$	
$Aa \to aA$	
$Ab \to bA$	
$Ba \to aB$	
$Bb \to bB$	
$C \to \varepsilon$	
$D \to \varepsilon$	

Tabella 2.7: Esercizio 7

Spiegazione di Tab.2.7:

- Prima di tutto partiamo da S, l'unica produzione possibile ci porta a CD; ora possiamo dimenticarci di S;
- vediamo subito che sia C che D possono essere trasformate in ε a piacimento, quindi possiamo interrompere le derivazioni da loro in ogni momento;
- ora, C ci permette di utilizzare le due produzioni $C \to aCA$ e $C \to bCB$ (il simbolo | ci dice che da C posso produrre aCA *or* bCB);
- notiamo che se siamo in una forma del tipo $aCAD$ possiamo proseguire con le produzioni di C oppure possiamo utilizzare $AD \to aD$ ed osservando bene notiamo che questo è l'unico modo in cui potremo eliminare il non terminale A, il che è speculare per quanto riguarda il caso $bCBD$;
- notiamo inoltre che per ogni nuovo non terminale che aggiungiamo sulla destra dovremo farlo scorrere fino al limite estremo destro della parola (dove si trova D) per trasformarlo in terminale; a questo aggiungiamo che per ogni a sulla sinistra aggiungo A sulla destra ed in egual modo funziona per b;
- ora dovrebbe essere chiaro che, al termine delle derivazioni, il numero di a sulla sinistra sarà uguale al numero di a sulla destra e lo stesso per b; inoltre al non terminale appena a sinistra dell'ultimo C corrisponderà un terminale uguale in posizione estrema destra (appena prima di D).

La soluzione è quindi $\{ww \mid w \in \{a^*b^*\}^*\}$.

2.3 Grammatiche Generative, più formalmente

Una grammatica generativa è una tupla con questa forma $\mathcal{G} = (V, T, S, \mathcal{P})$:

- V è il vocabolario (simboli terminali e non, completo)
- T è il set dei simboli terminali
- S è lo start symbol
- P è il set delle produzioni

Una volta definiti i componenti di una grammatica generativa, definiamo le convenzioni notazionali (o notazioni convenzionali) che si usano in questo campo:

- Lettere maiuscole all'inizio dell'alfabeto (A, $B\ldots$) significano simbolo non terminale ($\in (V \setminus T)$).
- Lettere maiuscole alla fine dell'alfabeto (X, Y, $Z\ldots$) un generico simbolo del vocabolario, terminale o meno, non si sa ($\in V$).
- Lettere minuscole inizio alfabeto (a, $b\ldots$) per indicare caratteri terminali ($\in T$).
- Lettere minuscole greche per indicare stringhe che possono essere composte da terminali o non terminali $\in V^*$, dove * significa che possono esserci zero o più ripetizioni di elementi nella base.
- Lettere minuscole in fondo all'alfabeto con pedice numerico (w, $w_0\ldots$) per indicare stringhe di caratteri terminali, dette anche $words$ o $parole$ del linguaggio.

La produzione Definiamo ora formalmente l'oggetto produzione, esso è una traduzione in questa forma:

$$\delta \to \beta \tag{2.5}$$

Dove $\delta \in V^+$, δ contiene almeno un non terminale (non traduciamo terminali in altri terminali); il $^+$ significa che c'è una o più ripetizioni di elementi nella base, che quindi non è vuota (ε).
δ è detto **driver** della produzione mentre β è detto **body** della produzione.

Il linguaggio Il linguaggio generato da una grammatica $\mathcal{G} = (V, T, S, \mathcal{P})$ è così definito:

$$\mathcal{L}(\mathcal{G}) = \{w \mid w \in T^* and\ S \implies^* w\} \tag{2.6}$$

Quell simbolo $\implies^*$ è il simbolo utilizzato per indicare una sequenza di derivazioni successive compatibili secondo le produzioni (anche nessuna); in altre parole, il linguaggio generato è l'insieme di tutti gli elementi (anche vuoti) che si possono derivare con uno o più passi dallo start symbol.

2.3.1 Tipi di grammatiche generative

Esiste una gerarchia delle grammatiche, che non dipende dal genere di analisi che permettono di fare, ma dipende solo dalla struttura delle loro produzioni.

Le grammatiche libere Si parla di *context-free grammars*, o free grammars (grammatiche libere), quando tutte le produzioni sono in forma $A \to \beta$, ovvero che hanno driver composti *solamente* da non terminali. Queste grammatiche sono molto importanti perché praticamente tutti i linguaggi di programmazione sono derivati da questo tipo di grammatica.

Si dice di un linguaggio che è context-free se esiste una grammatica libera che genera quello stesso linguaggio: $\mathcal{L}$ è un linguaggio libero se e solo se esiste una grammatica libera $\mathcal{G}$ tale che $\mathcal{L} = \mathcal{L}(\mathcal{G})$.

L'operazione di analizzare una stringa e verificare se appartiene ad una determinata grammatica dipende naturalmente dal metodo con cui si decide di applicare la derivazione dallo starting point della grammatica stessa. Se una stringa si può derivare da una certa grammatica, si dice che la prima appartiene al linguaggio di quest'ultima. Esistono due tipi di derivazione canonica:

- Rightmost (si rimpiazzano sempre prima i non terminali più a destra)
- Leftmost (si rimpiazzano sempre prima i non terminali più a sinistra)

Se utilizziamo come strategia una di queste due tutti i passi analitici di una derivazione sono deterministici, perché ad ogni passo o si procede con l'elemento più a destra o si procede con quello più a sinistra. Per questo motivo queste sono le due modalità di derivazione canonica.

Gli alberi di derivazione Le grammatiche libere si prestano in modo naturale ad essere processate tramite il cosiddetto albero di derivazione. Questo albero ha come radice lo starting point, poi, per ogni passo di derivazione secondo una

produzione in forma $A \to X_1 \mid X_2 \mid \cdots \mid X_n$, si generano n figli del nodo con la derivazione di A messa in atto. Le foglie dell'albero saranno composte da ε oppure da caratteri terminali. In figura 2.1 è riportato come esempio l'albero di derivazione della grammatica:

$$S \to aSb \mid \varepsilon \tag{2.7}$$

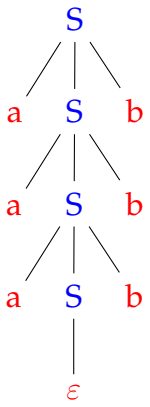


Figura 2.1: Albero di derivazione di 2.7

Ma tornando con i piedi per terra, cosa centra tutto questo con i compilatori? Un certo programma che è stato trasformato in una stringa viene verificato controllando che la tale stringa sia derivata dalla grammatica del linguaggio in cui il programma è scritto. Un modo per verificare questo è trovare l'albero di derivazione della stringa che rappresenta tale programma.

2.3.2 Ambiguità delle grammatiche

Una grammatica $\mathcal{G}$ si dice ambigua se esiste $w \in \mathcal{L}(\mathcal{G})$ che può essere derivata da due derivazioni canoniche distinte, entrambe rightmost o entrambe leftmost. Distinte significa che l'ordine di applicazione delle produzioni è differente, ma questo sarà più chiaro in seguito. Noi non vogliamo grammatiche ambigue, ci complicano la vita.

Esempio: le espressioni aritmetiche Prendiamo ad esempio la grammatica delle espressioni aritmetiche:

$$E \to E + E \mid E * E \mid n \tag{2.8}$$

dove il simbolo | è una modo compatto per scrivere diverse produzioni con lo stesso driver. La grammatica proposta è equivalente alla seguente:

$$E \to E + E$$
$$E \to E * E$$
$$E \to n$$

Intuitivamente si può leggere come $exp \to exp + exp$ o $exp * exp$ o $numero$. Questa grammatica ci permette di fare operazioni di somma o moltiplicazione, è ambigua o meno? Per verificarlo proviamo a derivare questo $w = n + n * n$ utilizzando la tecnica leftmost e generando due derivazioni distinte.

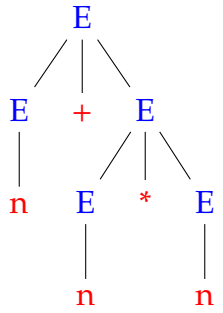


Figura 2.2: Derivazione leftmost 1

- Nel primo caso di derivazione leftmost (2.2) partiamo con il trasformare E in $E + E$;
- di seguito, proseguendo in leftmost trasformiamo la prima E in n, poi la seconda E in $E * E$;
- infine, trasformiamo (sempre da sinistra a destra) entrambe le E rimanenti in n.

Semplice, no? Ora proviamo ad ottenere la stessa parola in modo alternativo:

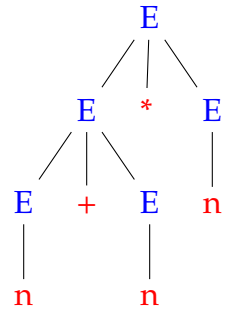


Figura 2.3: Derivazione leftmost 2

Si nota subito che la differenza sostanziale è che in questo secondo caso (2.3) la prima produzione che utilizziamo non è $E \to E + E$, bensì $E \to E * E$. Questo comporta due derivazioni differenti anche se entrambe sono nella stessa forma canonica leftmost. La grammatica di partenza è quindi ambigua: nonostante il significato (l'interpretazione matematica dei due alberi) sia diverso riusciamo a costruire w con due alberi differenti che usano la stessa strategia canonica di derivazione.

Ora abbiamo visto questo processo, sappi che la prof.ssa Quaglia preferisce 14 a 1 la derivazione con albero a quella con stringa perché è più chiara.

Dangling Else

Introduciamo una nuova forma di grammatica

$$S \to \ if\ b\ then\ S \mid if\ B\ then\ S\ else\ S \mid other \tag{2.9}$$

La domanda che ci poniamo a questo punto è: si tratta di una grammatica ambigua? Prendiamo ad esempio questa parola:

$$w = \ if\ b\ then\ if\ b\ then\ other\ else\ other \tag{2.10}$$

Con quale $then$ è accoppiato l'$else$? Non si può sapere! Di fatto possiamo ricostruire w con questi due diversi alberi:

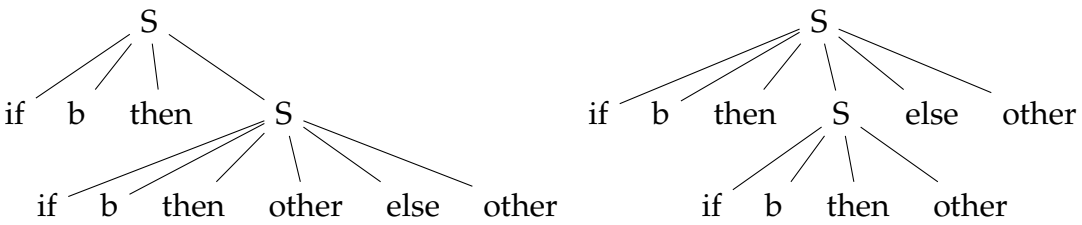


Figura 2.4: Due possibili alberi di derivazione per w, entrambi corretti

Questo ci introduce il problema di come trattare la gestione del comando $if\ then\ else$ in un linguaggio di programmazione. Per questo nei vari linguaggi troviamo sempre qualche forma di strategia per risolvere questo problema:

- Mettere le parentesi
- Eliminare l'else e usare solo if
- Usare la regola del closest unmatched then (l'$else$ è sempre valutato come appartenente all'ultimo $then$ non ancora elsato)

L'else è intrinsecamente ambiguo.

Un'osservazione L'ambiguità in alcuni casi può essere indecidibile, in quanto non sempre esiste un algoritmo che ci dice se la grammatica è ambigua o no.

2.3.3 Piccola parentesi sui linguaggi naturali

I linguaggi naturali spesso nascondono molte più ambiguità delle grammatiche che affronteremo in questo corso. D'altro canto le grammatiche per linguaggi di programmazione sono molto difficili da scrivere anche perché sono pensate e studiate per ottenere una certa velocità di verifica dei programmi. Il linguaggio naturale è ben più complesso: immagina quanto dev'essere difficile descrivere una grammatica per il linguaggio naturale ed immagina poi quale complessità può avere un algoritmo di verifica di tale grammatica, ammesso che questo possa esistere. Pensa solo al fatto che spesso e volentieri il linguaggio naturale dipende dal contesto! Questa cosa nei linguaggi di programmazione non s'è mai vista per fortuna. I parser per linguaggi di programmazione hanno complessità lineare di norma, se si va a vedere i parser del linguaggio naturale. . . sayonara!

Capitolo 3

Proprietà dei linguaggi liberi

È il momento di fare una gita domenicale di famiglia presso il supermercato MondoMatematica®e perderci tra i gli scaffali invitanti e variopinti, stregati dalla capitalistica attrazione che tutte quelle proprietà, quei lemmi e quei teoremi esercitano su di noi.

3.1 Proprietà di chiusura

Chiusura rispetto all'unione

Lemma 3.1.1. *La classe dei linguaggi liberi è chiusa rispetto all'operazione di unione d'insiemi.*

$$\textit{Se } \mathcal{L}_1 \textit{ è libero} \wedge \mathcal{L}_2 \textit{ è libero} \implies \mathcal{L}_1 \cup \mathcal{L}_2 \textit{ è libero}$$

Dimostrazione. Siano $\mathcal{L}_1$ e $\mathcal{L}_2$ dei linguaggi liberi; questo vuol dire che devono esistere due grammatiche libere, ciascuna delle quali ha generato uno dei due linguaggi. In simboli:

$$\exists\, \mathcal{G}_1 = (V_1, T_1, S_1, \mathcal{P}_1), \mathcal{G}_2 = (V_2, T_2, S_2, \mathcal{P}_2) : \mathcal{L}_1 = \mathcal{L}(\mathcal{G}_1) \text{ e } \mathcal{L}_2 = \mathcal{L}(\mathcal{G}_2)$$

Vediamo quindi come costruire una nuova grammatica libera che possa generare l'unione dei due linguaggi generati dalle rispettive grammatiche viste sopra. Consideriamo quindi una nuova grammatica $\mathcal{G}$, i cui elementi sono così definiti:

$$\mathcal{G} = (V_1' \cup V_2' \cup \{S\}, T_1 \cup T_2, S, \mathcal{P}_1' \cup \mathcal{P}_2' \cup \{S \to S_1' \mid S_2'\})$$

Andiamo a vedere da vicino come sono stati ricavati gli elementi:

- V_1' e V_2' si ottengo effettuando un'operazione di *refresh* sui non terminali, in modo da evitare collisioni di nomi (vedremo più avanti come mai quest'operazione è importante), inoltre questo vocabolario è completato con l'introduzione di un nuovo start symbol;
- l'insieme dei terminali è l'unione dei sue insiemi di terminali di $\mathcal{G}_1$ e $\mathcal{G}_2$;
- S è il nuovo start symbol, non presente in $V_1' \cup V_2'$, e S_1' e S_2' sono di nuovo il refresh degli start symbol rispettivamente di $\mathcal{G}_2$ e $\mathcal{G}_2$;
- infine, l'insieme delle produzioni è dato dall'unione tra i refresh delle produzioni delle due grammatiche di partenza $\mathcal{G}_1$ e $\mathcal{G}_2$, unito a una nuova produzione che, dal nuovo start symbol, produce il resfresh dello start symbol di una delle due grammatiche di partenza.

Questa definizione ci basta per asserire che $\mathcal{L}(\mathcal{G})$ è libero e $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}_1) \cup \mathcal{L}(\mathcal{G}_2)$.
QED

Osservazione 3.1.1.1. *Cerchiamo di capire come mai possiamo essere certi che la grammatica $\mathcal{G}$, per come l'abbiamo definita, è libera.*

Prendiamo l'insieme delle produzioni di $\mathcal{G}$; abbiamo già visto come è stato ricavato. Il refreshing va a colpire solamente i nomi dei terminali o non-terminali, non la forma delle produzioni che li interessano, la quale rimane immutata ($A \to \alpha$). Questa forma è rispettata anche dalle due nuove produzioni aggiunte ($S \to S_1'$ e $S \to S_2'$) e quindi non vengono introdotte produzioni che rendono la grammatica context dependent.

Osservazione 3.1.1.2. *Cerchiamo di capire come mai $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}_1) \cup \mathcal{L}(\mathcal{G}_2)$ è valido.*

- *Consideriamo una stringa $w \in \mathcal{L}(\mathcal{G})$; questo è vero se e solo se esiste una certa derivazione per ottenerla a partire dallo start symbol ($S \implies {}^*w$).*
- *Per come abbiamo definito la grammatica $\mathcal{G}$, il secondo passo della derivazione dev'essere necessariamente il refresh di uno dei due start symbol delle grammatiche di partenza ($S \implies S_1' \implies {}^*w \vee S \implies S_2' \implies {}^*w$);*
- *questo naturalmente è valido se e solo se, a questo punto, la stringa w può essere derivata in qualche modo a partire da uno degli start symbol delle grammatiche di partenza ($S_1 \implies {}^*w \vee S_2 \implies {}^*w$).*
- *Se w può essere ottenuta attraverso qualche derivazione a partire da S_1 o S_2, allora vuol dire che w appartiene a uno dei due linguaggi generati dalle nostre due grammatiche di partenza ($w \in \mathcal{L}(\mathcal{G}_1) \vee w \in \mathcal{L}(\mathcal{G}_2)$).*
- *Questo è equivalente ad affermare che w appartiene all'unione dei due linguaggi ($w \in \mathcal{L}(\mathcal{G}_1) \cup \mathcal{L}(\mathcal{G}_2)$), e ci convince quindi della validità dell'asserto.*

Osservazione 3.1.1.3. *È semplice convincersi dell'importanza dell'operazione di refreshing. Se consideriamo due grammatiche $\mathcal{G}_1, \mathcal{G}_2$ con le seguenti produzioni:*

$$\begin{aligned} \mathcal{G}_1 : S_1 &\to aA & \qquad \mathcal{G}_2 : S_2 &\to bA \\ A &\to a & A &\to b \end{aligned}$$

Avremo quindi generati i seguenti linguaggi: $\mathcal{L}(\mathcal{G}_1) = \{aa\}$ e $\mathcal{L}(\mathcal{G}_2) = \{bb\}$.

Se non effettuassimo refreshing, ci troveremo ad avere una nuova grammatica con le seguenti produzioni:

$$\begin{aligned} \mathcal{G} : S &\to S_1 \mid S_2 \\ S_1 &\to aA \\ S_2 &\to bA \\ A &\to a \mid b \end{aligned}$$

Il linguaggio prodotto da questa nuova grammatica sarebbe diverso da quello dato dall'unione delle due grammatiche di partenza, poiché avremmo $\mathcal{L}(\mathcal{G}) = \{aa, ab, ba, bb\} \neq \mathcal{L}(\mathcal{G}_1) \cup \mathcal{L}(\mathcal{G}_2)$.

Con un'opportuno resfreshing otteniamo invece:

$$\begin{aligned} \mathcal{G} : S &\to S_1 \mid S_2 \\ S_1 &\to aA \\ S_2 &\to bB \\ A &\to a \\ B &\to b \end{aligned}$$

Questo insieme di produzioni garantisce che la grammatica generi il linguaggio desiderato.

Chiusura rispetto alla concatenazione

Lemma 3.1.2. *La classe dei linguaggi liberi è chiusa rispetto all'operazione di concatenazione.*

Se $\mathcal{L}_1$ è libero $\wedge$ $\mathcal{L}_2$ è libero $\implies$ $\{w_1 w_2 \mid w_1 \in \mathcal{L}_1 \wedge w_2 \in \mathcal{L}_2\}$ è un linguaggio libero.

Dimostrazione. Consideriamo due linguaggi liberi $\mathcal{L}_1$ e $\mathcal{L}_2$; ciò implica che esistano due grammatiche libere $\mathcal{G}_1$ e $\mathcal{G}_2$ che soddisfano la seguente equazione:

$$\exists\, \mathcal{G}_1 = (V_1, T_1, S_1, \mathcal{P}_1), \mathcal{G}_2 = (V_2, T_2, S_2, \mathcal{P}_2) : \mathcal{L}_1 = \mathcal{L}(\mathcal{G}_1) \text{ e } \mathcal{L}_2 = \mathcal{L}(\mathcal{G}_2)$$

Consideriamo quindi una nuova grammatica $\mathcal{G}$, i cui elementi sono così definiti:

$$\mathcal{G} = (V_1' \cup V_2' \cup \{S\}, T_1 \cup T_2, S, \mathcal{P}_1' \cup \mathcal{P}_2' \cup \{S \to S_1' S_2'\})$$

La definizione dei vari elementi della precedente equazione procede pari passo a quella già vista nel lemma della chiusura rispetto all'unione. L'unica differenza è la definizione del linguaggio $\mathcal{L}(\mathcal{G})$ che appunto qui è costruito per concatenazione e non per unione:

$$\mathcal{L}(\mathcal{G}) = \{w_1 w_2 \mid w_1 \in \mathcal{L}(\mathcal{G}_1) \wedge w_2 \in \mathcal{L}(\mathcal{G}_2)\}$$

La dimostrazione del lemma di chiusura rispetto alla concatenazione procede pari passo a quella del lemma di chiusura rispetto all'unione, di conseguenza non è qui riportata. QED

Chiusura rispetto all'interezione

Lemma 3.1.3. *La classe dei linguaggi liberi non è chiusa rispetto all'operazione d'intersezione.*

Dimostrazione. La dimostrazione è per contraddizione, è sufficiente trovare due linguaggi liberi la cui intersezione non è libera.

Si considerino, ad esempio, due linguaggi $\mathcal{L}_1$ e $\mathcal{L}_2$, definiti come segue:

$$\mathcal{L}_1 = \{a^n b^n c^j \mid n, j > 0\}$$
$$\mathcal{L}_2 = \{a^j b^n c^n \mid n, j > 0\}$$

Il linguaggio generato dalla loro intersezione è il seguente:

$$\mathcal{L}_1 \cap \mathcal{L}_2 = \{a^n b^n c^n \mid n > 0\}$$

Questo linguaggio non è libero, e provarlo è molto semplice (si veda Tab. 3.1). QED

3.2 Chomsky Normal Form

Definizione Una grammatica libera $\mathcal{G} = (V, T, S, \mathcal{P})$ è in *Chomsky Normal Form* se e solo se:

- non ha alcuna ε-produzione, al massimo $S \to \varepsilon$;
- tutte le sue non ε-produzioni hanno una tra le due seguenti forme:
  - $A \to a$
  - $A \to BC$

  in cui sia B sia C sono diversi da S.

Lemma 3.2.1. *Sia $\mathcal{G}$ una grammatica libera; allora esiste una certa grammatica $\mathcal{G}'$ in Chomsky Normal Form tale che $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}')$.*

La dimostrazione del lemma è lasciata per esercizio.

3.3 Come epurare le grammatiche libere

Esiste un teorema che ci permette di fare pulizia etnica sui linguaggi.

Teorema 3.3.1. *Consideriamo un linguaggio libero $\mathcal{L}$; allora esiste una grammatica libera $\mathcal{G}$ tale che $\mathcal{L}(\mathcal{G}) = \mathcal{L} \setminus \{\varepsilon\}$ e tale da:*

- *non avere alcuna ε-produzione, ovvero produzioni di forma $A \to \varepsilon$;*
- *non avere alcuna produzione d'unità (*unit production*), ovvero produzione di forma $A \to B$;*
- *non avere alcun non-terminale non utile, ovvero non-terminali che non appaiono mai in alcuna derivazione di qualche stringa di terminale.*

Le grammatiche che possiedono queste caratteristiche sono più sintetiche e generano linguaggi equivalenti a livello espressivo, ma più puliti; ad esempio, non c'è alcun bisogno di avere non-terminali che abbiano una produzione in ε e, qualora richiesto, la produzione $S \to \varepsilon$ sarebbe sufficiente allo scopo, spiegheremo in seguitoquesta affermazione.

Non è nostro interesse entrare nel dettaglio degli algoritmi per effettuare questa pulizia, poiché dimostrarne la correttezza richiederebbe un considerevole impegno. Andremo però ad analizzare come possiamo, se non altro, eliminare le ε-produzioni.

3.3.1 Come eliminare le produzioni di parole vuote

Per eliminare le ε-produzioni bisogna procedere ad individuare i non-terminali *annullabili* (*nullable*), ovvero quei non-terminali che, con qualche derivazione, producono la parola vuota ($A : A \to^* \varepsilon$). Si può procedere ricorsivamente:

Base se la grammatica comprende una produzione della forma $A \to \varepsilon$, allora quel non-terminale A è annullabile;

Step se la grammatica comprende una produzione della forma $A \to Y_1 Y_2 \dots Y_n$ e i non terminali $Y_1, Y_2, \dots, Y_3$ sono tutti annullabili, allora anche A è annullabile.

A questo punto, andiamo a sostituire ogni produzione della forma $A \to Y_1 Y_2 \dots Y_n$ con una famiglia di produzioni dove ogni combinazione di Y_i annullabili è rimossa dal body della produzione. Se tutti gli Y_i sono annullabili, la stessa produzione $A \to Y_1 Y_2 \dots Y_n$ diventa di questa forma $A \to \varepsilon$ e non viene quindi inclusa nelle produzioni finali. Quindi, si elimina ogni produzione $A \to \varepsilon$.

Esempio di applicazione Andiamo a vedere un esempio considerando la seguente grammatica:

$$\begin{aligned} S &\to ABC \mid abc \\ A &\to aB \mid \varepsilon \\ B &\to bA \mid C \\ C &\to \varepsilon \end{aligned}$$

Osserviamo subito che C ha solo una produzione e questa conduce a ε, pertanto è annullabile; allo stesso modo, B è annullabile attraverso C e S è annullabile attraverso A, B e C. Con le operazioni che abbiamo illustrato poc'anzi, la grammatica che otteniamo è la seguente:

$$\begin{aligned} S &\to ABC \mid abc \mid AB \mid AC \mid BC \mid A \mid B \mid C \\ A &\to aB \\ B &\to bA \mid C \end{aligned}$$

Si noti che, in questa grammatica, il non-letterale C è non utile.

3.4 Pumping lemma per linguaggi liberi

3.4.1 Definizione e formulazione

Si segua lentamente formulazione e dimostrazione per questo tanto articolato quanto importante lemma.

Lemma 3.4.1. *Consideriamo un linguaggio libero $\mathcal{L}$, allora:*

- $\exists p \in \mathbb{N}^+$ *tale che*
- $\forall z \in \mathcal{L} : |z| > p$, *allora*
- $\exists u, v, w, x, y$ *tale che:*
  - $z = uvwxy \wedge$
  - $|vwx| \leq p \wedge$
  - $|vx| > 0 \wedge$
  - $\forall i \in \mathbb{N}. uv^i wx^i y \in \mathcal{L}$

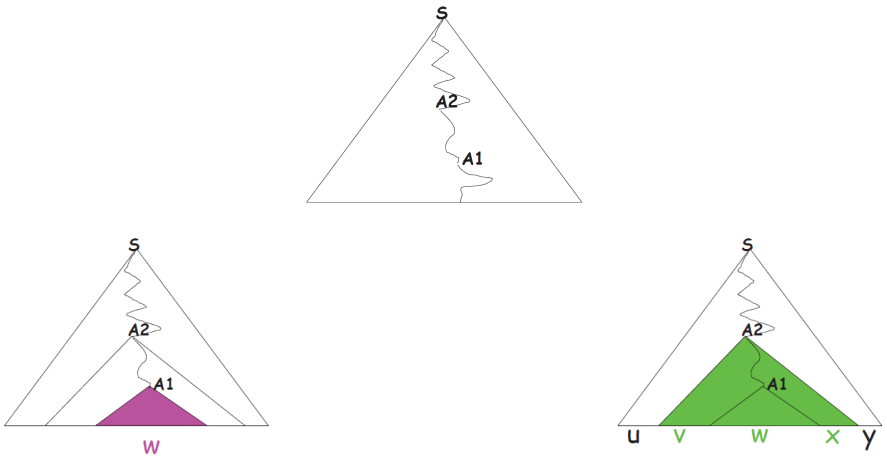
Dimostrazione. Partiamo considerando un linguaggio libero $\mathcal{L}$ che non comprende la parola vuota ($\varepsilon \notin \mathcal{L}$); sappiamo quindi che deve esistere una grammatica libera "epurata" $\mathcal{G}$ che generi il linguaggio $\mathcal{L}$ ($\mathcal{L} = \mathcal{L}(\mathcal{G})$).

Osserviamo che, per ogni possibile albero di derivazione, ogni cammino dalla radice a un terminale attraversa tanti non-terminali quanto il valore della lunghezza del cammino stesso. Ad esempio, la lunghezza del cammino $\langle S, B_1, B_2, \ldots, B_{k-1}, a\rangle$ è k, così come il numero di non-terminali attraversati.

Adesso definimo p come la lunghezza della più lunga parola derivabile con degli alberi di derivazione, e che inoltre sia tale che i suoi camini, a partire dalla radice, siano lunghi al più come il numero di non-terminali di $\mathcal{G}$. Consideriamo quindi anche una parola $z \in \mathcal{L}$, più lunga di p ($|z| > p$). Possiamo quindi essere sicuri che esiste un albero di derivazione per z che possiede almeno un cammino la cui lunghezza è strettamente maggiore del numero di non-terminali, poiché $|z| > p$.

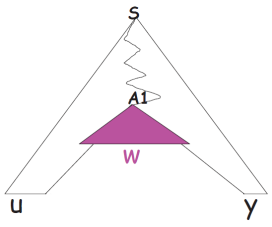
Adesso, per un qualche albero di derivazione, andiamo a considerare la più profonda coppia di occorrenze dello stesso non-terminale lungo un qualche cammino; con più profonda coppia di occerrenze intendiamo la coppia composta dalla più lontana e dalla seconda più lontana occorrenza dalla radice di uno stesso non terminale.

Andiamo a visualizzare la situazione attraverso un albero di derivazione, considerando le due occorrenze di un non-terminale A. Possiamo dire, nelle due occorrenze di A, sono radicati due sottoalberi distinti, che noi qui rappresenteremo



colorati di verde e porpora. La parola z può quindi essere "spacchettata" in cinque sottoparole, come illustrato di seguito. Dal momento che A_1 e A_2 sono occorrenze di uno stesso non-terminale, condividono la medesima famiglia di produzioni. Questo fatto è notevole: se, a partire da A_2, siamo riusciti a trovare una qualche sequenza di derivazioni tale da essere arrivati a A_1, allora con la stessa sequenza possiamo trovare una nuova occorrenza di A, il che ci rende ragionevolmente certi che possiamo generare un numero arbitrario di sottostringhe v, x di z.

Allo stesso modo sappiamo che, se a partire da A_1 possiamo ottenere un sottoalbero di derivazioni w, questo stesso sottoalbero può essere ottenuto a partire da A_2. Queste affermazioni possono essere riassunte, per l'appunto, nella



formula:

$$\forall i \in \mathbb{N}.uv^iwx^iy \in \mathcal{L}$$

Ulteriori considerazioni:

- per come abbiamo scelto le occorrenze di A, la profondità del sottoalbero radicato in A_2 è minore del numero di non-terminali presenti nella grammatica considerata, da cui, per definizione stessa del valore p, $|vwx| < p$;
- la grammatica $\mathcal{G}$ che genera il linguaggio che stiamo considerando è ripulita, per cui nelle derivazioni di forma $A \implies {}^*\alpha A\beta$, almeno uno dei due simboli α, β deve fornire uno ulteriore; questo ci permette di affermare che $|vx| > 0$.

Dimostrazione se $\varepsilon \in \mathcal{L}$ La dimostrazione che abbiamo appena concluso è relativa a linguaggi privi di parola vuota, ma con un attimo di cura nell'impostazione possiamo estenderla anche ai linguaggi che comprendono ε.

Considerando un linguaggio $\mathcal{L}$ che comprenda ε, partiamo ricavando una grammatica $\mathcal{G} = (V, T, S, \mathcal{P})$ che generi $\mathcal{L} \setminus \{\varepsilon\}$. A questo punto definiamo una seconda nuova grammatica $\mathcal{G}'$, in cui andiamo semplicemente ad aggiungere un nuovo start symbol assieme a due produzioni: una che punti verso lo start symbol della precedente grammatica, e una seconda che generi invece ε. Formalmente:

$$\mathcal{G}' = (V, T, S', \mathcal{P} \cup \{S' \to S, S' \to \varepsilon\})$$

In questo modo abbiamo che $\mathcal{L} = \mathcal{L}(\mathcal{G}')$ e possiamo procedere analogamente a prima

QED

3.4.2 Applicazioni

Scopo del pumping lemma Tutto (quasi) bello, ma a cosa serve questo pumping lemma? È presto detto: il pumping lemma di permette di determinare con certezza se un certo linguaggio $\mathcal{L}$ *non* è libero. Perché mai dovremmo volere questa informazione? Perché gli algoritmi di parsing[1] che abbiamo a disposizione per verificare se una parola appartiente a un linguaggio generato da una grammatica libera sono esageratamente più efficienti di quelli che abbiamo per per i linguaggi generati da grammatiche dipendenti da contesto[2].

[1] Il processo inverso della derivazione; non abbiate fretta, che fra poco arrivano un centinaio di pagine solo su quello.

[2] Tutti questi algoritmi sono al più $\mathcal{O}(n^3)$, ma per i linguaggi generati da alcune sottoclassi di grammatiche libere abbiamo a disposizione algoritmi lineari.

Il negato del pumping lemma Di preciso, come possiamo usare il pumping lemma per ottenere quest'informazione? Pensiamo al fatto che questo lemma è, formalmente, una semplice implicazione logica $A \implies B$, dove l' ipotesi A è che un certo linguaggio $\mathcal{L}$ sia libero e B è invece la tesi stessa del pumping lemma, la quale afferma che $\mathcal{L}$ avrà una certa forma. Questo vuol dire che, usando il suo negato $\neg(A \implies B) \equiv \neg B \implies \neg A$, possiamo vedere come verificare il negato della tesi del pumping lemma per un linguaggio $\mathcal{L}$ implichi necessariamente che $\mathcal{L}$ non è libero:

$$\neg ThesisPL(\mathcal{L}) \implies \mathcal{L} \text{ non è libero, } \forall \mathcal{L} \tag{3.1}$$

Insomma, dobbiamo verificare il negato della tesi del pumping lemma per un certo linguaggio $\mathcal{L}$. La tesi del pumping lemma però appare alquanto ostica da manipolare:

$$\neg(\exists p \in \mathbb{N}^+ . \forall z \in \mathcal{L} : |z| > p . \exists u, v, w, x, y . P)$$

Dove quel vaso di pandora P è definito come $P \equiv P_1 \wedge P_2 \wedge P_3 \wedge P_4$:

- $P_1 \equiv z = uvwxy \wedge$
- $P_2 \equiv |vwx| \leq p \wedge$
- $P_3 \equiv |vx| > 0 \wedge$
- $P_4 \equiv \forall i \in \mathbb{N} . uv^i wx^i y \in \mathcal{L}$

Richiamando alla mente le regole di equivalenza tra i quantificatori ($\neg\forall . x \equiv \exists . x$ e $\neg\exists . x \equiv \forall . x$) e facendo un passaggio per volta non dovrebbe esserce difficile raggiungere il risultato cercato:

$$\neg(\exists p \in \mathbb{N}^+ . \forall z \in \mathcal{L} : |z| > p . \exists u, v, w, x, y . P)$$

$$\forall p \in \mathbb{N}^+ . \neg(\forall z \in \mathcal{L} : |z| > p . \exists u, v, w, x, y . P)$$

$$\forall p \in \mathbb{N}^+ . \exists z \in \mathcal{L} : |z| > p . \neg(\exists u, v, w, x, y . P)$$

$$\forall p \in \mathbb{N}^+ . \exists z \in \mathcal{L} : |z| > p . \exists u, v, w, x, y . \neg(P)$$

L'espressione P sarà invece manipolata in modo un po' meno "liscio", perché questa formulazione ci risulterà più comoda nelle applicazioni reali.

$$\neg(P_1 \wedge P_2 \wedge P_3 \wedge P_4)$$

$$\neg((P_1 \wedge P_2 \wedge P_3) \wedge P_4)$$

$$\neg(P_1 \wedge P_2 \wedge P_3) \vee P_4$$

$$(P_1 \wedge P_2 \wedge P_3) \implies \neg P_4$$

$$(P_1 \wedge P_2 \wedge P_3) \implies \neg(\forall i \in \mathbb{N}.uv^iwx^iy \in \mathcal{L})$$

$$(P_1 \wedge P_2 \wedge P_3) \implies \exists i \in \mathbb{N}.\neg(uv^iwx^iy \in \mathcal{L})$$

In sostanza, il negato della tesi del pumping lemma apparirà così:

$$\forall p \in \mathbb{N}^+ \text{ tale che} \tag{3.2}$$
$$\exists z \in \mathcal{L} : |z| > p, \text{ allora} \tag{3.3}$$
$$\forall u, v, w, x, y : (z = uvwxy \ \wedge |vwx| \leq p \ \wedge |vx| > 0) \implies \tag{3.4}$$
$$\exists i \in \mathbb{N}.uv^iwx^iy \notin \mathcal{L}) \tag{3.5}$$

Che tutto questo possa non risultare proprio chiarissimo a tutti è una cosa che francamente non ci stupisce; a tal proposito, proponiamo una descrizione discorsiva di questo negato della tesi del pumping lemma:

> Per un qualsiasi numero naturale p **arbitrariamente** scelto e scegliendo una parola z appartenente al linguaggio e la cui lunghezza è maggiore di p, dobbiamo mostrare che qualsiasi decomposizione di z in sottostringhe di forma $uvwxy$, in cui la lunghezza delle sottostringhe vwx è minore di p e la lunghezza di vx è non nulla, possiamo sempre trovare un indice i per cui la parola z, le cui sottostringhe sono state alterate secondo uv^iwx^iy, non appartiene al linguaggio considerato.

Bene, questo blocco infinito di logica e insiemistica è finito; per chi avesse sentito una paura folle montare dentro, niente paura, adesso passeremo a degli esempi pratici, che sapranno spiegare la teoria su cui essi stessi sono basati meglio di qualsiasi spiegazione a parole.

3.4.3 Esempi di utilizzo

Esercizio 1

Consideriamo questa grammatica $\mathcal{G}$:

$$\begin{aligned} \mathcal{G}: \quad & S \to aSBc \mid abc \\ & cB \to Bc \\ & bB \to bb \end{aligned}$$

L'abbiamo già incontrata, è una grammatica contestuale che genera un linguaggio di questo tipo: $\mathcal{L}(\mathcal{G}) = \{a^n b^n c^n \mid n > 0\}$. A questo punti ci chiediamo se $\mathcal{L}(\mathcal{G})$ è libero oppure no, il che significa, lo ripetiamo, riuscire a trovare una seconda grammatica $\mathcal{G}'$. che generi lo stesso linguaggio ($\mathcal{L}(\mathcal{G}') = \mathcal{L}(\mathcal{G})$) ma che sia anche libera. La risposta è negativa, quello non è un linguaggio libero, ed è il momento di tirare fuori il nostro pumping lemma e rendere ragione di ciò.

Dimostrazione. La dimostrazione è ovviamente per contraddizione, perché la nostra supposizione è che $\mathcal{L}$ sia libero e stiamo per tentare di verificare la validità del negato della tesi del pumping lemma e questo contraddirrebbe la nostra supposizione (Eq.3.1). Diamo il via alle danze:

- prendiamoci un intero $P \in \mathbb{N}^+$ scelto arbitrariamente (Eq.3.2);
- scegliamoci una parola z che appartenga a $\mathcal{L}$ e abbia una lunghezza maggiore di p; noi scegliamo la parola $z = a^p b^p c^p$, che ha lunghezza $|z| = 3p > p$ e certamente appartiene al linguaggio:

  $$\underbrace{aa\ldots a}_{p}\underbrace{bb\ldots b}_{p}\underbrace{cc\ldots c}_{p}$$

  la scelta di quale parola z usare è fondamentale per arrivare agevolmente alla conclusione, quindi si presti bene attenzione in questa fase (Eq.3.3);
- questo è il punto in cui si parla della decomposizione di z (Eq.3.4): consideriamo infatti delle sottostringe u, v, w, x, y arbitrarie, per cui:
  - sono una decomposizione della parola z ($z = uvwxy$);
  - la sottostringa vwx ha dimensione inferiore o uguale a p ($|vwx| \leq p$);
  - le due sottostringe v e x hanno dimensione non nulla ($|vx| > 0$);

  notiamo in particolare quali posizioni potrà assumere la sottostringa vwx all'interno della parola z:

  $$a\,\underbrace{a\ldots a}_{vwx}\,a\,\underbrace{a\ldots abb\ldots b}_{vwx}\,b\,\underbrace{b\ldots b}_{vwx}\,b\,\underbrace{b\ldots bc\ldots c}_{vwx}\,c\,\underbrace{c\ldots c}_{vwx}$$

quindi vwx o non conterrà alcuna occorrenza di c (nei primi tre casi), o non conterrà alcuna occorrenza di a (negli ultimi tre);

- l'implicazione di ques'ultima affermazione è che è possibile costruire una sottoparola vwx non bilanciata, e quindi non appartenente al linguaggio $\mathcal{L}$, semplicemente considerando uv^0wx^0y (Eq.3.5).

Abbiamo quindi verificato che il negato del pumping lemma, per il nostro linguaggio $\mathcal{L}$, non è valido: questo contraddice la nostra supposizione e implica che $\mathcal{L}$ non è libero. QED

Esercizio 2

$$
\begin{aligned}
\mathcal{G}: \quad S &\to CD \\
C &\to aCA \mid bCB \mid \varepsilon \\
AD &\to aD \\
BD &\to bD \\
Aa &\to aA \\
Ab &\to bA \\
Ba &\to aB \\
Bb &\to bB \\
D &\to \varepsilon
\end{aligned}
$$

Il linguaggio generato Trovandoci davanti a una grammatica contestuale così articolata, la prima cosa che dobbiamo chiederci è quale sia il linguaggio $\mathcal{L}$ generato. Facciamo delle rapide considerazioni:

- la produzione iniziale è sempre $S \to CD$, per cui non possiamo prescindere dallo sviluppo di C (nel prossimo punto capiremo perché è importante);
- le stringhe crescono in lunghezza solamente grazie sviluppo del non-terminale C;
- il non-terminale D agisce come un delimitatore destro, dal momento che quando ha alla sua sinsitra un terminale A o B ci permette di svilupparlo nel relativo terminale;
- quando un terminale (a o b) si trova a destra di un non-terminale (B o B), la posizione di questi può essere scambiata ($Ab \to bA$).

Proviamo quindi a sviluppare delle derivazioni di prova e vedere cosa succede.

$\underline{S}$	con $S \to CD$
$\Rightarrow \underline{C}D$	con $C \to aCA$
$\Rightarrow a\underline{C}AD$	con $C \to aCA$
$\Rightarrow aa\underline{C}AAD$	con $C \to bCB$
$\Rightarrow aab\underline{C}BAAD$	con $C \to \varepsilon$
$\Rightarrow aabBA\underline{AD}$	con $AD \to aD$
$\Rightarrow aabB\underline{Aa}D$	con $Aa \to aA$
$\Rightarrow aab\underline{Ba}AD$	con $Ba \to aB$
$\Rightarrow aabaB\underline{AD}$	con $AD \to aD$
$\Rightarrow aaba\underline{Ba}D$	con $Ba \to aB$
$\Rightarrow aabaa\underline{BD}$	con $BD \to bD$
$\Rightarrow aabaab\underline{D}$	con $D \to \varepsilon$
$\Rightarrow aabaab$	

A questo punto, sfruttando anche un filo di intuizione e fantasia, possiamo concludere che il linguaggio generato comprende tutte quelle parole formate dalla ripetizione di una stessa parola su a e b: $\mathcal{L}(\mathcal{G}) = \{ww \mid w \in \{a, b\}^*\}$; questo linguaggio, sfortunatamente, non è libero.

Dove la chiusura alla concatenazione fallisce Prima di vedere la dimostrazione effettiva, però, qualcuno potrebbe giustamente chiedersi se possiamo fare la dimostrazione considerando un linguaggio $\mathcal{L}'(\mathcal{G}) = \{ww \mid w \in \{a, b\}^*\}$ utilizzando la chiusura rispetto alla concatenazione. Questo però non è corretto, perché le parole che appartengono a $\mathcal{L}(\mathcal{G})$ hanno una struttura precisa, ossia possono sempre essere divise in due sottostringe di pari lunghezza ed equivalenti; la chiusura alla concatenazione, impostata in quel modo, rischia di non rispettare questa struttura[3].
Detto questo, andiamo a vedere rapidamente la dimostrazione corretta.

Dimostrazione. Dal momento che la dimostrazione procede in maniera analoga a quella precedente, focalizziamoci su uno dei punti cardini, ossia la scelta di z: in questo caso scegliamo $z = a^p b^p a^p b^p$, perché ci permette di concludere la dimostrazione quasi a colpo d'occhio, perché a questo punto possiamo decomporre z in $uvwxy$ e scegliere $vwx = b^p$; questa vwx rispetta il vincolo $|vwx| \leq p$ e possiamo

[3]Questa ultima affermazione è moooolto dubbia, vorrei approfondire but damn time is running out.

sbilanciarla scegliendo, di nuovo, un indice 0 (uv^0wx^0y), per cui il negato del pumping lemma è verificato e il linguaggio considerato non è libero. QED

3.5 Esercizi di riepilogo

Esercizio 1 - I seguenti linguaggi sono liberi?

Linguaggio	Risposta
$\{a^n b^n c^n \mid n > 0\}$	Non è libero

Tabella 3.1: Esercizio 1

Spiegazione 3.1 Come osservato per l'esempio precedente, questo linguaggio non è libero e ciò si dimostra mediante il Pumping Lemma.

Linguaggio	Risposta
$\{a^n b^n c^j \mid n, j > 0\}$	Libero

Tabella 3.2: Esercizio 2

Spiegazione 3.2 Per provare che questo linguaggio è libero possiamo sfruttare la chiusura dei linguaggi liberi rispetto alla concatenazione. È semplice trovare una grammatica libera che generi il linguaggio $\{a^n b^n \mid n > 0\}$, come ad esempio la seguente:

$$S \rightarrow aSb \mid ab$$

Altrettanto semplice è farlo per il linguaggio $\{c^j \mid j > 0\}$ (qui usiamo C come start symbol):

$$C \rightarrow c \mid cC$$

A questo punto, sfruttando la proprietà di concatenazione, riusciamo a identificare una grammatica libera $\mathcal{G}$ che, senza alcun problema, ci consente di ottenere il nostro linguaggio di partenza:

$$
\begin{aligned}
S &\rightarrow SC \\
S &\rightarrow aSb \mid ab \\
C &\rightarrow c \mid cC
\end{aligned}
$$

Si noti che questa procedura, con le dovute accortezze, è la medesima utilizzata per ricavare la risposta di 3.3.

Linguaggio	Risposta
$\{a^j b^n c^n \mid n, j > 0\}$	Libero

Tabella 3.3: Esercizio 3

Spiegazione 3.3 Si proceda analogamente a 3.2.

Esercizio 2

Consideriamo la seguente grammatica:

$$
\begin{aligned}
\mathcal{G} : S &\rightarrow aSc \mid aTc \mid T \\
T &\rightarrow bTa \mid ba
\end{aligned}
$$

Ci poniamo quindi i seguenti problemi: $\mathcal{G}$ è ambigua? Qual è il linguaggio $\mathcal{L}(\mathcal{G})$ generato?

È molto semplice dimostrare che $\mathcal{G}$ è ambigua, si veda come, con le due seguenti derivazioni, si giunge ad ottenere la medesima stringa $w = abac$.

$$
\begin{aligned}
S &\Longrightarrow aSc \Longrightarrow aTc \Longrightarrow abac \\
S &\Longrightarrow aTc \Longrightarrow abac
\end{aligned}
$$

Per convincerci che le derivazioni sono entrambe rightmost, possiamo tracciare un albero di derivazione per $\mathcal{G}$.

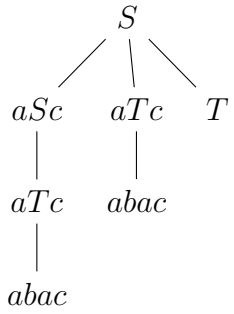


Figura 3.1: Albero di derivazione per $\mathcal{G}$

Infine, andiamo a definire il linguaggio $\mathcal{L} = \mathcal{L}(\mathcal{G})$, molto semplice da determinare:

$$\mathcal{L} = \{a^n b^m a^m c^n \mid n \geq 0,\ m > 0\}$$

Esercizio 3

In questo esercizio ci chiediamo quale sia il linguaggio generato dalla seguente grammatica.

$$\begin{aligned} \mathcal{G} : S &\rightarrow 0B \mid 1A \\ A &\rightarrow 0 \mid 0S \mid 1AA \\ B &\rightarrow 1 \mid 1S \mid 0BB \end{aligned}$$

Il linguaggio generato è tale che ogni sua stringa possiede un egual numero di 0 e 1.

Intuitivamente, è facile convincersene: supponiamo infatti di produrre, a partire da S, una stringa $0B$. A questo punto ci sono dati tre casi:

- se applichiamo $B \rightarrow 1$ andiamo a chiudere la derivazione con una stringa "bilanciata" (si passi sopra all'abuso di terminologia);
- se invece applichiamo la seconda produzione possibile $B \rightarrow 1S$ non stiamo facendo altro che tornare al caso base del ragionamento, mantenendo comunque una stringa bilanciata;
- l'unico modo che abbiamo per introdurre un altro 0 è applicare la terza produzione $S \rightarrow 0BB$, ma quei due B, a questo punto, non hanno altro modo di terminare se non diventando degli 1.

Il linguaggio generato può essere scritto come segue:

$$\{w \mid w \in \{0,1\}^* \wedge\ |0_w| = |1_w|\}$$

Esercizio 4

Si richiede di dire quali grammatiche generano i seguenti linguaggi:

1. $\mathcal{L}_1(\mathcal{G}_1) = \{a^k b^n c^{2k} \mid k, n > 0\}$
2. $\mathcal{L}_2(\mathcal{G}_2) = \{a^k b^n c^{2k} \mid k, n \geq 0\}$

Soluzione 4.1 La soluzione quindi avrà la seguente forma:

$$\begin{aligned} S &\rightarrow aScc \mid aTcc \\ T &\rightarrow bT \mid b \end{aligned}$$

Soluzione 4.2 Nonostante l'esercizio, in apparenza, sia solo leggermente diverso dal precedente, ci dobbiamo porre il problema di come aggiungere la generazione della parola vuota ε al linguaggio. Questo può essere risolto in maniera molto elegante con una grammatica context-free e non ambigua, come ad esempio quella seguente:

$$\begin{aligned} S &\rightarrow aScc \mid T \\ T &\rightarrow bT \mid \varepsilon \end{aligned}$$

Esercizio 5 - Grammatiche context dependent

Considerata la seguente gramatica $\mathcal{G}$, è vero che $\mathcal{L}(\mathcal{G}) = \varnothing$?

$$\begin{aligned} \mathcal{G} : S &\rightarrow aBS \mid bA \\ aB &\rightarrow Ac \mid a \\ bA &\rightarrow S \mid Ba \end{aligned}$$

Il modo miglior per procedere alla soluzione di questo esercizio è procedere alla stesura dell'albero di derivazione.

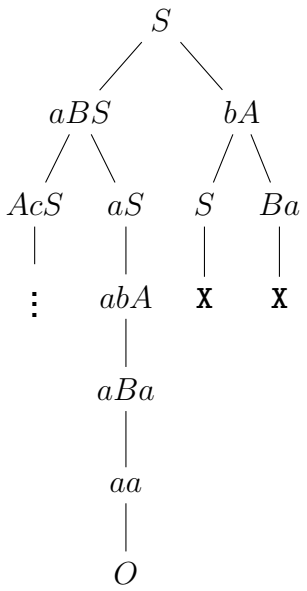


Figura 3.2: Albero di derivazione per $\mathcal{G}$

Dal momento che siamo riusciti a trovare almeno una parola $w = aa \in \mathcal{L}(\mathcal{G})$, possiamo dire senza timore di smentita che $\mathcal{L}(\mathcal{G}) \neq \varnothing$.

Esercizio 6

Quest'ultimo esercizio sarà disivo in tre parti.

Parte 1 Si definisca una grammatica $\mathcal{G}$ tale che $\mathcal{L}(\mathcal{G})$ sia l'insieme di tutti i numeri pari nella rappresentazione binaria. Risposta:

$$\mathcal{G} : \; S \; \rightarrow \; 1S \; \mid 0S \; \mid 0$$

Parte 2 Si definisca una grammatica $\mathcal{G}'$ tale che $\mathcal{L}(\mathcal{G}') = \{1^n 0 \mid n \geq 0\}$. Risposta:

$$\mathcal{G}' : \; S \; \rightarrow \; 1S \; \mid 0$$

Parte 3 Ci chiediamo: $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}')$?

La risposta è no. Ad esempio, condieriamo la stringa $w = 00$; questa appartiene a $\mathcal{L}(\mathcal{G})$ attraverso la produzione $S \implies 0S \implies 00$, ma non appartiene a $\mathcal{L}(\mathcal{G}')$.

Capitolo 4

Linguaggi regolari e introduzione agli automi a stati finiti

4.1 Introduzione

Ora che abbiamo acquisito dei mezzi più potenti per affrontare questo corso, ribadiamo il concetto di analisi lessicale: dato il programma in ingresso, l'analisi lessicale restituisce una lista di stringhe che corrisponde alle parti del linguaggio identificate; questa lista di stringhe è nota come *flusso dei token*.

Lo studio del capitolo precedente di consente di sapere come vengono generati linguaggi come il seguente:

$$\{a^n b^n \mid n > 0\} \tag{4.1}$$

Come detto in passato, un liguaggio con questa forma può essere utilizzato, ad esempio, nel caso in cui vogliamo avere un egual numero di parentesi aperte e chiuse. Il problema che vogliamo arrivare a risolvere è, più in generale, riconoscere se una stringa faccia parte o meno del linguaggio generato da una certa grammatica.

Ad esempio, per affrontare il problema del linguaggio 4.1 viene naturale l'utilizzo di una struttura di tipo stack:

1. leggiamo i simboli uno alla volta e li inseriamo nel nostro stack;
2. per ogni a che leggiamo, inseriamola nella pila;
3. se troviamo una b, allora facciamo un pop dalla pila;
4. se ad un certo punto stiamo tentando di togliere un elemento da una pila vuota o se, finita l'analisi, rimangono ancora elementi nella pila, allora c'è qualche errore;

5. nel caso in cui al termine dell'analisi è andato tutto liscio e abbiamo svuotato la pila, allora la parola analizzata appartiene al linguaggio 4.1.

La precedente strategia sicuramente è ottima quando il linguaggio ha una forma simile all'esempio che abbiamo considerato. Ma consideriamo invece la grammatica che genera tutte le parole dell'alfabeto:

$$S \to a \mid b \mid \cdots \mid z \mid aS \mid bS \mid \cdots \mid zS \tag{4.2}$$

Per riconoscere parole generate da una tale grammatica il metodo di analisi naturale è una automa a stati finiti come quella in figura 4.1.

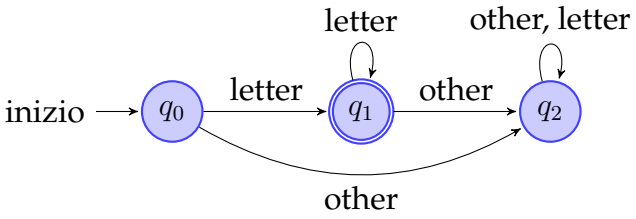


Figura 4.1: Macchina a stati finiti

Il funzionamento di questi strumenti di analisi ci sarà ben più chiaro in seguito.

La grammatica che produce tutte le lettere dell'alfabeto (il linguaggio 4.2) è una grammatica libera, ma ha anche una caratteristica in più: è regolare. Andiamo a capire meglio di cosa si tratta.

4.2 Grammatiche regolari

Le grammatiche regolari sono un sottoinsieme delle grammatiche libere tale che le loro produzioni sono in una delle seguenti forme:

- il body è un solo terminale ($A \to a$);
- il body è composto da un terminale e un non-terminale, nella seguente forma: $A \to aB$;
- il body è la parola vuota ($A \to \varepsilon$).

Queste grammatiche possono generare espressioni regolari, che introdurremo a breve; inoltre, i linguaggi generati da queste grammatiche sono riconosciuti dagli automi a stati finiti, sia deterministici che non deterministici.

Queste due osservazioni sono la base fondante dell'utilizzo delle grammatiche regolari per l'analisi lessicale.

4.2.1 Espressioni regolari

Prima di tutto partiamo con il definire una grammatica regolare.
Sia fissato un alfabeto $\mathcal{A}$ da cui estrarre tutte le basi e sia fissato un certo numero di operatori.

Le espressioni regolari sono esprimibili tramite il meccanismo dell'induzione in questo modo.

Base Sono un'espressione regolare tutti i simboli dell'alfabeto che abbiamo scelto; in aggiunta a questi, anche ε lo è, indipendentemente dall'alfabeto scelto.

Step Se r_1 e r_2 sono espressioni regolari allora:

- $r_1 \mid r_2$ è un'espressione regolare, detta *alternanza*;
- $r_1 \cdot r_2$ è un'espressione regolare, scritta anche come $r_1 r_2$ e detta *concatenazione*;
- ${r_1}^*$ è un'espressione regolare che significa ripetizione di r per un certo numero k di volte, detta *Kleene star*;
- (r_1) è un'espressione regolare; è usata per definire l'ordine di svolgimento delle operazioni ed è detta *parentesi*.

4.2.2 I linguaggi delle espressioni regolari

Se un linguaggio può essere ricavato da un'espressione regolare si dice che l'espressione regolare *denota* quel linguaggio; si presti attenzione a non utilizzare il termine *generare*, poiché quest'ultimo è riservato per le grammatiche generative.

Detto ciò, come capiamo qual è il linguaggio denotato da un'espressione regolare? Consideriamo un'espressione regolare r su $\mathcal{A}$, il linguaggio denotato da quell'espressione $\mathcal{L}(r)$ è anch'esso definibile tramite induzione:

Base
- $\mathcal{L}(a) = \{a\}\ \forall a \in \mathcal{A}$
- $\mathcal{L}(\varepsilon) = \{\varepsilon\}$

Step
- se $r = r_1 \mid r_2$
  allora $\mathcal{L}(r) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$
- se $r = r_1 \cdot r_2$
  allora $\mathcal{L}(r) = \{w_1 w_2 \mid w_1 \in \mathcal{L}(r_1) \wedge w_2 \in \mathcal{L}(r_2)\}$
- se $r = {r_1}^*$
  allora $\mathcal{L}(r) = \{\varepsilon\} \cup \{w_1 w_2 ... w_k \mid k \geq 1 \wedge \forall i : 1 \leq i \leq k . w_i \in \mathcal{L}(r_1)\}$

- se $r = (r_1)$
  allora $\mathcal{L}(r) = \mathcal{L}(r_1)$

Di seguito l'ordine di precedenza per le operazioni appena descritte.

$$\text{Kleene Star} \prec \text{Concatenazione} \prec \text{Alternanza}$$

Tutte queste operazioni hanno associatività a sinistra. Ecco un esempio esplicativo:

$$a \mid bc^*$$

Una volta applicate le regole di precedenza, l'espressione si legge in questo modo:

$$(a \mid (b(c^*)))$$

Esercizi sulle operazioni con espressioni regolari

Ecco presentati in veloce sequenza una serie di semplici esercizi sulle operazioni con grammatiche regolari.

- $\mathcal{L}(a \mid b) = \{a, b\}$;
- $\mathcal{L}((a \mid b)(a \mid b)) = \{aa, ab, ba, bb\}$;
- $\mathcal{L}(a^*) = \{a^n \mid n \geq 0\}$;
- $\mathcal{L}(a \mid a^*b) = \{a\} \cup \{a^n b \mid n \geq 0\}$;
- $(a \mid b \mid \ldots \mid z)(a \mid b \mid \ldots \mid z)^*$ denota l'insieme di tutte le parole generabili con l'alfabeto;
- $(0 \mid 1)^*0$ denota l'insieme di tutti i numeri binari pari;
- $b^*(abb^*)^*(a \mid \varepsilon)$ denota l'insieme delle parole su $\{a, b\}$, senza alcuna occorenza consecutiva di a;
- $(a \mid b)^*aa(a \mid b)^*$ denota l'insieme delle parole su $\{a, b\}$ in cui ci sono sicuramente delle occorrenze consecutive di a (date da aa in posizione centrale).

4.3 Automa a stati finiti

Gli automi a stati finiti sono usati per decidere se le parole appartengono ad un linguaggio denotato da una certa espressione regolare. Vedremo due tipi diversi di automa a stati finiti: deterministico e non deterministico; di questi tipi poi studieremo i casi di utilizzo ottimali.

Nel caso di automa a stati finiti non deterministico i calcoli sono spesso più pesanti, perché un automa non deterministico deve vagliare più percorsi di derivazione rispetto alla loro controparte deterministica, i quali hanno il vantaggio di dover vagliare solo i percorsi deterministici, che sono un sottoinsieme del totale.

4.4 Automa a stati finiti non deterministico

Un automa a stati finiti non deterministico (Non-Deterministic Finite Automata abbreviato in NFA) è rappresentabile con una tupla

$$\mathcal{N} := (S, \mathcal{A}, \text{move}_n, S_0, F) \tag{4.3}$$

in cui:

- S è un insieme di stati;
- $\mathcal{A}$ è un alfabeto con $\varepsilon \notin \mathcal{A}$;
- $s_0 \in S$ è lo stato iniziale;
- $F \subseteq S$ è l'insieme degli stati finali o accettati;
- $\text{move}_n : S \times (\mathcal{A} \cup \{\varepsilon\}) \to 2^S$ è la funzione di transizione: da un certo stato e con un certo simbolo (che può essere anche ε), mi muovo in un certo sottoinsieme di stati compreso nei sottoinsiemi di S.

4.4.1 Rappresentazione grafica

La tupla $(S, \mathcal{A}, \text{move}_n, S_0, F)$ può essere rappresentata con un grafo diretto seguendo queste convenzioni:

- gli stati rappresentano i nodi;
- lo stato iniziale è identificato da una freccia entrante;
- gli stati finali sono rappresentati da un doppio cerchio;

- le funzioni di transizione rappresentano gli archi, ad esempio $\text{move}_n(S_1, a) = \{S_2, S_3\}$ si rappresenta come in figura 4.2.

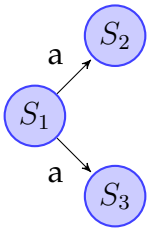


Figura 4.2: Grafo rappresentante un NFA

Si noti anche che, nella rappresentazione grafica, gli archi da S_1 a S_2 ed S_3 vengono denotati da a.

Esempio 1 Proponiamo ora al lettore il seguente esempio (figura 4.3):

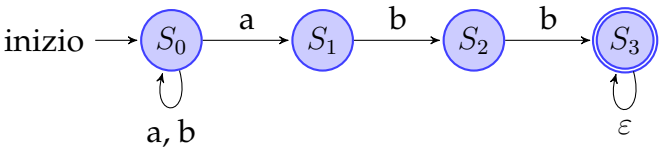


Figura 4.3: Secondo esempio di NFA

L'elemento non deterministico è rappresentato dalle due frecce a uscenti da S_0, che potrebbero portarmi sia in S_1 che in S_0 di nuovo; inoltre, la ε introduce sempre indeterminismo, poiché permette di passare a uno stato successivo senza consumare un carattere del terminale (o produrre un carattere nella stringa, che dir si voglia), e quindi inserisce incertezza riguardo al percorso di derivazione compiuto.

Notiamo che dalla rappresentazione grafica si evincono tutti gli elementi che compongono l'NFA; questa è infatti una rappresentazione completa. Si può anche dare una descrizione tabellare delle funzioni di transizione (move_n) appartenenti a questo NFA:

	ε	a	b
S_0	$\emptyset$	$\{S_0, S_1\}$	$\{S_0\}$
S_1	$\emptyset$	$\emptyset$	$\{S_2\}$
S_2	$\emptyset$	$\emptyset$	$\{S_2\}$
S_3	$\{S_3\}$	$\emptyset$	$\emptyset$

Tabella 4.1: Tabella della funzione di transizione per l'automa 4.3

Qual è quindi il linguaggio accettato da un automa di questo tipo? Un NFA $\mathcal{N}$ accetta (o riconosce) una parola w se e solo se esiste almeno un cammino che parte dallo stato iniziale di $\mathcal{N}$ ed arriva a w.

Ricorda queste regole di scrittura:

- $\varepsilon\varepsilon$ si scrive ε;
- $a\varepsilon$ si scrive a;
- εa si scrive a.

Il linguaggio accettato dall'automa è l'insieme di tutte le parole accettate dall'automa. Ricaviamo, ad esempio, il linguaggio generato dall'automa descritto in figura 4.3:

- a non appartiene al linguaggio, non esiste un percorso che mi porti ad uno stato finale attraversando un solo arco a;
- allo stesso modo non posso trovare la parola b nel linguaggio;
- una parola possibile in questo linguaggio è, ad esempio, abb;
- tutte le parole nella forma $\mathcal{L}(a \mid b)^* abb$ sono parte del linguaggio.

Esempio 2 Proviamo a determinare il linguaggio dell'NFA descritto in figura 4.4:
Il linguaggio accettato da questo NFA è $\mathcal{L}((aa^*) \mid (bb^*))$.

4.5 La costruzione di Thompson

La costruzione di Thompson è una procedura algoritmica che permette di costruire l'automa $\mathcal{N}$ che genera lo stesso linguaggio denotato da una certa espressione regolare r, ovvero $\mathcal{L}(\mathcal{N}) = \mathcal{L}(r)$.

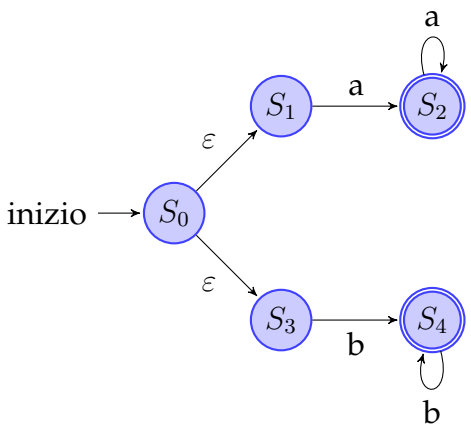


Figura 4.4

4.5.1 Definizione

Anche questa costruzione è espressa in modo induttivo.

Base L'espressione regolare r è o ε, oppure un simbolo dell'alfabeto $a \in \mathcal{A}$; assumiamo di avere sempre un NFA che riconosce $\mathcal{L}(\varepsilon)$ e uno che riconosce $\mathcal{L}(a)$ $\forall a \in \mathcal{A}$.

Step L'espressione regolare r è una tra le seguenti opzioni:

- $r_1 \mid r_2$;
- $r_1 r_2$;
- $r_1{}^*$;
- (r_1);

Dati gli NFA $\mathcal{N}_1$ e $\mathcal{N}_2$ tali che $\mathcal{L}(\mathcal{N}_1) = \mathcal{L}(r_1)$ e $\mathcal{L}(\mathcal{N}_2) = \mathcal{L}(r_2)$, dobbiamo definire i vari NFA per le quattro operazioni $r_1 \mid r_2$, $r_1 r_2$, $r_1{}^*$ ed r_1. Le regole per definire questi nuovi NFA sono elencate successivamente.

A questo punto possiamo fare due osservazioni sulla costruzione di Thompson.

1. Ogni passo della costruzione introduce al massimo due nuovi stati, vale a dire che l'NFA generato contiene al massimo $2k$ stati, dove k è il numero di simboli e operatori nell'espressione regolare.

2. In ogni stato intermedio dell'NFA c'è esattamente uno stato finale, e inoltre lo stato iniziale non ha nessun arco entrante e lo stato finale non ha alcun arco uscente.

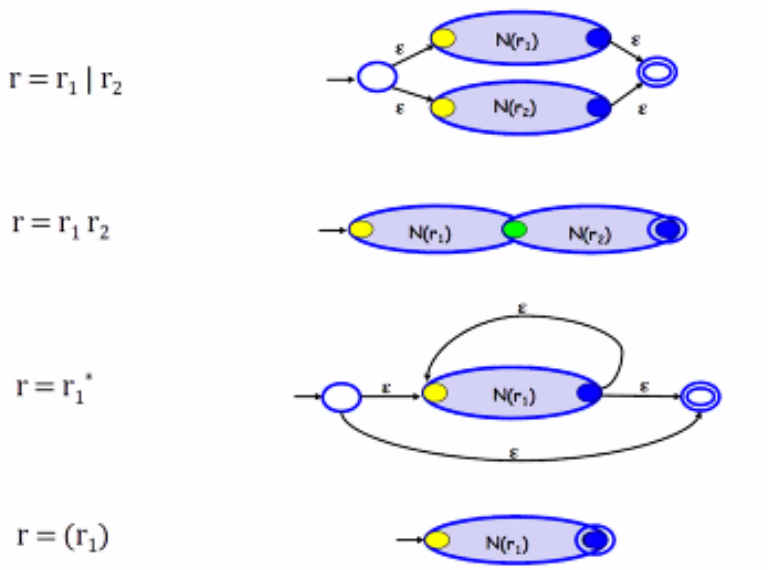


Figura 4.6: Thompson nello step induttivo

4.5.2 Spiegazione in dettaglio

Vediamo ora la rappresentazione grafica di questo algoritmo. Partiamo dai passi base.

Figura 4.5: Thompson nel caso base

La spiegazione dell'algoritmo nel caso base è banale, osserviamo invece con più interesse lo step induttivo.

Supponiamo di avere $\mathcal{N}(r_1)$ e $\mathcal{N}(r_2)$ descritti come in precedenza, discutiamo le applicazioni di Thompson descritte in in figura 4.6.

Alternanza

Guardiamo il primo caso, l'operazione di addizione $r = r_1 \mid r_2$.
Vogliamo creare un automa che accetta parole di $\mathcal{L}(r_1)$ o anche parole di $\mathcal{L}(r_2)$. Abbiamo i due $\mathcal{N}$, che rappresentano gli stati intermedi; li posizioniamo uno sopra all'altro. Ora possiamo creare uno stato vuoto e usarlo come stato iniziale

per entrambi gli $\mathcal{N}$, e quindi collegarlo a questi ultimi tramite una ε-transizione; allo stesso modo possiamo creare uno stato finale raggiungibile dai due $\mathcal{N}$ tramite una ε-transizione.

Concatenazione

Guardiamo ora l'operazione di concatenazione $r = r_1 r_2$.
Per la nostra ipotesi induttiva, sappiamo di poter far affidamento sui due automi $\mathcal{N}(r_1)$ ed $\mathcal{N}(r_2)$; in questo momento ci viene in aiuto quella proprietà della costruzione che dice che i passi intermedi (i due $\mathcal{N}$) hanno esattamente uno stato finale ed uno stato iniziale, senza alcun arco entrante nello stato iniziale e nessun arco uscente dagli stati finali. In questo modo possiamo far coincidere lo stato iniziale di $\mathcal{N}(r_2)$ con lo stato finale di $\mathcal{N}(r_1)$. Di conseguenza, una parola riesce ad arrivare allo stato terminale blu solo se riesce a passare sia da $\mathcal{N}(r_1)$ che da $\mathcal{N}(r_2)$.

Kleene Star

Guardiamo il caso della Kleene star $r = r_1^*$.
Vogliamo un automa che riconosce o ε o tutte le parole che sono composte dalla ripetizione di altre parole appartenenti al linguaggio $\mathcal{L}(r_1)$. Posseggo l'automa $\mathcal{N}(r_1)$, che ha un solo stato iniziale ed un solo stato finale; aggiungo due stati che serviranno come nuovo stato iniziale e nuovo stato finale. Siccome voglio poter avere ε come parola possibile, introduco un nuovo arco ε che collega il nuovo stato iniziale al nuovo stato finale, quindi implemento la ripetizione con degli altri archi ε come in figura; la spiegazione è abbastanza banale.

Parentesi

Il caso della parentesizzazione $r = (r_1)$ risulta estremamente banale: di fatto l'automa non subisce alcuna modifica.

Questo è il modo di utilizzare il processo di Thompson per costruire automi complessi, ci renderemo conto più avanti di quante ε questo ci costringe ad utilizzare e di cosa questo comporti.

4.5.3 Applicazione di Thompson

Presentiamo ora un esempio particolare per assimilare meglio questa costruzione. Analizziamo la seguente espressione regolare

$$r = (a \mid b)^* abb \tag{4.4}$$

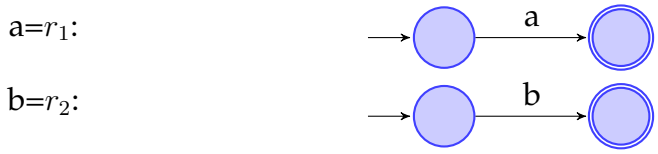


Figura 4.7: Applicazione del passo base della costruzione di Thompson

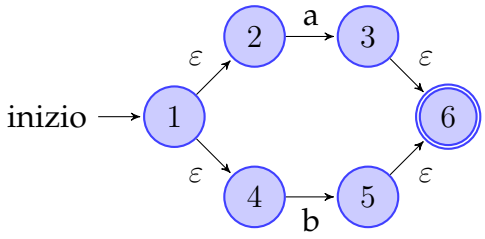


Figura 4.8: Applicazione della costruzione per l'alternanza

Ci sono quindi due espressioni che compaiono in questo esempio: $a = r_1$ e $b = r_2$. In primis applichiamo la regola di base (4.5).

Concentriamoci ora sulla prima espressione, $a \mid b = r_1 \mid r_2 = r_3$; applichiamo la costruzione di Thompson per l'alternanza ai due automi per r_1 e r_2, otteniamo quindi l'automa per r_3.

La procedura è rappresentata in figura 4.8 ed è qui brevemente descritta: prendiamo i due automi per r_1 ed r_2, li mettiamo uno sopra l'altro e generiamo due stati (nodo 1 e nodo 6); il nodo 1 sarà collegato tramite archi ε agli stati iniziali di entrambi gli automi, e simmetricamente il nodo 6 verrà collegato tramite archi ε agli stati finali dei due automi.

Il prossimo passaggio è la traduzione dell'operatore di parentesi, il che è banale, quindi ne riportiamo solo una formulazione nel linguaggio delle espressioni regolari:

$$(a \mid b) = (r_3) = r_4$$

Ora dobbiamo rappresentare la seguente espressione:

$$(a \mid b)^* = r_4^* = r_5$$

Quindi applichiamo la costruzione di Thompson per la Kleene star all'automa per r_4, ottenendo quindi l'automa per r_5.
Continuando con la scansione dell'espressione regolare dobbiamo riscrivere la seguente:

$$(a \mid b)^* a = r_5 r_1 = r_6 \tag{4.5}$$

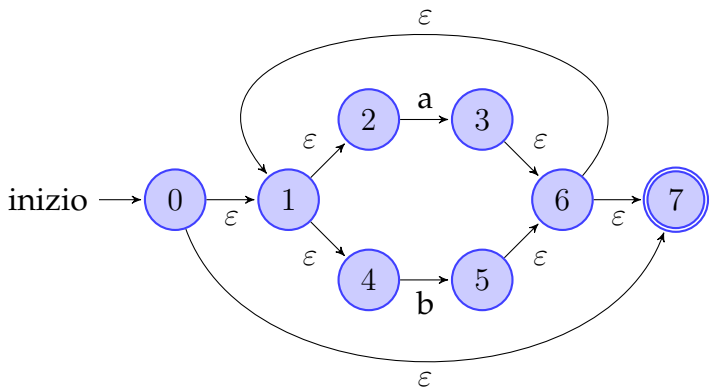


Figura 4.9: Applicazione della costruzione per la Kleene star

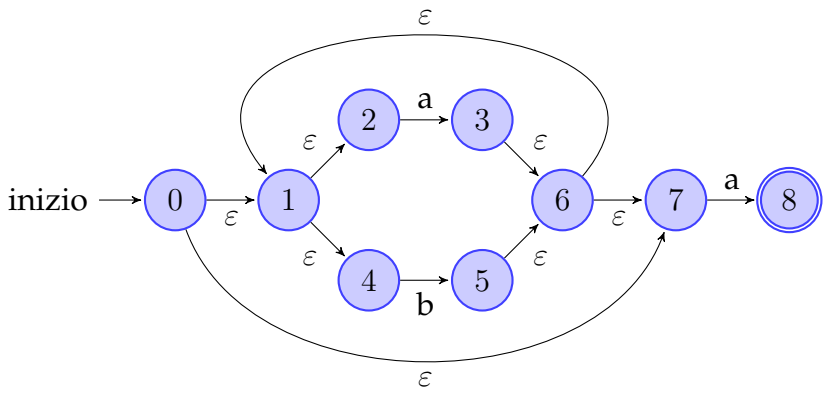


Figura 4.10: Applicazione della costruzione per la concatenazione

Applichiamo quindi la costruzione di Thompson per la concatenazione.
Dobbiamo infine riapplicare il passo della concatenazione per le due b mancanti.

Possiamo notare come il risultato cui giungiamo sia pieno di ε; la costruzione di Thompson ci dà sicuramente un risultato esatto, ma non è detto che sia il risultato migliore, anzi; abbiamo già visto in passato un automa che riconosce lo stesso linguaggio di questo, e quell'automa era più semplice ed elegante (vedi 4.3).

Notiamo che comunque l'algoritmo di Thompson garantisce dei limiti superiori sulla complessità dell'automa che genera: ad ogni passo si aggiungono al massimo 2 nuovi nodi. La lunghezza dell'automa finale infatti è limitata da $2n$, con n che rappresenta il numero di simboli nell'espressione regolare, dove con simboli si intende sia simboli del vocabolario che operatori.

Anche il numero di archi è limitato in quanto per ogni passaggio si aggiungono

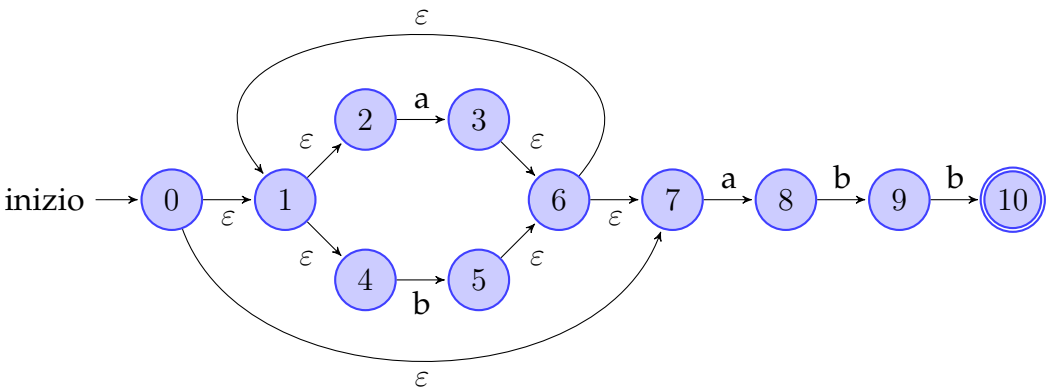


Figura 4.11: Applicazione della costruzione per la concatenazione (II)

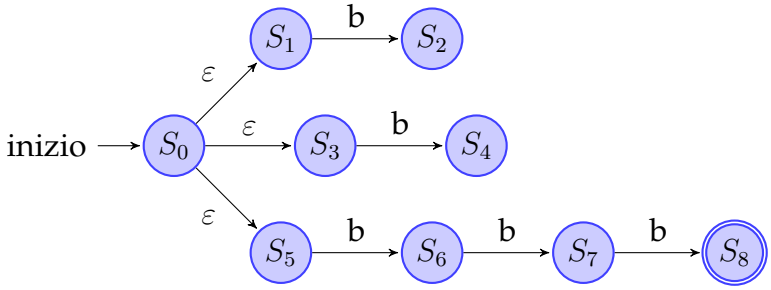


Figura 4.12: Esempio di automa

al massimo 4 archi.

4.6 Il processo di simulazione dell'automa

Le nostre conoscenze attuali ci permettono di costruire un automa per una certa espressione regolare, ma come si fa a decidere se una certa parola w fa parte del linguaggio di un dato automa $\mathcal{N}$? Si utilizza una procedura detta *simulazione dell'automa*.

Abbiamo già definito in 4.4.1 come viene deciso se una parola fa parte del linguaggio riconosciuto dall'automa; prendiamo ora ad esempio $w = bbb$ e l'automa presentato in figura 4.12, e presentiamo la procedura di simualzione di un automa.

Se dovessimo fare "a occhio", questo caso sarebbe molto semplice da risolvere: si parte dallo stato S_0 e si va in un altro stato tramite o archi ε o tramite archi b; una volta che raggiungo uno stato con un arco b elimino il primo carattere della parola e procedo con il secondo, cerco archi con ε o con il secondo carattere e così

via. La parola appartiene al linguaggio riconosciuto dall'automa se consumo l'ultimo carattere della parola in uno stato finale.

Noi però stiamo cercando un algoritmo formale che faccia queste operazioni, e che le faccia senza necessità di backtrack per tornare indietro se un percorso si rivela errato, operazione altamente costosa. Cosa possiamo fare per velocizzare il processo?

Vediamo che leggere bbb da S_0 è uguale a leggerlo da uno qualunque tra gli stati $\{S_0, S_1, S_3, S_5\}$, perché arrivo a questi stati solo tramite archi ε; non potremmo quindi semplificare tutti questi in un solo nodo?

La risposta è sì, possiamo farlo! Per vedere in che modo dobbiamo prima introdurre il concetto di ε-chiusura.

4.6.1 ε-chiusura di un automa

Sia $(S, \mathcal{A}, \text{move}_n, S_0, F)$ un NFA, sia t uno stato in S e T un sottoinsieme di S.

Definiamo ε-chiusura$(\{t\})$ l'insieme di stati in S che sono raggiungibili da t tramite zero o più ε-transizioni (nota che t è sempre in questo insieme).

Definiamo ε-chiusura(T) nel seguente modo:

$$\varepsilon\text{-chiusura}(T) = \bigcup_{t \in T} \varepsilon\text{-chiusura}(\{t\}) \tag{4.6}$$

Calcolo della ε-chiusura

Per calcolare la ε-chiusura di un nodo di un automa useremo queste strutture:

1. uno stack;
2. un array booleano `alreadyOn` che ci serve per segnalare se uno stato t è già sulla pila o meno (la sua dimensione è $|S|$);
3. un array bidimensionale per ricordare move_n. Ogni entry (t, x) è una lista linkata contenente tutti gli stati che sono raggiungibili con un x-transizione da t.

Ora che sappiamo quali sono le strutture dati che andremo ad utilizzare, studiamo la fase di inizializzazione:

- all'inizio dei tempi non c'è niente sulla pila, poi inseriamo t e lo segnaliamo su `alreadyOn`;
- a questo punto estraiamo dalla cima dello stack, prendiamo il nodo estratto e e cerchiamo tutti i nodi che da e si raggiugono tramite un ε-arco; inseriamo tutti questi nello stack (se non sono già presenti) e ne segnaliamo l'inserimento settando il loro flag su `alreadyOn`.

Continuiamo così finché lo stack non si svuota. L'algoritmo appena descritto si può trovare in Alg.2.

Algorithm 1: Wrapper per il calcolo della ε-chiusura

```
1 Stack S = Stack()
2 foreach i = 1 to |S| do
3     alreadyOn[i] = false
4 ε-chiusura(t, S)
```

Algorithm 2: ε-chiusura(State $\{t\}$, Stack S)

```
  // Assumiamo la struttura move_n globale
1 S.push(t)
2 alreadyOn[t] = true
3 foreach u ∈ move_n(t, ε) do
4     if not alreadyOn[u] then
5         ε-chiusura(u, S)
```

Ora che abbiamo l'arma della ε-chiusura possiamo procedere con l'algoritmo per la verifica dell'appartenenza di una parola w al linguaggio riconosciuto da un automa, ovvero l'algoritmo di simulazione (Alg.5).

Algorithm 3: boolean SimulazioneNFA(NFA $\mathcal{N}$, Word w)

```
1 states = ε-chiusura({s_0})
2 symbol = nextChar()
3 while symbol ≠ $ do
4     states = ε-chiusura(⋃_{t∈states} move_n(t, symbol))
5     symbol = nextChar()
6 if states ∩ F ≠ ∅ then
7     return true
8 else
9     return false
```

Dove $ è l'end-marker per la parola w (alcuni testi usano la "gratella" invece che $). Il funzionamento dell'algoritmo è questo: calcoliamo la ε-chiusura di S_0 e la aggiungiamo all'insieme `states`; prendiamo il primo simbolo nella parola tramite $nextChar()$ e poi entriamo nel ciclo che compone il corpo della procedura.

In questo corpo si utilizza la ε-chiusura per salvare in `states` la ε-chiusura di tutti gli stati che posso raggiungere tramite una `symbol`-transizione (una transizione marcata con `symbol`); una volta trovati questi nodi, il `symbol` prende il valore del prossimo carattere di w.

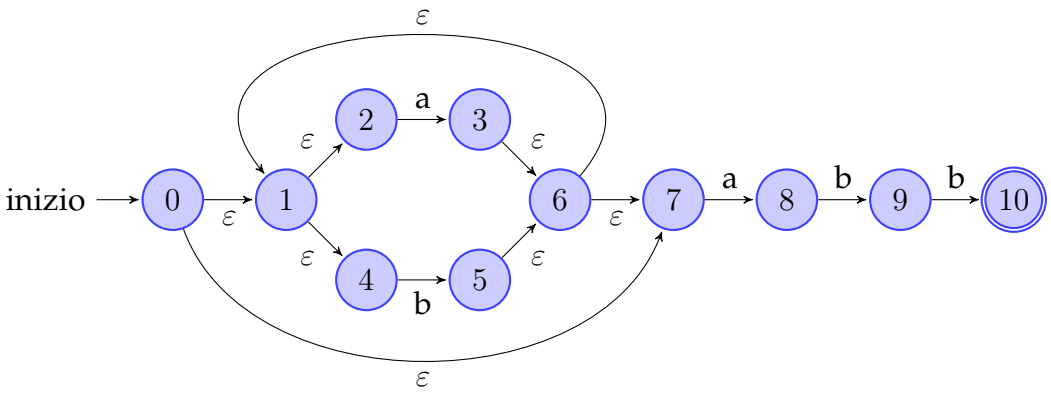


Figura 4.13: Esempio di simulazione

Una volta che `symbol` prende il valore di `$`, mi fermo e controllo il contenuto di `states`.

Se `states` $\cap$ $F \neq \emptyset$, allora significa che esiste uno stato in `states` che è anche uno stato finale dell'automa e quindi abbiamo trovato un percorso che inizia da s_0 riconosce la parola w e termina in uno stato finale (ammissibile). Ottimo, abbiamo appena dimostrato che w appartiene al linguaggio riconosciuto dall'automa.

Se invece la condizione precedente non è verificata, allora si è avverato uno dei seguenti casi (o anche entrambi): o non esistono percorsi che riconoscono w, o non esistono percorsi che riconoscono w e terminano in uno stato finale.

4.6.2 Esempi di simulazione

Applichiamo ora, come esempio, l'algoritmo di simulazione di un NFA dato (Fig.4.13), verificando se accetta la parola $w = ababb$. Lo svolgimento è rappresentato in Tab.4.2.

`states`	`symbol`	$\bigcup_t \text{move}_n(t, \texttt{symbol})$	ε-chiusura
$T0 = \{0, 1, 2, 4, 7\}$	a	$\{3, 8\}$	$\{1, 2, 3, 4, 6, 7, 8\}$
$T1 = \{1, 2, 3, 4, 6, 7, 8\}$	b	$\{5, 9\}$	$\{1, 2, 4, 5, 6, 7, 9\}$
$T2 = \{1, 2, 4, 5, 6, 7, 9\}$	a	$\{3, 8\}$	$T1$
$T1$	b		$T2$
$T2$	b	$\{5, 10\}$	$\{1, 2, 4, 5, 6, 7, 10\}$
$T3 = \{1, 2, 4, 5, 6, 7, 10\}$	`$`		

Tabella 4.2: Tabella risolutiva della simulazione sull'automa 4.13

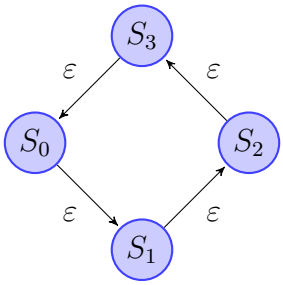


Figura 4.14: Se non scegliessimo il più piccolo X potremmo incorrere in cicli infiniti.

In seguito una breve descrizione della rappresentazione dell'algoritmo.

Inizializziamo la variabile `states` a $T0$ inserendo la ε-chiusura di 0; poi estraiamo il simbolo da aggiungere (il primo simbolo di w) che è a.

Ora devo comporre l'insieme di stati che posso raggiungere da `states` con una a-transizione, e questo insieme è $\{3, 8\}$.

Di questo insieme l'algoritmo dice che devo calcolare la ε-chiusura, che corrisponde all'insieme nell'ultima colonna della prima riga; questo è il mio nuovo insieme di stati $T1$ da cui partirò nello step successivo.

Nel secondo step il simbolo che devo aggiungere è la seconda lettera di w, ovvero b, quindi vado a cercare quei nodi che posso raggiungere da $T1$ con una b-transizione; questi nodi sono $\{5, 9\}$, e una volta che li ho trovati ne calcolo la ε-chiusura, che compone $T2$ e via così.

Continuo così finché non finisco i simboli; a quel punto devo verificare se esiste nel mio insieme $states$ uno stato finale. In questo caso è presente (10), e quindi la parola w appartiene al linguaggio riconosciuto dall'automa raffigurato.

4.6.3 Nota sulla ε-chiusura

Teorema 4.6.1. *Sia $N = (S, \mathcal{A}, move_n, S_0, F)$ un NFA e sia $M \subseteq S$.*

Allora la ε-chiusura(M) è il più piccolo insieme $X \subseteq S$ tale che X è una soluzione alla seguente equazione:

$$X = M \cup \{N' \mid N \in X \cap N' \in move_n(N, \varepsilon)\} \tag{4.7}$$

Nota che diciamo il più piccolo insieme per evitare di proseguire in loop infiniti come quello rappresentato in figura 4.14.

Osserviamo con attenzione la formula appena descritta: X è M stesso, unito anche a tutti gli stati N' raggiungibili da N tramite una ε-transizione, dove N è uno stato qualsiasi in M.

Balza subito all'occhio come la formula per definire X dipenda da X stessa; per questo motivo non dovremmo poterla calcolare, giusto? Sbagliato! In informatica possiamo risolvere queste equazioni date certe caratteristiche, ed è qui che giunge in nostro aiuto il teorema del punto fisso.

4.6.4 Teorema del punto fisso

L'equazione 4.7 vista poco fa è un'istanza di una più generale equazione su insiemi, della forma $X = f(X)$, sulla quale possediamo un risultato importante.

Sia $f : 2^D \to 2^D$ per qualche insieme finito D e sia inoltre f monotona, vale a dire

$$X \subseteq Y \implies f(X) \subseteq f(Y)$$

allora esiste una precisa tecnica, basata su approssimazioni successive, per risolvere l'equazione $X = f(X)$. Adesso vediamo meglio il teorema cui fa riferimento.

Teorema 4.6.2. *Sia S un insieme finito e sia $f : 2^S \to 2^S$ una funzione monotona; allora $\exists m \in \mathbb{N}$ tale che esiste un'unica soluzione minima all'equazione $X = f(X)$, e questa soluzione è $f^m(\varnothing)$.*

Dimostrazione. I due asserti andranno dimostrati separatamente; come prima cosa, vogliamo dimostrare che $\exists m \in \mathbb{N}$ tale che $f^m(\varnothing)$ è soluzione per $X = f(X)$; prima di procedere è inoltre consigliabile avere ben presente la forma della funzione $f(X)$ si veda 4.7.

Per definizione stessa di $f(X)$, possiamo subito dire che $\varnothing \subseteq f(\varnothing)$ e, poiché $f(X)$ è monotona, possiamo subito dire anche che $\varnothing \subseteq f^2(\varnothing)$. Quindi, per il principio di induzione, possiamo affermare senza timore di smentita che $f^i(\varnothing) \subseteq f^{i+1}(\varnothing), \forall i \in \mathbb{N}$. A questo punto, possiamo quindi dire di avere una sorta di catena di relazioni tra insiemi:

$$f(\varnothing) \subseteq f^2\varnothing \subseteq f^3(\varnothing) \subseteq \dots$$

Questa successione è infinita, ma 2^S è definito come finito, per cui gli insiemi della successione non possono essere tutti diversi l'uno dall'altro! Arriveremo infatti, a un certo punto, a non riuscire più a osservare cambiamenti a successive applicazioni di $f(X)$; formalmente, troveremo un qualche indice m tale che:

$$f^m(\varnothing) = f^{m+1}(\varnothing) = f(f^m(\varnothing))$$

Si dice che siamo arrivate al punto di *saturazione*; in questa situazione, possiamo dire che $f^m(\varnothing)$ è una soluzione per $X = f(X)$.

Andiamo adesso a dimostrare che $f^m(\varnothing)$ è l'unica soluzione minima.
Per assurdo, supponiamo che esista un'altra soluzione A per $X = f(X)$; quindi, per ipotesi, deve valere $A = f(A)$, e quindi dovremo avere che $A = f(A) = f^2(A) = \ldots = f^m(A)$. Sappiamo anche che $f^m(\varnothing) \subseteq f^m(A)$, poiché naturalmente l'insieme vuoto è compreso in qualsiasi insieme ($\varnothing \subseteq A$) e la funzione f è monotona. Ma allora possiamo mettere assieme le due precedenti osservazioni e affermare quindi che $f^m(\varnothing) \subseteq A$, poiché $f^m(\varnothing) \subseteq f^m(A)$ e $A = f^m(A)$. Per cui $f^m(\varnothing)$ è una soluzione unica, ed è anche l'unica minima.

QED

Nonostante a prima vista il significato o anche la stessa utilità di questo risultato possa non essere del tutto chiaro, vale la pena di vederlo almeno una volta, poiché tutta la teoria che riguarda la semantica dei linguaggi di programmazione è basata su questo teorema.

Ad esempio, l'esecuzione di una funzione fattoriale: essa è costituita di una sequenza di costrutti `while` tali da produrre il risultato per l'input desiderato; ogni iterazione del `while` calcola un'approssimante del risultato, che viene quindi ricostruito piano piano. È chiaro che in questo esempio non stiamo aggiungendo insiemi ma valori interi, però è utile pensarlo per avere quantomeno un'idea intuitiva di come il teorema del punto fisso sia applicato alla teoria della semantica dei linguaggi di programmazione.

4.7 Considerazioni sull'efficienza degli algoritmi sugli NFA

All'inizio della trattazione sugli automi non deterministici, il nostro obiettivo era riconoscere un linguaggio regolare attraverso degli automi a stati finiti; in particolare, abbiamo detto che l'analisi lessicale utilizzerà le espressioni regolari per identificare i vari elementi del programma scritto. Dalle espressioni regolari vogliamo quindi avere degli strumenti che ci permettono di prendere come input il testo del programma e ottenere in output un flusso di token, che saranno i terminali della grammatica che ha generato il nostro linguaggio.

Ci chiediamo quale sia la complessità degli algoritmi che abbiamo visto per risolvere questa richiesta, anche in vista del momento in cui ci troveremo a scegliere quale tipo di automa scegliere (NFA o DFA) per assolvere diverse tipologie di compiti.

Sunto delle procedure viste Data una parola e data un'espressione regolare che denota quello stesso linguaggio, si dica se la parola appartiene a quel particolare

linguaggio oppure no. Gli algoritmi che abbiamo visto ci consento di rispondere a questa richiesta applicando le seguenti operazioni:

- consideriamo un'espressione regolare r;
- applichiamo la costruzione di Thompson e generiamo un NFA che riconosce esattamente il linguaggio denotato da r;
- lancia l'algoritmo di simulazione per l'NFA generato.

Andiamo quindi ad analizzare la complessità di queste procedure.

4.7.1 Complessità della costruzione di Thompson

Consideriamo la complessità di generazione di un NFA con n nodi e m archi; conoscendo le quattro operazioni, sappiamo che, per ogni passo, aggiungeremo al massimo 2 nodi e 4 archi (dalla Kleene star, l'operazione più costosa), per cui avremo che $n \leq 2|r|$ e $m \leq 4|r|$, per cui abbiamo che la somma $n+m$ è dell'ordine della dimensione dell'espressione regolare ($\mathcal{O}(|r|)$) e, poiché avremo in totale $|r|$ passi, ciascuno dei quali eseguibile in tempo costante, possiamo felicemente concludere che $T(\text{Thompson}(r)) = \mathcal{O}(|r|)$.

4.7.2 Complessità del calcolo della ε-chiusura

Prima di analizzare il costo della simulazione, dobbiamo analizzare il costo della ε-chiusura. Tenendo sempre a mente il codice (Alg.1 e Alg.2), evidenziamo che le strutture usate sono:

- uno stack per contenere gli elementi non ancora incontrati;
- un vettore booleano per tenere traccia dei visitati;
- move$_n$, al solito un vettore bidimensionale di puntatori a liste linkate, ciascuna contenente tutti gli stati raggiungibili a partire da uno stato t tramite un simbolo x.

In sunto, la procedura si compone dei seguenti quattro passi:

1. inserimento del nodo t nello stack;
2. imposta `alreadyOn[t] = true`;
3. trova un successivo nodo $u \in \text{move}_n(t, \varepsilon)$;
4. verifica se è già stato visitato con `alreadyOn[u]`.

Ognuna di queste operazioni opera in tempo costante, e ci chiediamo quindi quante volte venga ripetuto ognuna di queste.

Osserviamo che le prime due operazioni vengono ripetute a ogni chiamata della procedura, e mai più di una volta per ogni nodo, dal momento che il vettore `alreadyOn` non ci permetterà di richiamare la funzione se lanciato su un nodo già settato a `true`, e non capita mai che qualche bit venga reinizializzato a `false`. Per cui, il costo delle prime due operazioni è $\mathcal{O}(n)$.

La terza e la quarta oeprazione vengono eseguite per ogni nodo raggiungibile con una ε-transizione, per cui possiamo immaginare un caso pessimo in cui ogni stato possiede almeno una ε-transizione; nel caso peggiore, quindi, andremo a visitare ogni arco del grafo ($\mathcal{O}(m)$).

Complessivamente, il costo della procedura è $\mathcal{O}(n + m)$.

4.7.3 Complessità della simulazione di NFA

Adesso possiamo affrontare il problema dello studio della complessità della simulazione di NFA. Anche qui, teniamo presente il codice:

Algorithm 4: boolean SimulazioneNFA(NFA $\mathcal{N}$, WORD w)

```
1 states = ε-chiusura({s_0})
2 symbol = nextChar()
3 while symbol ≠ $ do
4 |   states = ε-chiusura(⋃_{t∈states} move_n(t, symbol))
5 |   symbol = nextChar()
6 if states ∩ F ≠ ∅ then
7 |   return true
8 else
9 |   return false
```

Anche qui, un veloce ripasso delle strutture dati utilizzate. Innanzitutto si tenga presente che, per tenere traccia della variabile `states`, utilizzeremo ben due stacks:

- il primo (`currentStack`) è nel right value della riga 4 e viene utilizzato per conservare il contenuto del vettore `states` durante l'attuale iterazione dell'algoritmo;
- il secondo (`nextStack`) è invece nel left value e verrà aggiornato durante la attuale iterazione.

Abbiamo anche il nostro vettore `alreadyOn` di dimensione $|S|$ e, naturalmente, il vettore bidimensionale per conservare move$_n$.

Ci rendiamo conto che il ciclo `while` domina la complessità, soprattutto perché al suo interno avviene una chiamata di ε-chiusura, completata dall'estrazione di tutti gli elementi da `nextStack` e il loro inserimento in `currentStack`, oltre alla reinizializzazione di `alreadyOn`.

Algorithm 5: Dettaglio di `simulazioneNFA(NFA` $\mathcal{N}$`, WORD` w`)` (riga 4)

```
1 foreach t ∈ currentStack do
2     foreach u ∈ move_n(t, symbol) do
3         if not alreadyOn[u] then
4             ε-chiusura(u, nextStack)
5     currentStack.pop(t)
6 foreach s ∈ nextStack do
7     nextStack.pop(s)
8     currentStack.push(s)
9     alreadyOn[s] = false
```

All'interno di ogni `while`, quindi, abbiamo uno swap degli stack, che costa $\mathcal{O}(n)$, poiché è limitato dal numero di nodi dell'NFA; inoltre, ogni stato può essere inserito solo una volta nella pila nel secondo `foreach`; quindi, ogni ciclo del `while` costa $\mathcal{O}(n+m)$. Questo ciclo verrà lanciato per ogni elemento della parola w analizzate, pertanto il costo complessivo della simulazione è $\mathcal{O}(|w|(n+m))$; dal momento che il nostro NFA è stato generato da un'esecuzione di costruzione di Thompson, avremo anche che $n+m = \mathcal{O}(|r|)$, per cui possiamo concludere che il costo dell'intera procedura di simulazione è dell'ordine di $\mathcal{O}(|w||r|)$.

Il costo è sicuramente più basso di quello che avrei facendo del backtrack, dal momento che si parla comunque di un modello non deterministico.

4.8 Automa a stati finiti deterministico

Non c'è modo migliore di introdurre questi automi se non per differenza rispetto agli NFA. Si tenga a mente la definizione di NFA:

$$\mathcal{N} := (S, \mathcal{A}, \text{move}_n, S_0, F), \text{ dove}$$
$$\text{move}_n : S \times (\mathcal{A} \cup \{\varepsilon\}) \to 2^S$$

Si noti la funzione di transizione, che prende come input uno stato e un simbolo da $\mathcal{A} \cup \varepsilon$, dove è la presenza stessa di ε a causare indeterminatezza; inoltre, un qualunque nodo di un grafo ha tendenzialmente più archi uscenti identificati dal medesimo simbolo.

Un automa a stati finiti deterministici è invece definito come segue:

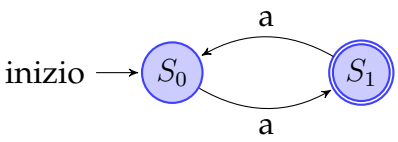
$$\mathcal{D} := (S, \mathcal{A}, \text{move}_d, S_0, F), \text{ dove} \tag{4.8}$$

$$\text{move}_d : S \times \mathcal{A} \to S \tag{4.9}$$

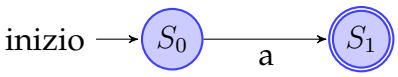
Balza subito la differente formulazione della funzione di transizione $move_d$, il cui dominio esclude la possibilità di compiere ε-transizioni. Inoltre, quest'ultima ha due modi di presentarsi: totale o parziale, e il possesso dell'una o dell'altra qualificazione inficerà la scelta delle procedure da applicare.

Evidenziamo le caratteristiche dei DFA:

- non presentano ε-transizioni (ma c'è un cavillo di cui ci occuperemo più avanti);
- se move$_d$ è *totale*, allora per ogni stato c'è **esattamente** una a-transizione $\forall a \in \mathcal{A}$;
- se move$_d$ è *parziale*, allora per ogni stato c'è **al massimo** una a-transizione $\forall a \in \mathcal{A}$; in altre parole, per alcune coppie del dominio ($(s, a) \in S \times A$) la funzione non è definita.



(a) Esempio di DFA con funzione di transizione totale



(b) Esempio di DFA con funzione di transizione parziale

Figura 4.15: Funzioni di transizione dei DFA

4.8.1 Linguaggi riconosciuti dai DFA

Prendiamoci un certo DFA $\mathcal{D} = (S, \mathcal{A}, \text{move}_d, S_0, F)$; il linguaggio $\mathcal{L}(\mathcal{D})$ da lui riconosciuto è l'insieme di parole w tale che:

- o esiste un cammino, che chiamiamo qui $w = a_1, \ldots, a_k$, con $k \geq 1$, che vada dallo stato iniziale S_0 a un qualche stato finale in F;
- oppure vale che $S_0 \in F \cap w = \varepsilon$; per l'appunto, quello in cui lo stato di partenza è uno stato finale è l'unico caso in cui un DFA riconosce la parola vuota.

4.8.2 Simulazione di un DFA con transizione totale

In questo caso abbiamo la certezza che, per ogni simbolo che leggiamo, ci sia un qualche arco che lo colleghi a uno stato nel grafo, e questo semplifica di molto la procedura per determinare se una certa parola w appartenga o meno al linguaggio del nostro DFA.

Per l'appunto, è sufficiente partire dallo stato iniziale S_0 e seguire il cammino dato dagli elementi di w; se terminiamo in un qualche stato finale in F, allora w appartiene al linguaggio, altrimenti no; la complessità, pertanto, è $\mathcal{O}(|w|)$.

4.8.3 Simulazione di un DFA con transizione parziale

Poiché qui abbiamo una funzione di transizione parziale, dobbiamo considerare la possibilità di arrivare a uno stato in cui non non possiamo più proseguire.

Iniziamo comunque dallo stato iniziale e, di nuovo, seguiamo il cammino dato dallo spelling della parola $w = a_1, a_2, \ldots, a_k$; se stiamo leggendo un qualche simbolo per cui non c'è una transizione "etichettata" da quel simbolo, allora possiamo direttamente ritornare una risposta negativa; se invece riusciamo a leggere tutta la parola e a raggiungere uno stato finale, possiamo dire che w appartiene al linguaggio.

4.8.4 Funzioni di transizioni a confronto

Dato un DFA $\mathcal{D}$ con funzione di transizione parziale, ho un modo per definire un DFA $\mathcal{D}'$ che abbia funzione di transizione totale e che riconosca esattamente lo stesso linguaggio ($\mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathcal{D}')$)?

Sì, posso farlo con l'uso di un *sink* (anche detto dead state, o in italiano stato trappola). In sostanza vado a creare un nuovo stato non finale che aggiungo agli altri stati di $\mathcal{D}$, e questo stato sarà la destinazione di tutte le transizioni non definite dalla funzione di transizione; infine, per ogni simbolo dell'alfabeto, aggiungo al sink un *self loop* su quel simbolo.

In questo modo, quando vado a trovarmi in quelle situazioni in cui non potevo più proseguire, posso invece proseguire e andare nel sink; in questo modo potrò comunque continuare, esaurire la parola in uno stato non terminale e quindi tornare un risultato coerente al linguaggio considerato.

In generale si preferisce lavorare con DFA con funzione di transizione parziale, e ancora più in generale con automi che abbiano il minor numero di stati (nodi e archi) possibile; tuttavia, l'algoritmo di minimizzazione (vedremo più avanti), che usiamo per "ridurre al minimo" il numero di archi e nodi, non funziona con DFA dotati di funzione di transizione parziale. Il motivo sta nel fatto che questa

procedura lavora sulla funzione inversa di move_d, che chiaramente non è definita se la funzione non è totale.

Capitolo 5

Automi a stati finiti: teoria, esercizi, algoritmi e simili amenità

5.1 Traduzione da NFA a DFA

Abbiamo già discusso in passato di come la simulazione su DFA sia più efficiente che su NFA, quindi ora siamo interessati a risolvere questo problema: dato un qualunque NFA come si può ricavare un DFA che riconosca lo stesso linguaggio?

Un mezzo per raggiungere questo scopo è l'utilizzo di una sorta di meccanismo di ε-chiusura permanente (che elimini in modo permanente le ε-transizioni), questo meccanismo si chiama Subset Construction.

5.1.1 Algoritmo di Subset Costruction

L'algoritmo di Subset Construction serve per ricavare un DFA dato un certo NFA.

L'idea di base è quella di utilizzare la ε-chiusura per mappare sottoinsiemi degli stati del dato NFA (che in seguito chiameremo rozzamente *stati non deterministici*, o *snd*) in stati per il nuovo DFA (che in seguito chiameremo ancor più rozzamente *stati deterministici* o *sd*).

Per dare un'idea di come lavora questo algoritmo descriviamone i primi passi. Si individua qual è lo stato iniziale dell'NFA, diciamo S_0, lo si prende come punto di partenza e se ne calcola la ε-chiusura T; sarà proprio T ad essere lo stato iniziale del DFA. Ora dovrebbe essere più chiaro quello che si fa in questo algoritmo: si eliminano tutte le ε-transizioni per ottenere un set di transizioni move_d che contiene solo transizioni "deterministiche".

Un fatto da tenere sempre ben presente è che gli stati del DFA che otteniamo in questo modo sono insiemi di stati dell'NFA di partenza. Il T appena descritto è un sottoinsieme degli stati dell'NFA di partenza.

Dopo questa piccola introduzione passiamo a visualizzare subito il codice dell'algoritmo di Subset Construction.

Algorithm 6: DFA subsetConstruction(NFA $\mathcal{N}$)

```
1  T_0 = ε-chiusura({S_0})
2  R = T_0
3  while qualche T ∈ R è unmarked do
4      mark(T)
5      foreach a ∈ A do
6          T' = ε-chiusura(⋃_{t∈states} move_d(t, a))
7          if T' ≠ ∅ then
8              move_d(T, a) = T'
9              if T' ∉ R then
10                 add T' to R
11                 unmark(T')
12 foreach T ∈ R do
13     if (T ∩ F) ≠ ∅ then
14         set T ∈ E
```

Procediamo quindi ora con una spiegazione discorsiva di questo algoritmo.

Input L'algoritmo prende come input un NFA specificato dalla tupla $\mathcal{N} := (S, \mathcal{A}, \text{move}_n, S_0, F)$

Output L'algoritmo ritorna come output un DFA specificato dalla tupla $\mathcal{D} := (R, \mathcal{A}, \text{move}_d, T_0, E)$ tale da soddisfare la seguente condizione: $\mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathcal{N})$.

Introduzione Per prima cosa si calcola quello che sarà lo stato iniziale del DFA, ovvero T_0; esso coinciderà, come già detto in precedenza, con la ε-chiusura dello stato iniziale dell'NFA S_0. Una volta calcolato questo T_0, lo si inserisce nell'insieme R e lo si segna come `unmarked`.

Ciclo while Qui inizia il corpo centrale dell'algoritmo: si va ad iterare su tutti gli stati che si stanno inserendo mano a mano nell'insieme R degli stati del DFA.

Di fatto, dentro al ciclo si scorrono tutti gli stati in R e ci si sofferma sugli stati $T \in R$ segnati come `unmarked`. Di volta in volta, quando se ne identifica uno, lo si marca, dimodoché non venga più elaborato in seguito, quindi si procede con le successive istruzioni.

Foreach interno al ciclo while In questa fase si scorrono tutti gli elementi a dell'alfabeto $\mathcal{A}$ e si prosegue quindi in questo modo:

- si prendono tutti gli stati (snd) raggiungibili dallo stato (sd) T che soddisfino la seguente condizione:
$$\cup_{t \in T} \text{move}_n(t, a)$$
ovvero tutti gli stati in S raggiungibili tramite una a-transizione da uno degli stati in T, se ne calcola la ε-chiusura e la si salva nella variabile T';
- una volta calcolato T' si verifica se questo è vuoto, e se lo è si passa alla prossima iterazione del `foreach`;
- altrimenti, si aggiunge all'insieme delle transizioni ("deterministiche") move_d la transizione $\text{move}_d(T, a) = T'$;
- infine, se T' non è già presente in R allora ve lo si aggiunge e si segna come `unmarked`.

Foreach finale Questa fase serve per definire quali sono gli stati (sd) finali del nuovo insieme di stati R, ovvero quali sono gli stati finali del DFA $\mathcal{D}$. Questi stati (ribadiamo, sd) si identificano perché contengono *almeno uno* stato (snd) finale, ovvero uno stato $T \in R$ è uno stato finale per $\mathcal{D}$ se $T \cap F \neq \emptyset$.

Arriva però il momento di porsi un'importante domanda: qual è la complessità di questo algoritmo? Poniamo le basi per affrontare una risposta: supponiamo che l'NFA abbia n stati ed m archi, mentre supponiamo che il DFA abbia n_d stati.

- Osserviamo che il ciclo dominante naturalmente è il primo `while`; quest'ultimo è ritpetuto n_d volte, ma quanto vale n_d? Questo è un aspetto che affronteremo nel prossimo futuro.
- All'interno del `while` troviamo il nostro caro `foreach` $a \in \mathcal{A}$, che viene ripetuto $|A|$ volte.
- L'ultimo fattore di peso rilevante è la chiamata alla funzione di calcolo della ε-chiusura, la quale ha costo $\mathcal{O}(n + m)$.

La complessità globale è quindi $\mathcal{O}(n_d \cdot |A| \cdot (n + m))$.

Ora la domanda si fa più pressante: quanto vale effettivamente n_d? Fin massa... possiamo già anticipare che esso è un $\mathcal{O}(2^n)$. Nonostante questo possa essere oltremodo sconfortante, è opportuno specificare che quella appena vista è la complessità per *creare* il DFA, un'operazione che viene effettuata una tantum solamente in sede di creazione del compilatore; è vero, è una grossa spesa iniziale, ma permette di avere grandi vantaggi in tutti gli utilizzi successivi.

Tuttavia per applicazioni con tempo di vita più limitato spesso non ha senso creare il DFA, poiché non si avrebbe il tempo di compensare la spesa iniziale; è questa la ragione per cui ad esempio grep utilizza NFA.

5.1.2 Esercizi di applicazione della Subset Construction

Esercizio 1

Dato l'automa a stati finiti non deterministico in figura 5.1, calcola, tramite l'algoritmo di Subset Construction, un automa a stati finiti deterministico che riconosca lo stesso linguaggio.

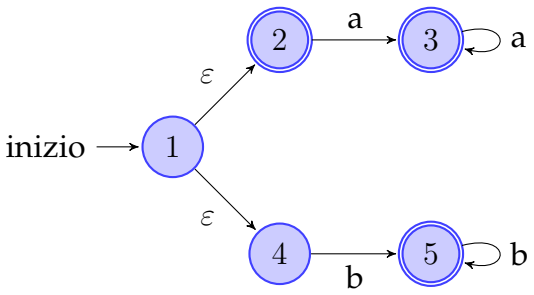


Figura 5.1: NFA dato

Il modo più semplice e chiaro per risolvere questo tipo di esercizi è quello di utilizzare una tabella per scrivere i vari passaggi; noi riportiamo qui sotto la soluzione dell'esercizio in tale forma, seguita da un esteso commento della procedura seguita.

Stati (deterministici)	ε-chiusura dei punti di arrivo delle a-transizioni	ε-chiusura dei punti di arrivo delle b-transizioni
$T0 = \varepsilon - cl(\{1\}) = \{1,2,4\}$ (stato iniziale) (2 $\implies$ stato finale)	$T1 = \varepsilon - cl(\{3\}) = \{3\}$ [T1 unmarked]	$T2 = \varepsilon - cl(\{5\}) = \{5\}$ [T2 unmarked]
$T1 = \{3\}$ (stato finale perché c'è 3)	$T3 = \varepsilon - cl(\{3\}) = \{3\} = T1$ T1 già nella collezione, non lo rimetto	$\emptyset$
$T2 = \{5\}$ (stato finale perché c'è 5)	$\emptyset$	$T4 = \varepsilon - cl(\{5\}) = \{5\} = T2$ T1 già nella collezione, non lo rimetto

Tabella 5.1: Soluzione esercizio 1

Lo stato iniziale dell'NFA è 1, quindi per calcolare lo iniziale del DFA devo calcolare la ε-chiusura di 1; questo passaggio è riportato a riga 1, colonna 0.

Poi entro nel vivo del `while`: estraggo $T0$ e devo trovare tutti gli stati raggiungibili da $T0$ tramite una a-transizione. In questo modo ottengo lo stato 3; di questo insieme di stati ($\{3\}$) devo calcolare la ε-chiusura, e questa sarà il nuovo insieme $T1$; aggiungo la transizione $\text{move}_d(T0, a) = \{T1\}$, segno $T1$ come `unmarked` e passo avanti.

Adesso devo trovare tutti gli stati raggiungibili da $T0$ attraverso una b-transizione. In questo caso ottengo solo lo stato 5, e di questo insieme ($\{5\}$) calcolo la ε-chiusura; questa risulta essere $\{5\}$. Ho appena trovato $T2$, lo segno come `unmarked` e aggiungo la transizione $\text{move}_d(T0, b) = \{T2\}$.

Passo infine ad analizzare lo stato $T1$. Estraggo $T1$ da R e vado a cercare la ε-chiusura dei punti di arrivo delle a-transizioni che partono da $T1$; trovo che l'unica a-transizione che, partendo da $T1$, mi porta allo stato 3, e la ε-chiusura di $\{3\}$ mi ritorna esattamente $T1$. Quindi, quello che sto facendo altro non è che aggiungere la transizione $\text{move}_d(T1, a) = \{T1\}$ al nostro DFA, ma dato che $T1$ è già `marked` non lo riaggiungo ad R e termino qui la mia analisi.

Passo ora alla ε-chiusura dei punti di arrivo delle b-transizioni. Da $T1$ non parte nessuna b-transizione, quindi termino qui l'analisi di $T1$ e passo a $T2$. Per $T2$ compio banalmente le stesse azioni che ho svolto per $T1$, che quindi non sono qui riportate.

Alla fine della fiera, devo trovare quali sono gli stati finali del nostro automa deterministico, quindi faccio le varie intersezioni (come descritto dall'algoritmo) e segno come stati (deterministici) finali tutti quegli stati (deterministici) che

contengono almeno uno stato (non deterministico) che sia esso stesso finale nell'NFA.

Una volta terminata questa operazione abbiamo trovato quindi l'automa a stati finiti deterministico che riconosce lo stesso linguaggio dell'NFA di partenza; possiamo vederne una rappresentazione in figura 5.2.

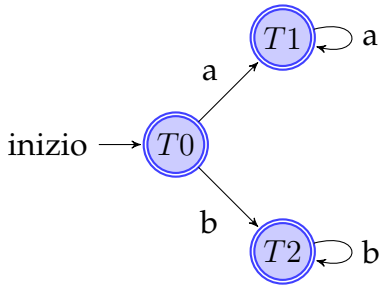


Figura 5.2: DFA ricavato tramite subset construction

Esercizio 2

Dato l'automa a stati finiti non deterministico in figura 5.3, calcola tramite l'algoritmo di Subset Construction un automa a stati finiti deterministico che riconosce lo stesso linguaggio.

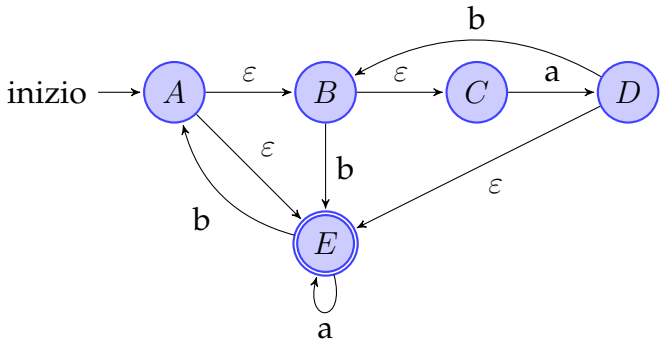


Figura 5.3: NFA dato

È riportata in seguito la tabella risolutiva per questo esercizio.

Stati (deterministici)	ε-chiusura dei punti di arrivo delle a-transizioni	ε-chiusura dei punti di arrivo delle b-transizioni
Stato iniziale n.d.: A $T0 = \varepsilon - cl(\{A\}) = \{A, B, C, E\}$	Stati tramite a-transizione $\{D, E\}$ $\varepsilon - cl(\{D, E\}) = \{D, E\} = T1$ [T1 unmarked]	Stati raggiunti tramite b-transizione $\{E, A\}$ $\varepsilon - cl(\{A, E\}) = \{A, B, C, E\} = T0$ [T0 è già presente!]
$T1 = \{D, E\}$	Possibili a-tranizioni $\{E\}$ $\varepsilon - cl(\{E\}) = T2$ [T2 unmarked]	Possibili b-transizioni $\{A, B\}$ $\varepsilon - cl(\{A, B\}) = \{A, B, C, E\} = T0$ [T0 è già presente!]
$T2 = \{E\}$	Possibili a-transizioni $\{E\}$ $\varepsilon - cl(\{E\}) = T2$	Possibili b-transizioni $\{A\}$ $\varepsilon - cl(\{A\}) = T0$

Tabella 5.2: Soluzione esercizio 2

Dato che lo svolgimento dei pass dell'algoritmo non presenta né novità né difficoltà in questo esempio, non è fornita una spiegazione estesa dei passaggi.

A questo punto abbiamo considerato tutti gli stati in R quindi ora è il momento di determinare quali sono gli stati (deterministici) finali. L'unico stato finale nell'NFA è E, quindi tutti e solo gli stati deterministici che contengono E sono stati deterministici finali. $T0$, $T1$ e $T2$ sono tutti stati finali. Il DFA risultante si può ammirare in figura 5.4.

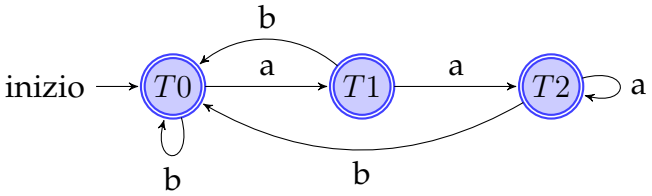


Figura 5.4: DFA ottenuto tramite

Esercizio 3

Dato l'automa a stati finiti non deterministico in figura 5.5, calcola tramite l'algoritmo di Subset Construction un automa a stati finiti deterministico che riconosce lo stesso linguaggio.

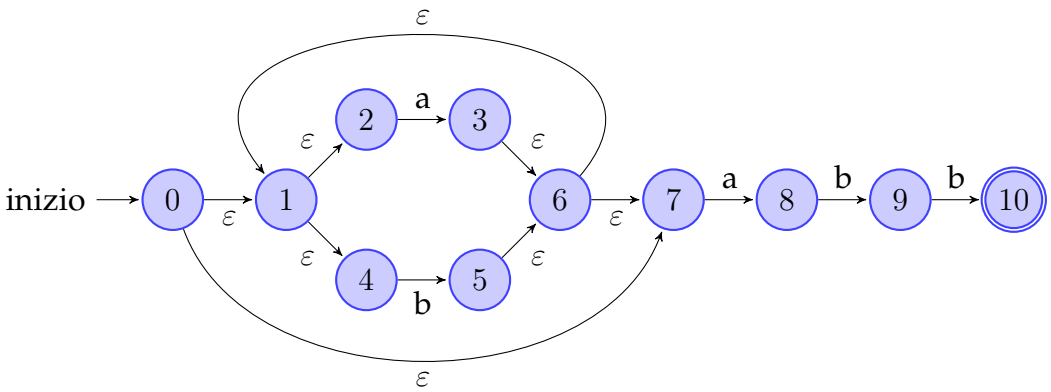


Figura 5.5: NFA dato

Come per l'esercizio precedente, è riportata in seguito la tabella risolutiva.

Stati (deterministici)	ε-chiusura dei punti di arrivo delle a-transizioni	ε-chiusura dei punti di arrivo delle b-transizioni
Stato iniziale n.d. : 0 $T0 = e - cl(\{0\}) = \{0,1,2,4,7\}$	Stati tramite a-transizione {3,8} $\varepsilon - cl(\{3,8\}) = \{1,2,3,4,6,7,8\} = T1$ [T1 unmarked]	Stati raggiunti tramite b-transizione {5} $\varepsilon - cl(\{5\}) = \{1,2,4,5,6,7\} = T2$ [T2 unmarked]
$T1 = \{1,2,3,4,6,7,8\}$	$\{3,8\}$ $\varepsilon - cl(\{3,8\}) = T1$ [T1 già analizzato]	$\{5,9\}$ $\varepsilon - cl(\{5,9\}) = \{1,2,4,5,6,7,9\} = T3$ [T3 unmarked]
$T2 = \{1,2,4,5,6,7\}$	$\{3,8\}$ [T1 già analizzato]	$\{5\}$ [T2 già analizzato]
$T3 = \{1,2,4,5,6,7,9\}$	$\{3,8\}$ [T1 già analizzato]	$\{5,10\}$ $\varepsilon - cl(\{5,10\}) = \{1,2,4,5,6,7,10\} = T4$ [T4 unmarked]
$T4 = \{1,2,4,5,6,7,10\}$	$\{3,8\}$ [T1 già analizzato]	$\{5\}$ [T2 già analizzato]

Tabella 5.3: Soluzione esercizio 3

Quindi abbiamo ben 5 stati per il DFA questa volta! Ora, gli stati finali sono solo quelli che contengono lo stato 10 dell'NFA, il che significa che solo $T4$ è uno stato finale per il DFA ricavato; tale DFA si può vedere in figura 5.6.

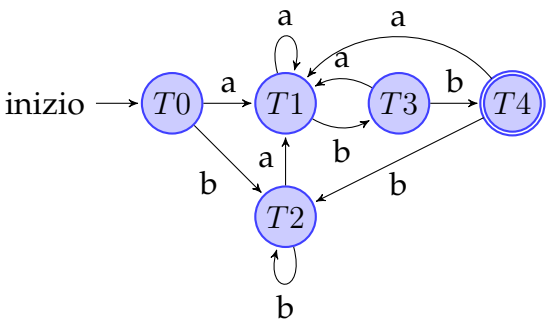


Figura 5.6: Esercizio 3: DFA

Il lettore attento avrà notato che, a suo tempo, avevamo già visto l'automa non deterministico in figura 5.5, e in quel momento avevamo già osservato come questo automa riconosca tutte le parole scritte secondo la formulazione $(a \mid b)^*abb$.

Il fatto interessante è che avevamo anche incontrato un altro automa che riconosce lo stesso linguaggio, il quale presentava solo 4 stati ed era molto più semplice e compatto di quello che abbiamo appena ricavato. Questo automa è riportato in figura 5.7 per completezza.

Questo va detto per sottolineare come l'algoritmo di Subset Construction sia un algoritmo che produce *una* soluzione al problema di traduzione NFA $\rightarrow$ DFA, ma non è detto che produca la soluzione *migliore* per questo problema.

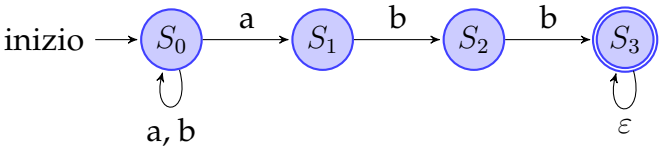


Figura 5.7: Esercizio 3: soluzione alternativa

Da notare che l'automa a stati finiti riportato in figura non è deterministico, ma solo per la presenza di una ε-transizione e di due transizioni da S_0 col simbolo a.

5.2 Minimizzazione di DFA

Spesso e volentieri non siamo interessati ad un generico DFA $\mathcal{D}$ che generi un altrettanto generico linguaggio $\mathcal{L}(\mathcal{D})$, ma desideriamo invece il DFA $\mathcal{D}'$ minimo, ovvero quello con numero minimo di stati tale che $\mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathcal{D}')$.

Prima di procedere, dobbiamo definire la nozione di *State Equivalence*.

5.2.1 Definizione di State Equivalence

Definizione 5.2.1. *Sia $\mathcal{D} = (S, \mathcal{A}, move_d, s_0, F)$ un DFA con funzione di transizione totale. Allora due stati $s, t \in S$ si dicono equivalenti se e solo se la seguente condizione è verificata:*

$$move_d^*(s, x) \in F \iff move_d^*(t, x) \in F, \forall x \in \mathcal{A}^* \tag{5.1}$$

dove la funzione multi-passo $move_d^*$ è definita per induzione sulla lunghezza della stringa considerata nel seguente modo:

Base $move_d^*(s, \varepsilon) = s$

Step $move_d^*(s, wa) = move_d(move_d^*(s, w), a)$

L'idea generale su cui si basa l'algoritmo di minimizzazione qui proposto è che alcuni stati sono ridondanti, e per questo possono essere rimossi senza perdere informazione, vale a dire senza modificare il linguaggio riconosciuto dal DFA. Per eliminare questi stati in eccesso, utilizziamo la procedura di *partition refinement*

5.2.2 Partition Refinement

Grazie a questa procedura arriveremo a partizionare gli stati in 2 blocchi, dove per blocchi intendiamo dei sottinsiemi disgiunti di stati.

- Iniziamo con 2 blocchi generici, $B_1 = F$ e $B_2 = S \setminus F$. Questa scelta fa sì che i due blocchi non abbiano stati equivalenti, poiché se consideriamo due stati s, t tali che $s \in B_1$ e $t \in B_2$, allora vale che $\text{move}_d^*(s, \varepsilon) \in F$ e $\text{move}_d^*(t, \varepsilon) \notin F$.
- Proseguiamo controllando se un blocco ha stati non equivalenti; se sì, li dividiamo in blocchi distinti, separando tra di loro gli stati non equivalenti; se no, analizziamo altri blocchi. Ripetiamo quest'operazione fino a quando non vi sono più blocchi con stati non equivalenti.

- In maniera formale, se tutti gli stati $B_i = \{s_1, \ldots, s_k\}$ di un blocco sono equivalenti, per ogni transizione $a \in \mathcal{A}$ il target di tale transazione appartiene allo stesso blocco B_j.
- Il blocco B_i può essere spezzato in due se per qualche $s, t \in B_i$ vale $\text{move}_d(s, a) \in B_j$ e $\text{move}_d(t, a) \notin B_j$, ossia se ci sono due stati non equivalenti.
- Nel concreto, dividere un certo blocco B_i rispetto a (a, B_j) consiste nel sostituire B_i con due nuovi blocchi:
  - $B_{i_1} = \{s \in B_i \mid \text{move}_d(s, a) \in B_j\}$;
  - $B_{i_2} = \{s \in B_i \mid \text{move}_d(s, a) \notin B_j\}$.

Di seguito lo pseudocodice per la procedura appena descritta.

Algorithm 7: partitionRefinement(DFA $\mathcal{D}$)

```
1 Set B_1 = F
2 Set B_2 = S \ F
3 Set P = {B_1, B_2 }
4 while qualche B_i ∈ P può essere spezzato rispetto a qualche (a, B_j) do
5 |   P.remove(B_i)
6 |   P.insert({s ∈ B_i | move_d(s,a) ∈ B_j})
7 |   P.insert({s ∈ B_i | move_d(s,a) ∉ B_j})
```

La divisione di B_i considerando (a, B_j) consiste nel sostituire B_i con due blocchi, $B_{i1} = \{s \in B_i \mid move_d(s, a) \in B\}$ e $B_{i2} = \{s \in B_i \mid move_d(s, a) \notin B\}$

5.2.3 Esercizi sulla minimizzazione DFA

Esercizio 1

Passiamo ad applicare la procedura di minimizzazione ad uno degli esempi già visti in passato.

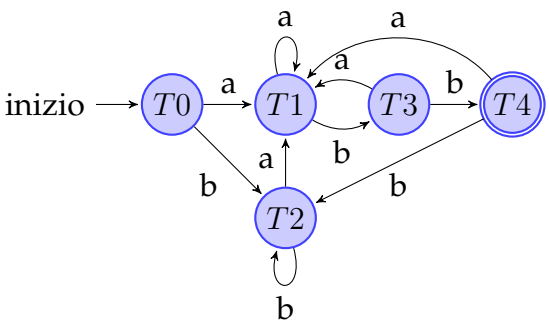


Figura 5.8: DFA da minimizzare dato

Iniziamo con due blocchi iniziali: $B_1 = \{T4\}$, dato dallo stato finale, e $B_2 = \{T0, T1, T2, T3\}$. Notiamo che (b, B_1) spezza B_2, per cui abbiamo due blocchi $B_{21} = \{T3\}$ e $B_{22} = \{T0, T1, T2\}$. Notiamo ancora che è possibile spezzare B_{22} in $B_{221} = \{T1\}$ e $B_{222} = \{T0, T2\}$. Otteniamo quindi il DFA minimo illustrato in figura 5.9.

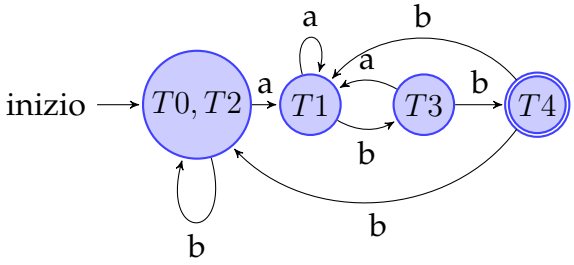


Figura 5.9: Soluzione di questo primo esercizio

Esercizio 2

Proviamo con questo altro esercizio:

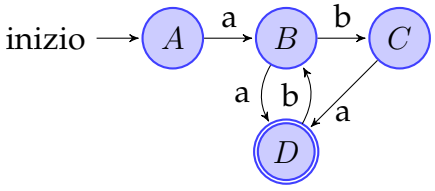


Figura 5.10: DFA dato

In questo caso la funzione di transizione non è totale; dobbiamo quindi renderla tale aggiungendo un nodo $sink$ a cui facciamo arrivare ogni coppia $stato, simbolo$ mancante dall'equivalente funzione di transizione parziale.

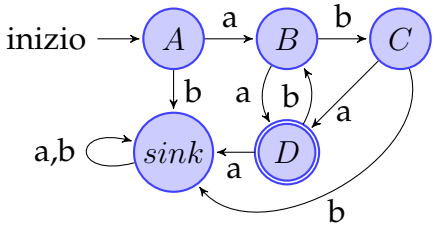


Figura 5.11: DFA con transizione totale

Dopo aver realizzato questo, iniziamo con gli insiemi $B_1 = \{D\}$ e $B_2 = \{A, B, D, sink\}$. Possiamo fare uno *split* di B_2, in quanto move$_d(A, a) \in B_2$ e move$_d(B, a) \notin B_2$; da questo split otteniamo i nuovi blocchi $B_{21} = \{A, sink\}$ e $B_{22} = \{B, C\}$.

Possiamo fare un ulteriore *split* di B_{22}, poiché move$_d(B, b) \in B_{22}$ e move$_d(C, b) \notin B_{22}$; da questo otteniamo i blocchi $B_{221} = \{B\}$ e $B_{222} = \{C\}$.

Infine notiamo che è possibile uno *split* di B_{21}, dal momento che move$_d(A, a) \in B_{221}$ e move$_d(Sink, a) \notin B_{221}$. Otteniamo quindi i blocchi $B_{211} = \{A\}$ e $B_{212} = \{sink\}$. La situazione finale diventa $B_1 = \{D\}$, $B_{211} = \{A\}$, $B_{212} = \{sink\}$, $B_{221} = \{B\}$, $B_{222} = \{C\}$.

Da questo eliminiamo il blocco B_{212}, in quanto il blocco *Sink* non trova posto nel nostro DFA minimo; notiamo però che il DFA minimo che otteniamo è identico a quello di partenza. Il DFA, infatti, era già minimo.

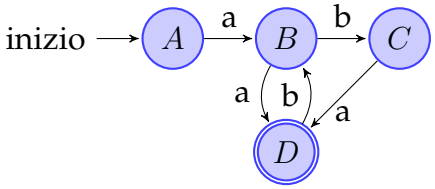


Figura 5.12: Soluzione Esercizio 2

Esercizio 3

Passiamo ora ad un esercizio più completo. Si richiede di trasformare l'NFA in Fig.5.13 in un DFA e successivamente in un min DFA.

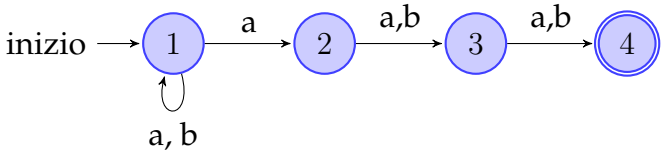


Figura 5.13: Esercizio 3

Procediamo dunque, come da prassi, con lo svolgimento della procedura di subset construction per creare un DFA corrispondente. La tabella che riassume tale procedimento è riportata come Tab.5.4.

		a	b
$T0 = \{1\}$	Initial	$T1$	$T0$
$T1 = \{1, 2\}$		$T2$	$T3$
$T2 = \{1, 2, 3\}$		$T4$	$T5$
$T3 = \{1, 3\}$		$T6$	$T7$
$T4 = \{1, 2, 3, 4\}$	Final	$T4$	$T5$
$T5 = \{1, 3, 4\}$	Final	$T6$	$T7$
$T6 = \{1, 2, 4\}$	Final	$T2$	$T3$
$T7 = \{1, 4\}$	Final	$T1$	$T0$

Tabella 5.4: Tabella per la subset construction

A questo punto dobbiamo costruirne la rappresentazione del DFA, che si può osservare in Fig.5.14, e trovarne il minimo. Procediamo ora con l'algoritmo di partition refinement per minimizzare il precedente DFA.

Il primo passo è quello di dividere l'insieme S degli stati in due blocchi: gli stati finali ($B2$) e gli stati non finali ($B1$). In questo caso le partizioni sono le seguenti:

$$B1 = \{T0, T1, T2, T3\} \quad \text{e} \quad B2 = \{T4, T5, T6, T7\}$$

A questo punto procediamo con le raffinazioni successive. Se all'interno dello stesso blocco troviamo due stati che, tramite una a-transizione, portano a due blocchi distinti (con a terminale qualsiasi dell'alfabeto $\mathcal{A}$), allora splittiamo questo blocco in due nuovi blocchi nel seguente modo:

1. identifichiamo i due stati s_1 e s_2 che con una a-transizione portano rispettivamente uno in un blocco By e l'altro in un blocco Bz;

2. creiamo due nuovi sottoblocchi di Bx: Bx_1 conterrà tutti gli stati che da Bx arrivano a By con una a-transizione e Bx_2 conterrà tutti gli stati in $Bx \setminus Bx_1$.

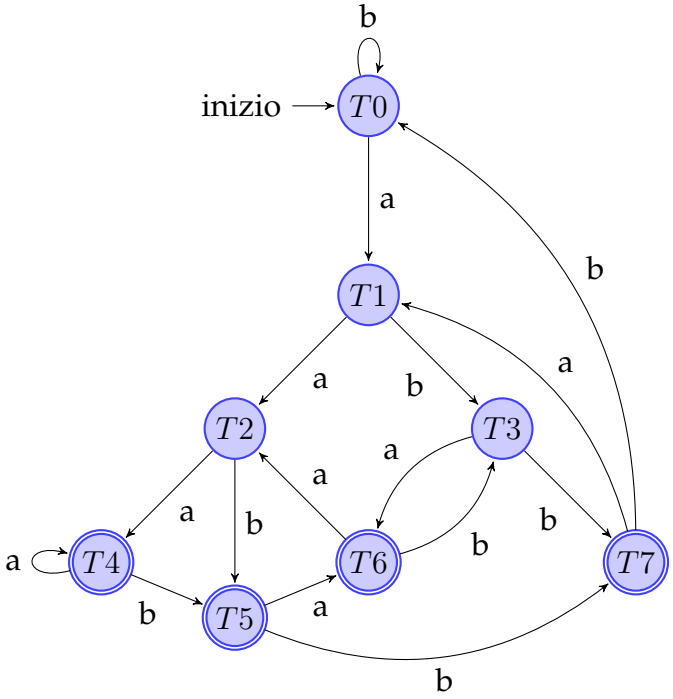


Figura 5.14: DFA ottenuto con subset construction

Preme sottolineare che la scelta di quale blocco splittare influisce molto sull'efficienza dell'algoritmo, e gli algoritmi più efficienti svolgono questa scelta in maniera mirata.

Osservo che move$_d(T2, a)$ non ha lo stesso blocco target di move$_d(T0, a)$: infatti, con una a-transizione da $T0$ raggiungo $B1$, con una a-transizione da $T2$ raggiungo $B2$. Decido quindi di splittare $B1$ secondo move$_d(T0, a)$. In questo caso il risultato della partizione è il seguente:

$$\{T0, T1\};\ \{T2, T3\};\ \{T4, T5, T6, T7\}$$

Questa nuova partizione è ulteriormente splittabile? Sì! Posso eseguire uno split su move$_d(T0, a)$, dato che attraverso una a-transizione da $T0$ mi muovo in $\{T0, T1\}$, mentre da $T1$ mi muovo in $\{T2, T3\}$. Splitto quindi il blocco $\{T1, T2\}$ e ottengo questa nuova suddivisione:

$$\{T0\};\ \{T1\};\ \{T2, T3\};\ \{T4, T5, T6, T7\}$$

Ora procediamo ad oltranza con i raffinamenti successivi. Posso splittare su move$_d(T6, b)$ e move$_d(T7, b)$, ottenendo:

$$\{T0\};\ \{T1\};\ \{T2, T3\};\ \{T4, T5, T7\};\ \{T6\}$$

Il prossimo è lo split su move$_d(T2, a)$ e move$_d(T3, a)$, che mi porta ad ottenere:

$$\{T0\};\ \{T1\};\ \{T2\};\ \{T3\};\ \{T4, T5, T7\};\ \{T6\}$$

E ora splitto su move$_d(T4, a)$ e move$_d(T7, a)$:

$$\{T0\};\ \{T1\};\ \{T2\};\ \{T3\};\ \{T4, T5\};\ \{T7\};\ \{T6\}$$

Infine, posso splittare anche su move$_d(T4, a)$ ed move$_d(T5, a)$, arrivando quindi al raffinamento con la seguente forma:

$$\{T0\};\ \{T1\};\ \{T2\};\ \{T3\};\ \{T4\};\ \{T5\};\ \{T7\};\ \{T6\}$$

A questo punto ho raggiunto una partizione in cui tutti i blocchi contengono un solo stato.

In questo caso ho terminato lo svolgimento dell'algoritmo di partition refinement; questo significa che il DFA da cui sono partito (ottenuto da subset construction) era esattamente minimo.

Discussione: "Qual è la scelta ottimale per lo split?"

Domanda interessante. La discussione, essendo un trivia, verrà trascritta in un ~~secondo~~ terzo momento[1].

5.2.4 Dimensione di un DFA

Lemma 5.2.1. *Per ogni $n \in \mathbb{N}^+$ esiste un NFA con $(n+1)$ stati il cui DFA minimo equivalente ha almeno 2^n stati.*

Dimostrazione. Prendiamo ad esempio il linguaggio:

$$\mathcal{L} = \mathcal{L}((a \mid b)^* a (a \mid b)^{n-1}) \tag{5.2}$$

Dal momento che è un linguaggio regolare, questo può essere rappresentato utilizzando un automa a stati finiti. Esiste un NFA con esattamente $n+1$ stati che accetta il linguaggio $\mathcal{L}$; tale automa è rappresentiato in figura 5.15.

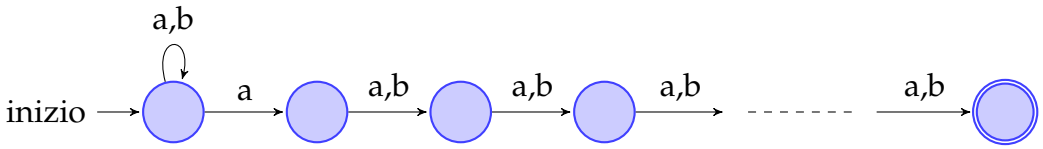


Figura 5.15: NFA rappresentante il linguaggio 5.2

[1] Ragazzi, chiediamo venia, abbiammo scoperto di questo todo a poche ore dalla release, non c'era modo di riprendere in mano il discorso.

Supponiamo per assurdo che esista un DFA minimo, diaciamo $\mathcal{D}$, che accetti il linguaggio $\mathcal{L}$ ed abbia $k < 2^n$ stati.

Sappiamo esserci esattamente 2^n parole distinte su $\{a, b\}$ la cui lunghezza è n. Esistono quindi 2 percorsi in $\mathcal{D}$ tali che:

- la loro lunghezza è n;
- compongono rispettivamente le parole w_1 e w_2, con $w_1 \neq w_2$;
- condividono almeno un arco.

Se non esistessero, allora i nodi sarebbero come minimo 2^n; di conseguenza per qualche x_1, x_2 e x vale una delle due opzioni seguenti:

- $w_1 = x_1ax$ e $w_2 = x_2bx$ **oppure**
- $w_1 = x_1bx$ e $w_2 = x_2ax$

altrimenti le due parole w_1 e w_2 sarebbero uguali.
Supponiamo in questo caso che sia valida la condizione:

$$w_1 = x_1ax \quad \text{e} \quad w_2 = x_2bx$$

Allora possiamo definire w_1' in questo modo:

$$w_1' = x_1ab^{n-1} \in \mathcal{L}(\mathcal{D})$$

Di conseguenza, lo stato nuovamente aggiunto da w_1' in $\mathcal{D}$ è uno stato finale. La precedente affermazione è una contraddizione: lo stato non può essere finale, perché può essere raggiunto anche tramite x_2bb^{n-1}, ma

$$x_2bb^{n-1} \notin \mathcal{L}(\mathcal{D})$$

questo è l'assurdo che prova la validità del lemma. QED

Tutto il lemma gioca sul fatto che un DFA ha bisogno esattamente di tutti i possibili cammini che siano composti da $n-1$ alternanze di a e di b per esprimere lo stesso linguaggio espresso dall'NFA.

Questo lemma quindi ci offre una stima per il numero minimo di stati di un DFA che rappresenta un certo altro NFA.

5.3 Pumping Lemma per linguaggi regolari

5.3.1 Formulazione

Lemma 5.3.1. *Sia $\mathcal{L}$ un linguaggio regolare; allora vale che:*

- $\exists p \in \mathbb{N}^+$ *tale che*
- $\forall z \in \mathcal{L}$ *tale che* $|z| > p$
- $\exists\ u, v, w$ *tale che*
  - $z = uvw$ *e*
  - $|uv| \leq p$ *e*
  - $|v| > 0$ *e*
  - $\forall i \in \mathbb{N}\ .\ uv^iw \in \mathcal{L}$

Ovvero, come il lettore avrà ben notato, questo è il corrispettivo del pumping lemma che già avevamo conosciuto parlando di linguaggi liberi. Illustriamo ora la dimostrazione di questo lemma.

Dimostrazione. Dato che $\mathcal{L}$ è un linguaggio regolare, sappiamo che esiste un DFA $\mathcal{D} = (S, \mathcal{A}, \text{move}_d, S_0, F)$ tale per cui $\mathcal{L} = \mathcal{L}(\mathcal{D})$.

Fissiamo quindi $p = |S| - 1$; vorremmo far notare che tutti i cammini che partono da S_0 ed arrivano ad uno stato finale, passando per ogni stato al massimo una volta, hanno lunghezza limitata da p.

Quindi, se per una parola z vale $|z| > p$, allora possiamo scomporre z in $z = a_1 \dots a_p z'$ e abbiamo la certezza che almeno uno stato, diciamo s^*, sia attraversato più di una volta lungo il cammino $a_1 \dots a_p$.

Questo significa che esiste un ciclo in $\mathcal{D}$ che parte da s^* e torna in s^*, e questo ciclo si può indicare come $a_{i+1} \dots a_j$, con $i < j \leq p$.

Definiamo ora u, v e w:

- $u = a_1 \dots a_i$
- $v = a_{i+1} \dots a_j$, (quindi v rappresenta il ciclo)
- $w = \begin{cases} z' & \text{se } j = p \\ a_{j+1} \dots a_p z' & \text{se } j < p \end{cases}$

Quindi possiamo tranquillamente affermare che $|uv| \leq p$ e che la lunghezza di v è almeno 1 (per definizione un ciclo ha almeno un nodo). Inoltre, essendo v il ciclo, esso è ripetibile (pumping) un numero indefinito di volte, con la garanzia che uv^iw sia accettato da $\mathcal{D}$ per ogni valore $i \in \mathbb{N}$. QED

Abbiamo quindi dimostrato questa versione del pumping lemma per i linguaggi regolari, ora la domanda che sorge spontanea è: per cosa si utilizza questo lemma? Come nel caso dei linguaggi liberi, questo lemma viene utilizzato per dimostrare, tramite contraddizione, che un certo linguaggio *non* è regolare. Quindi, si assume che un dato linguaggio $\mathcal{L}$ sia regolare, e se si riesce a dimostrare che per quel linguaggio la tesi non è soddisfatta (ovvero il negato della tesi è soddisfatto), allora si può affermare che $\mathcal{L}$ in realtà non è un linguaggio regolare.

Ripetiamo la tesi, per amor di chiarezza, e ne scriviamo in seguito la negazione.

Tesi $\exists p \in \mathbb{N}^+.\ \forall z \in \mathcal{L} : |z| > p.\ \exists\ u, v, w.\ P$,
dove
$P \equiv (z = uvw$ and $|uv| \leq p$ and $|v| > 0$ and $\forall i \in \mathbb{N}.\ uv^iw \in \mathcal{L})$

Negato $\forall p \in \mathbb{N}^+.\ \exists z \in \mathcal{L} : |z| > p.\ \forall\ u, v, w.\ Q$,
dove
$Q \equiv (z = uvw$ and $|uv| \leq p$ and $|v| > 0) \implies (\exists i \in \mathbb{N}.\ uv^iw \notin \mathcal{L})$

5.3.2 Applicazioni

Vediamo subito come possiamo applicare il lemma per dimostrare la seguente affermazione:

$$\mathcal{L} = \{a^n b^n \mid n > 0\} \quad \text{non è regolare.} \tag{5.3}$$

Prima di proseguire con la dimostrazione formale, proviamo a spiegare che intuizione potrebbe dirci che ci troviamo davanti ad un linguaggio non regolare.

Se $\mathcal{L}$ fosse un linguaggio regolare dovrebbe essere possbile rappresentarlo con un NFA o un DFA. Proviamo quindi a immaginare il DFA che riconosce questo linguaggio, come dovrebbe essere? Per bilanciare il numero di b e di a dovrebbe in qualche modo obbligare l'inserimento di una b per ogni a inserita in precedenza, ma questo non si può fare a meno di inserire infiniti stati, il che è impossibile.

Procediamo quindi con la dimostrazione tramite pumping lemma. Supponiamo $\mathcal{L}$ regolare e prendiamo p intero positivo arbitrario (il negato della tesi deve valere per ogni p). Ora, costruiamo la parola $z = a^p b^p$; abbiamo che $|z| > p$. Osserviamo che, per ogni decomposizione di z in u, v, w che rispetta i vincoli qui sotto riportati:

$$\forall\ u, v, w \text{ se } (z = uvw \quad \text{e} \quad |uv| \leq p \quad \text{e} \quad |v| > 0)$$

Possiamo quindi affermare che:

- la componente v contiene almeno una a (da $|v| > 0$);

- la componente v deve contenere solamente a (altrimenti violerebbe $|uv| \leq p$).

Se noi ora definiamo $j > 0$ tale per cui

$$uv^2w = a^p a^j b^p$$

abbiamo che $uv^2w \notin \mathcal{L}$, il che contraddice il pumping lemma per i linguaggi regolari.

Abbiamo quindi dimostrato per contraddizione del pumping lemma per linguaggi regolari che l'affermazione in equazione 5.3 è verificata.

5.3.3 Esercizi sul riconoscimento di linguaggi regolari

Esercizio 1

Sia $\mathcal{L}_1$ il linguaggio delle parole su $\{a, b\}$ con un numero dispari di occorrenze di b. $\mathcal{L}_1$ è regolare? E se lo è, riusciamo a trovare un'espressione regolare per questo linguaggio e un DFA che la riconosca?
Proponiamo di seguito delle espressioni regolari e dei DFA che sono stati selezionati durante la lezione con una breve spiegazione a riguardo

- l'espressione regolare $(a^*(bb)^*)^*b(a^*(bb)^*)^*$ non può denotare il linguaggio descritto poiché non riconosce $ababab$.

- definiamo un automa con due stati A iniziale e B finale. A presenta una a-transizione in A ed una b-transizione in B mentre B ha una a-transizione in B e una b-transizione in A. Notare che lo stato A rappresenta il caso in cui sono state lette un numero pari di b mentre B quello in cui sono state lette un numero dispari di b: nel caso iniziale, infatti, si è obbligati ad aggiungere almeno una b per raggiungere lo stato finale ma, nel caso in cui si voglia aggiungere un'altra b, è necessario ritornare nello stato A e rifare il procedimento descritto. Considerando che in entrambi gli stati è possibile mettere un numero arbitrario di a, possiamo dichiarare questa soluzione come corretta (Fig.5.16).

- l'espressione regolare $(a^*(b)a^*)(a^*(b)a^*(b)a^*)^*$ sembrerebbe essere corretta in quanto sicuramente accetterò parole composte da almeno una b e un numero arbitrario di a prima e dopo; nel caso in cui volessi aumentare il numero di b, sono obbligato a farlo a coppie con la possibilità di interporre un numero arbitrario di a prima, in mezzo o alla fine.

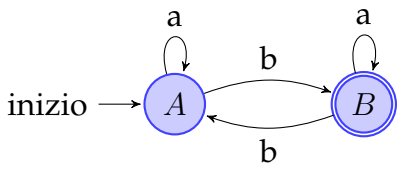


Figura 5.16: DFA che riconosce un numero dispari di occorrenze di b in un linguaggio su $\{a, b\}$

In questo caso, visto che siamo stati in grado di trovare un DFA con un numero finito di stati, possiamo affermare che $\mathcal{L}_1$ è un linguaggio regolare. Come possiamo però essere certi del fatto che, ad esempio, l'espressione regolare $(a^*(b)a^*)(a^*(b)a^*(b)a^*)^*$ denoti il linguaggio $\mathcal{L}_1$ e che quindi sia anch'essa soluzione al quesito? Il modo corretto di procedere è il seguente:

1. si utilizza la *Thompson's Construction* per generare l'NFA corrispondente;
2. si applica l'algoritmo di *Subset Construction* per trasformare l'NFA in un DFA;
3. si applica l'algoritmo per la *DFA minimization* (aggiungendo dunque lo stato sink se necessario) per ridurre il numero di stati;
4. si osserva se il DFA ridotto è in realtà un isomorfismo del DFA proposto come soluzione (in questo caso il DFA proposto come soluzione è sicuramente minimo, in quanto ha solamente due stati).

Notare che in questo caso una soluzione possibile (proposta dalla professoressa) è $(ba^*b \mid a)^*ba^*$

Esercizio 2

Sia $\mathcal{L}_2$ il linguaggio delle parole su $\{a, b\}$ con un numero pari di occorrenze di a. $\mathcal{L}_2$ è regolare?

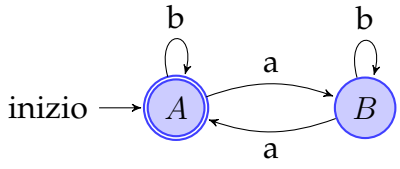


Figura 5.17: DFA che riconosce un numero pari di occorrenze di a in un linguaggio su $\{a, b\}$

Per rispondere a tale quesito si potrebbe pensare di costruire un DFA con lo stesso ragionamento di prima: uno stato il caso in cui le occorrenze di a sono pari mentre l'altro in cui sono dispari.

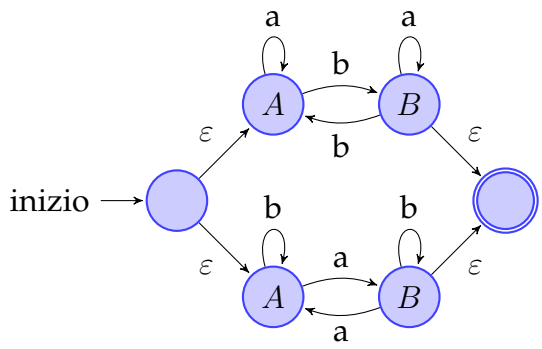
Poiché 0 è considerato pari, in questo caso lo stato iniziale coincide con quello finale. Le transizioni devono essere opportunamente costruite, per cui se voglio aggiungere una a alla mia parola sono costretto a seguire una a-transizione in uno stato non finale per poi tornare in quello finale con un'ulteriore a-transizione, in modo da assicurare che le condizioni siano rispettate. Ovviamente, deve esservi sempre la possibilità di inserire un numero arbitrario di b tra le varie occorrenze di a, per cui in entrambi gli stati si ha una b-transizione sullo stesso stato. Analogamente all'esercizio precedente si ha che $\mathcal{L}_2$ è un linguaggio regolare. In questo caso l'espressione regolare proposta è stata $(b^*ab^*ab^*)^*$.

Esercizio 3

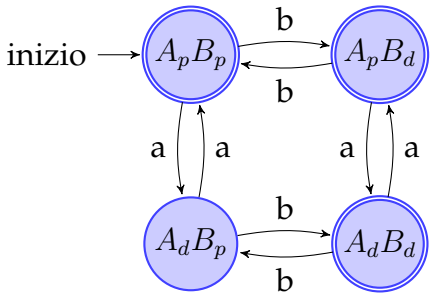
Sia $\mathcal{L}_3$ il linguaggio delle parole su $\{a, b\}$ con un numero dispari di occorrenze di b **oppure** un numero pari di occorrenze di a. $\mathcal{L}_3$ è regolare?

Una soluzione tanto banale quanto efficace consiste nel utilizzare la Thompson's Construction per poter concatenare gli automi descritti negli esercizi precedenti con l'operazione di *alternanza*. A livello formale questo si traduce come:

$$\mathcal{L}(r_1 \mid r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$$



(a) NFA che riconosce un numero dispari di occorrenze di b **oppure** un numero pari di occorrenze di a in un linguaggio su $\{a, b\}$



(b) DFA che riconosce un numero dispari di occorrenze di b **oppure** un numero pari di occorrenze di a in un linguaggio su $\{a, b\}$

Figura 5.18

Essendo che $\mathcal{L}_1$ e $\mathcal{L}_2$ sono linguaggi regolari, l'unione dei due così descritta non può che essere anch'essa un linguaggio regolare.

Esercizio 4

Riprendiamo la formulazione dell'esercizio precedente; nel caso in cui fosse stato richiesto di verificare un linguaggio dove valessero **entrambe** le condizioni precedenti avremmo dovuto procedere in modo diverso. Sono dati i seguenti casi:

- abbiamo un numero di a pari e di b pari;
- abbiamo un numero di a pari e di b dispari;
- abbiamo un numero di a dispari e di b pari;
- abbiamo un numero di a dispari e di b dispari.

Ovviamente il nostro stato iniziale non può essere altro che quello dove si hanno 0 occorrenze di a e di b (il primo descritto), mentre lo stato finale dovrà per forza essere il secondo dell'elenco. Dunque quello che abbiamo appena immaginato di costruire è un'automa che possiede 4 possibili stati di cui soltanto uno deve essere considerato quello finale.

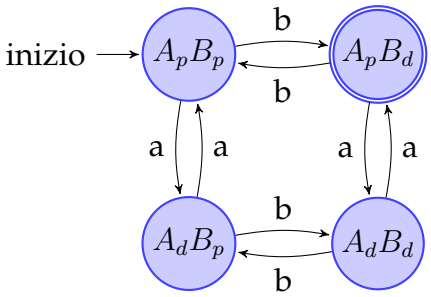
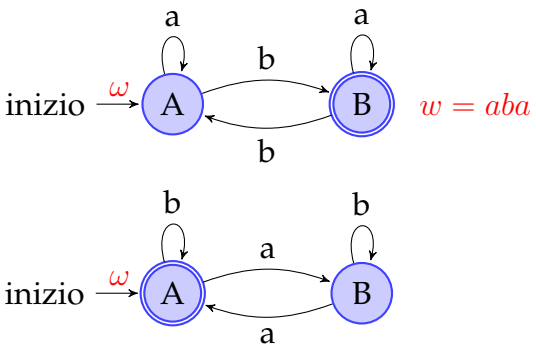


Figura 5.19: DFA che riconosce un numero dispari di occorrenze di b **e** un numero pari di occorrenze di a su un linguaggio su $\{a, b\}$

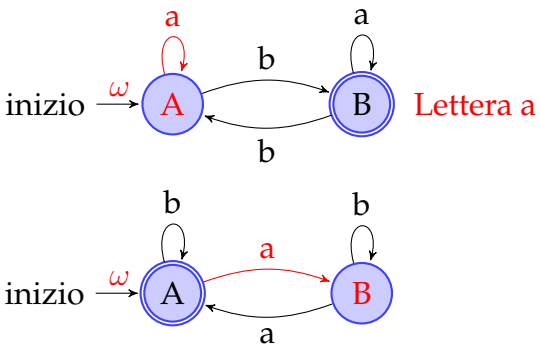
Sicuramente l'idea precedente è corretta, ma c'era un altro modo per arrivare direttamente alla soluzione senza dover fare un ragionamento di quel tipo, ma pensando invece all'intersezione: immaginiamo di avere una parola w a disposizione e di poterci contemporaneamente muovere su entrambi i cammini degli automi descritti come soluzioni di $\mathcal{L}_1$ e di $\mathcal{L}_2$. Continuiamo a spostarci eseguendo le transizioni coerentemente alla parola w scelta, fino a che non arriviamo al carattere terminatore. A questo punto sono dati due casi:

1. se mi trovo in entrambi gli automi su uno stato finale allora $w \in \mathcal{L}$
2. se uno dei due stati su cui mi trovo non è uno stato finale allora $w \notin \mathcal{L}$

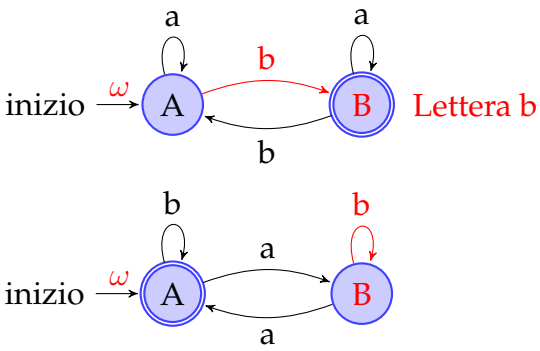
Osserviamo subito con un esempio questo approccio messo in pratica: vogliamo verificare se la parola $aba \in \mathcal{L}$. Per fare questa verifica useremo appunto entrambi gli automi ricavati per $\mathcal{L}_1$ e per $\mathcal{L}_2$.



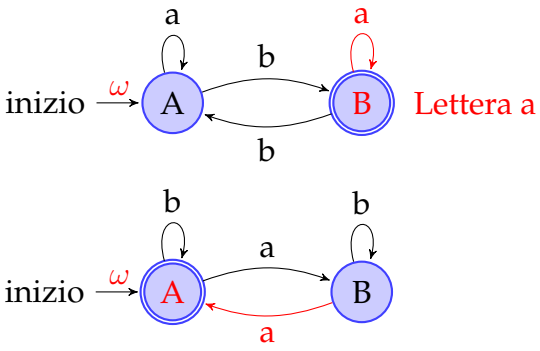
(a) siamo all'inizio della simulazione, l'automa in alto riconosce il $\mathcal{L}_1$, quello in basso $\mathcal{L}_2$, a questo punto andremo a leggere i caratteri della parola aba uno alla volta e ci sposteremo su entrambi gli automi per verificare che la parola sia riconosciuta da entrambi;



(b) leggiamo il carattere a, nel primo automa questo ci porta a compiere un self-loop su A, nel secondo porta a transitare verso lo stato B, in nessuno dei due automi siamo in uno stato finale, questo perché la parola a che ha a dispari e b pari non appartiene né a $\mathcal{L}_1$ né $\mathcal{L}_2$;



(c) A questo punto leggiamo b, quindi nel primo automa ci spostiamo nello stato B, mentre nel secondo abbiamo un self-loop su B, in questo caso vediamo che ci troviamo in uno stato finale nell'automa per $\mathcal{L}_1$: questo perché la parola ab ha un numero dispari di b, quindi appartiene al linguaggio $\mathcal{L}_1$;



(d) infine leggiamo il carattere a, che ci porta in B nel primo automa ed in A nel secondo, entrambi sono stati finali: la parola aba è riconosciuta da entrambi gli automi poiché contiene un numero dispari di b ed un numero pari di a.

Figura 5.20

Per poter semplificare la procedura possiamo immaginare di fare il prodotto cartesiano degli stati dei due automi anche se questo potrebbe risultare più complesso da immaginare.

5.4 Proprietà di chiusura nei linguaggi regolari

I linguaggi regolari sono chiusi rispetto alle seguenti operazioni:

- unione;
- concatenazione;
- complementazione;
- intersezione.

Andiamo a vedere più da vicino le motivazioni per ciascuna di queste.

Unione Se prendiamo due linguaggi regolari e ne operiamo l'unione, il risultato di questa è ancora un linguaggio regolare. Come in esercizio 5.3.3, essendo che i linguaggi sono regolari, allora è possibile costruire un DFA sia per il primo che per il secondo; se a questo punto si prendono i due automi così costruiti e li si lega tramite una ε-transizione, per l'operazione di *alternanza* si ottiene nuovamente un linguaggio regolare.

Concatenazione È possibile eseguire l'operazione di *concatenazione* della Thompson's Construction e ottenere ancora una volta un'automa che denota un linguaggio regolare.

Complementazione Se $\mathcal{L}$ è un linguaggio regolare su un certo alfabeto $\mathcal{A}$, la complementazione è data dal linguaggio delle parole su $\mathcal{A} \setminus \mathcal{L}$. Se un linguaggio è regolare allora anche il suo complemento è regolare.

In precedenza abbiamo visto l'automa che riconosce $\mathcal{L}_2$, ovvero che accetta tutte quelle parole dove a occorre un numero pari di volte. Il complementare di tale automa corrisponde a quello per cui si hanno parole con un numero dispari di occorrenze. Utilizzando l'automa che denota $\mathcal{L}_2$, vi è la possibilità di applicare una particolare operazione per trasformarlo nel suo complementare? Sì, è possibile: basta semplicemente rendere finali tutti quegli stati che prima non lo erano e, simmetricamente, non finali tutti quelli che prima lo erano.

Relativamente a questo procedimento è però necessario fare una precisazione: tale metodo funziona solamente se il DFA presenta una funzione di transizione **totale**: intuitivamente, è necessario riconoscere tutte quelle parole che non appartengono al linguaggio regolare in quanto queste apparterranno al linguaggio del DFA complementare; se per assurdo si avesse la mancanza di una a-transizione (non fondamentale nel primo DFA) non si avrebbe la possibilità di costruire il set completo delle parole.

Intersezione È derivabile dall'unione utilizzando le regole di *De Morgan*:

$$\mathcal{L}_1 \cap \mathcal{L}_2 = \neg(\neg(L1 \cap L2)) = \neg(\neg\mathcal{L}_1 \cup \neg\mathcal{L}_2)$$

Si noti come abbiamo già discusso in precedenza come unione e negato mantengano la regolarità del codice, quindi grazie alla chiusura per unione e complementazione vale anche la chiusura per intersezione.

Parte II

Il processo di compilazione

Capitolo 6

Analisi lessicale

6.1 Rimandi all'analisi lessicale

Dopo i precedenti due capitoli, nei quali abbiamo introdotto e approfondito un buon numero di concetti, algoritmi e modelli necessari alla comprensione di come l'impostazione teorica dei linguaggi formali sia fondamentale nella progettazione dei compilatori, andiamo a tuffarci in quella che è la prima fase della compilazione: l'analisi lessicale.

Ricordiamo che questa è la fase in cui vogliamo identificare quali parti del sorgente che abbiamo scritto corrispondono alle keyword, quali agli identificatori, alle costanti e via di questo passo; per essere più formali, questi elementi che vogliamo riconoscere portano il nome di *lessemi*. Il nostro obiettivo in questa fase è trasformare quindi il sorgente in un flusso di tokens, i quali costituiscono i terminali della grammatica che genera il nostro linguaggio di programmazione.

Ricordiamo per l'ennesima volta che la grammatica di un linguaggio ci dirà quali sono le forme che un'espressione deve avere per essere considerata ben formata rispetto a quel linguaggio. Ad esempio, una grammatica ci può dire che la seguente forma denota un'espressione valida:

<identificatore> <simbolo di assegnamento> <numero>

e che quindi espressioni come la seguente sono grammaticali e ben formate secondo il linguaggio.

```
pippo = 2;
```

Il mestiere dell'analizzatore lessicale è proprio quello di ricevere in input un `pippo` qualsiasi (che è un token) e, in output, determinare che è un *identificatore*, ossia la categoria più astratta (lessema) di cui `pippo` è istanza.

6.1.1 Esempio: la grammatica di C99

Andiamo a vedere da vicino la grammatica di un reale linguaggio, anzi, del linguaggio preferito di tutti noi, ossia il C (nella versione 99, scritta per il parser *Bison*); il lettore interessato ad approfondire può trovare lo stesso file al seguente indirizzo: `http://www.quut.com/c/ANSI-C-grammar-y-1999.html`. Andiamo a vedere un frammento di quel codice:

```
primary_expression
  : IDENTIFIER
  | CONSTANT
  | STRING_LITERAL
  | '(' expression ')'
  ;
```

Notiamo subito alcune differenze rispetto alla notazione che abbiamo impiegato finora:

- la freccia ($\rightarrow$) è rappresentata da un altro simbolo, i due punti (:);
- il pipe (|), invece, possiede l'abituale significato;
- i terminali possono essere indicati precisamente se inseriti tra singoli apici '*terminale*';
- inoltre, la convenzione rispetto alla capitalizzazione è invertita: qui osserviamo che gli elementi in maiuscolo indicano i terminali, mentre invece quelli in minuscolo rappresentano non terminali; ad esempio, in figura sopra possiamo notare che il non-letterale `primary_expression` ha una produzione per cui può risultare o in una serie di letterali (`IDENTIFIER`, `CONSTANT`), oppure in una forma `'(' expression ')'`, dove `expression` è un altro non-letterale, che a sua volta avrà altre produzioni.

Questo file, inoltre, è pensato per essere utilizzato in tandem con un analizzatore sintattico; per questo motivo, nell'intestazione dello stesso possiamo trovare le dichiarazione di quelli che sono i token (vale a dire, lo ripetiamo, i terminali della grammatica descritta). La forma con cui saranno espressi è rappresentata dal listato di codice sottostante:

```
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME
```

```
6
7  %token TYPEDEF EXTERN STATIC AUTO REGISTER INLINE RESTRICT
8  %token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE
   ↪  VOID
9  %token BOOL COMPLEX IMAGINARY
10 %token STRUCT UNION ENUM ELLIPSIS
11
12 %token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK
   ↪  RETURN
```

Un'altra cosa che possiamo osservare è la dichiarazione di quello che è lo starting symbol della grammatica:

```
14 %start translation_unit
```

Questa dichiarazione in realtà potrebbe non essere presente su alcuni file contenenti grammatiche, dal momento che, in sua assenza, non otteniamo degli errori, bensì viene preso il non-letterale della prima produzione listata a seguito come starting symbol.

Il ruolo dell'analizzatore lessicale è quindi analizzare il sorgente e decidere, di volta in volta, quale derivazione può essere applicata a ciascuna delle righe del sorgente analizzato. Una volta completato l'albero di derivazione e quindi arrivato presso un terminale (ad esempio `IDENTIFIER`), l'analizzatore lessicale andrà anche ad associargli informazioni come valore e tipo, dal momento che è necessario distinguere quell'`IDENTIFIER` da un altro. Nel programma potremmo appunto avere due `IDENTIFIER` di diverso tipo, ma in ogni caso dovremo conservare delle informazioni aggiuntive di qualche tipo (nome, tipo, scope e altro ancora). Tutte queste informazioni vengono conservate in una *symbol table*.

Infine, le ultime righe del file contengono informazioni utili al particolare analizzatore sintattico utilizzato; avremo modo di parlarne nel dettaglio in futuro.

6.1.2 Classi di tokens

Si potrebbe a ragione considerare superfluo specificare che ogni diversa grammatica (e quindi ogni linguaggio) presenta diverse categorie di tokens; banalmente, il lettore è probabilmente ben cosciente che linguaggi diversi possiedono generalmente keyword diverse. Ad ogni modo, a seguito presentiamo alcune tra le scelte più ricorrenti:

- un token per ogni keyword, quindi un token per ogni nome di base già presente nel linguaggio (**`if`**, **`while`**, **`for`** e via dicendo);

- un token per ogni operatore (o anche per classe di operatori), quindi in C ne avremo uno per + ma anche uno dedicato per ++;
- un unico token per gli identificatori, valido per tutti quanti;
- un token per ogni simbolo di punteggiatura.

6.1.3 Il compito dell'analizzatore lessicale

L'obiettivo dell'analalizzatore lessicale è quindi riconoscere i cosiddetti *lessemi*, ossia quelle parti del programma che corrispondono ai token, e ritornarli. Ciascuno di questi, di solito, viene ritornato sotto forma di coppia `<token-name>: <token-value>`, dove:

- `<token-name>` è il nome scelto per denotare quel preciso token; seguendo l'esempio della grammatica precedente, `IDENTIFIER` è un `<token-name>`;
- `<token-value>` è tipicamente un puntatore a una entry della symbol table, in cui si va a salvare tutte le informazioni relative a quel preciso token di tipo[1] `<token-name>`.

6.2 Lessemi e espressioni regolari

Andiamo quindi a capire in che modo la teoria studiata nei due capitoli precedenti entra prepotentemente nell'analisi lessicale.

I lessemi sono estremamente facili da descrivere utilizzando le espressioni regolari: ad esempio, in un linguaggio che prevede un lessema identificatore, il quale è costituito di qualsiasi combinazione di lettere maiuscole e minuscole, quest'ultimo può essere facilmente denotato da un'espressione regolare del tipo:

$$(a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z)^*$$

Questo è molto interessante: se è vero che i lessemi sono perfettamente descrivibili con dei linguaggi regolari, allora vuol dire che possiamo utilizzare quelli in sede di analisi lessicale, senza dover tirare in ballo i più potenti ma decisamente più complessi linguaggi liberi.

Quindi questi lessemi, essendo descritti da delle espressioni regolari, possono anche essere riconosciuti agevolmente da una macchina a stati finiti, come ad esempio quella in figura sotto, che descrive la classe `relop` di token per gli operatori relazionali.

[1] In questo caso la parola "tipo" è chiaramente impropria, ma è molto utile per rendere la natura astratta di classe di token.

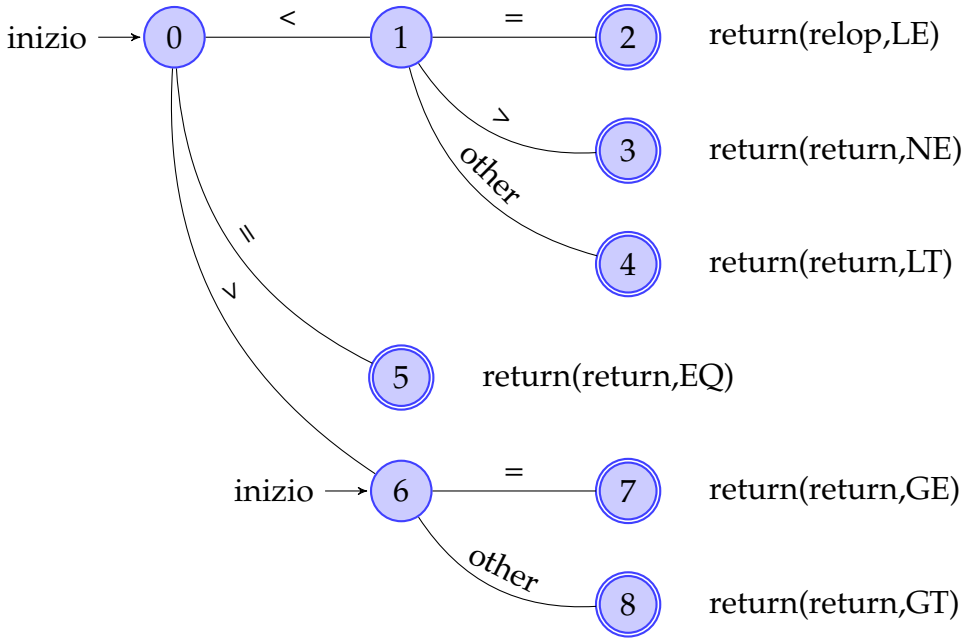


Figura 6.1: Automa che riconosce la classe `relop`

È molto semplice intuire graficamente che ciascun percorso che termini in uno stato finale ritorna un token, sotto forma di coppia, in cui il `token-name` è appunto quello della classe `relop`, mentre il `token-value` indica qual è l'elemento della classe da ritornare.

Retract Possiamo notare che i due stati finali il cui arco entrante è marcato come *other* hanno anche un asterisco: questo indica che, quando consumo quella transizione, devo ricordarmi di tornare indietro di un simbolo, perché quest'ultimo simbolo *other* (cioè che non fa parte di alcun elemento appartenente alla classe `relop`) potrebbe essere un elemento di un token successivo, e se non facessimo quanto detto sopra rischieremmo di saltarla nell'analisi e ottenere un flusso di token incorretto. Questa operazione è detta *retract*.

L'operazione di retract è uno dei tanti elementi legati alla gestione dell'input e del buffer che ci fanno pensare che la macchina a stati è un modello molto vicino al più noto, per noi, automa a stati finiti, ma quanto è profondo questo legame?

6.2.1 Pattern matching basato su NFA

Immaginiamo di avere delle espressioni regolari che denotano il linguaggio dei lessemi che siamo interessati a riconoscere. Si pensi ad esempio all'espressione regolare che denota il lessema `IDENTIFIER`, ossia l'espressione regolare che

denota tutte le possibili combinazioni di lettere maiuscole e minuscole (con le dovute peculiarità di ciascun linguaggio); oppure, si pensi a un'espressione regolare che denoti il lessema degli operatori relazionali, o ancora un'altra che denoti la scrittura dei floating numbers. In sostanza, per ciascuna categoria sintattica abbiamo un'espressione regolare che la denota.

Sappiamo anche che, avendo delle espressioni regolari, per ciascuna di queste possiamo costruire un NFA che riconosce il linguaggio denotato dall'espressione considerata; possiamo a questo punto immaginare di far collidere questi NFA inserendo uno stato iniziale extra e collegandolo a ciascun NFA tramite una ε-transizione, come mostrato in Fig.6.2 sotto. Da notare che nel caso d'uso del compilatore ci sono delle azioni associate agli stati finali (se l'analizzatore legge un assegnamento compie, appunto, le azioni per registrare l'assegnamento).

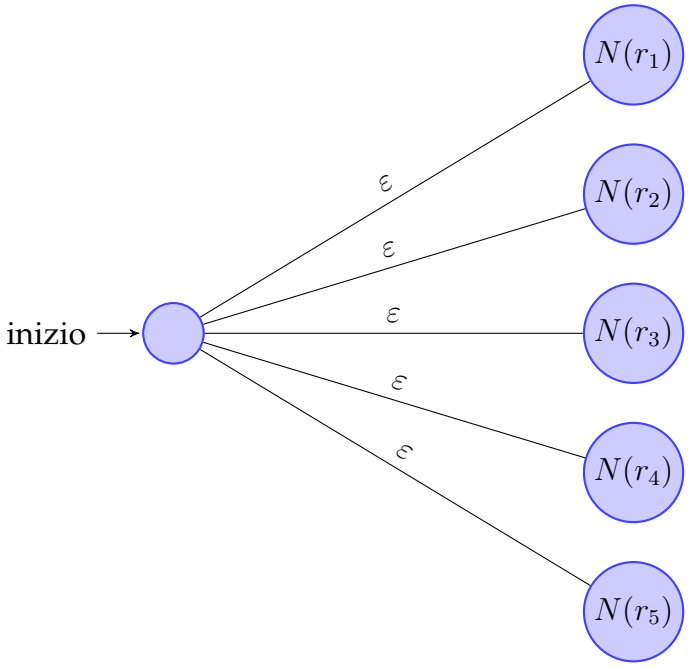


Figura 6.2

Una struttura di questo tipo può riconoscere i linguaggi denotati da tutte le espressioni regolari per cui abbiamo costruito degli NFA.

Adesso dobbiamo pensare come ottimizzare l'uso di una struttura simile per l'analisi lessicale. Potremmo trovarci infatti a gestire problemi di diversa natura, in particolare relativi all'input buffering: ad esempio, come posso fare a sapere se i due caratteri `i` e `f` che ho appena letto stanno a indicare la keyword **`if`** e non un ipotetico identificatore `iffoff`? Detto in maniera informale, come decido

quando fermarmi, quando tornare indietro, quando e se ho letto più di quanto mi serviva?

Il procedimento che seguiamo è il seguente:

1. innanzitutto simuliamo l'NFA creato come sopra descritto;
2. nel caso di ambiguità, ossia di situazioni come quella descritta sopra tra `if` e `iffoff`, la convenzione è di preoseguire la simulazione finché nessun'altra transizione è possibile, solitamente quando incontriamo uno spazio o un `\n`, privilegiando quindi il match più lungo (si parla di cercare il *longest match*);
3. se nell'insieme di stati che abbiamo raggiunto ci sono delle azioni disponibili, allora andremo ad eseguire quella appartenente al longest match, in caso di pareggio ciascuna azione avrà una priorità, e noi eseguiremo quella con più alta priorità;
4. se nessun'azione è disponibile, dobbiamo invece tornare indietro nella sequenza di stati percorsi, fermarci nel primo insieme di stati che presenta almeno uno stato finale e delle azioni associate e cercare di nuovo quella prioritaria; per ognuno di questi passi all'indietro, dobbiamo ricordari di aggiornare il puntatore all'input buffer.

Tutto questo funzionerebbe in maniera analoga se utilizzassimo un DFA: dovremmo semplicemente costruire il DFA a partire dal NFA, dimodoché l'insieme degli stati del DFA sia un sottoinsieme di quello dell'NFA corrispondente.

6.2.2 Generatori di analizzatori lessicali: Flex

Andiamo a parlare quindi di questi generatori. L'idea di creare un generatore di analizzatori lessicali è nata in concomitanza con la prima definizione e implementazione del compilatore del C, e l'idea era di evitare di dover scrivere ogni volta, per ogni programma, un analizzatore lessicale e uno sintattico a mano.

Flex è il primo della sua specie[2] ed è solitamente compreso nelle distribuzioni di C; l'idea è risultata talmente valida che oggigiorno ogni linguaggio possiede un proprio generatore di analizzatore lessiale e sintattico.

Il funzionamento di Flex (ma anche di tutti i generatori di questo tipo) è il seguente:

[2] o meglio, una sua versione più moderna: il primo vero e proprio si chiamava *lex* e quella *f*, aggiunta successivamente, sta per fast.

- andremo a creare un file del tipo `file.l`, il quale sarà l'input del generatore e in cui scriveremo quali sono i pattern che vogliamo riconoscere e quali sono le azioni da compiere in corrispondenza dei vari matches;
- a quel punto, possiamo compilare `file.l` utilizzando il comando `Flex`, e questo ci restituirà un file di nome `lex.yy.c`[3];
- compilando questo `lex.yy.c` con il compilatore `gcc` otteniamo il *lexer*, cioè quel signore che si occupa di gestire l'input buffering, di fare le operazioni di retract e altro ancora.

```
$   Flex file.l
$   gcc lex.yy.c -lfl
$   ./a
```

Si tenga bene a mente che queste tre righe descrivono come utilizzare Flex da solo; tuttavia, quest'ultima è un'eventualità piuttosto rara, dal momento che Flex è nato per essere usato in pipeline con un generatore di analizzatore di analisi sintattica, ma poiché questi sono elementi ancora sconosciuti per noi, al momento non ce ne preoccupiamo.

6.2.3 Struttura del `file.l`

I file con estensione `.l` che diamo da fagocitare a Flex hanno la seguente struttura:

```
...
//(Preambolo)
% { code
% }
// shorthand for patterns
%%
//(Parte centrale)
pattern-1 {action-1};
pattern-2 {action-2};
...
%%
//(Epilogo)
user routines
```

Il contenuto di queste tre macrosezioni è ben determinato, andiamo a vederlo più da vicino:

[3]La ragione di questo nome è puramente convenzionale e deriva dal fatto che, storicamente, il generatore Flex è stato usato in coppia all'analizzatore *Yacc*

Preambolo qui ci andrà del codice C,in particolare le inizializzazioni di variabili, o delle abbreviazioni per indicare dei particolari pattern nelle espressioni regolari che andremo a utilizzare sotto;

Parte centrale è il cuore del file, e si tratta di una lista pattern-azione, dove pattern è un'espressione regolare, e l'azione sarà ciò che dobbiamo compiere qualora dovessimo riconoscere il pattern corrispondente;

Epilogo questa sezione è in realtà facoltativa, il lexer funziona anche senza che vi sia scritto alcunché; tuttavia, quello che potremmo eventualmente trovare e/o scrivere qui dentro sono delle routine definite dall'utente, le quali verranno copiate e incollate dentro al genituro `lex.yy.c`.

6.2.4 Il linguaggio delle espressioni regolari in Flex

Dal momento che tutte le coppie `pattern {azione}` sono scritte in linguaggio `lex`, a differenza di altre parti di `file.l`, è opportuno conoscere questo linguaggio. Di seguito presentiamo i *metacaratteri* di Flex, ossia dei caratteri riservati:

/ \ − * + > ” { } . $ ( ) | % [ ] ^

Andiamo a vedere più da vicino quali sono le regole di matching dei metacaratteri:

. qualsiasi carattere, eccetto il newline;

\n il newline;

***** zero o più copie di un elemento;

+ una o più copie di un elemento;

? zero o una copia di un elemento;

[] denota le classi di caratteri: al posto di $a \mid b \mid \ldots \mid z$, posso scrivere `[a-z]`;

^ inizio di riga, negazione se usato in una classe di caratteri;

$ end of line;

a|b pipe, semantica consueta (i.e. a or b);

() raggruppamenti;

"+" il literal "+" (è necessario usare qusto metodo di escape per poter far riferimento ad un carattere qualunque e non al metacarattere)

{} espressioni regolari scritte nel preambolo

6.3 Esempi di file per Flex

Facciamo un breve riepilogo di quanto abbiamo visto: tutti i file predisposti per Flex si compongono di tre sezioni, ciascuna separata da coppie di %. Nel preambolo si inseriscono delle definizioni come le `define` che servono e i pattern (espressioni regolari per i lessemi che vogliamo inserire). I pattern sono inseriti insieme all'azione nella parte centrale. In fondo si inseriscono le routine dell'utente che vengono copiate nel file `lex.yy.c`: un testo minimale è l'invocazione della routine `yylex()` che se non viene inserita viene chiamata in automatico. I comandi, qualora non si usi Flex con Bison (analizzatore sintattico in pipeline con Flex), sono i seguenti:

```
$   Flex file.l
$   gcc lex.yy.c -lfl
$   ./a
```

file0.l

```
1 %%
2
3 .    printf("hello world!");
4
5 %%
```

In qusto primo esempio vediamo che nel preambolo non sono presenti né delle `define` né del codice; dal momento che non sono presenti routine inserite dall'utente, verrà invocata la procedura `yylex()` all'esecuzione. Basandoci sulla sintassi di Flex vista precedentemente, andiamo ad analizzare la parte centrale costituita da una sola coppia pattern-azione:

- il pattern è dato dalla regular expression composta solo da `.`, il che significa che l'azione seguente farà riferimento a tutti i caratteri incontrati, tranne quello di `\n` (*new line*);
- l'azione è quel `printf("hello world!")`, il che consiste banalmente nello stampare la stringa hello world ogni volta che viene eseguito un match sul pattern.

In sostanza, questo file ci farà sì che, nella lettura, qualsiasi carattere incontriamo, ad eccezione del new line, verrà sostituito da un'occorrenza della stringa `"hello world!"`; quindi, ad esempio:

- a diventerà `hello world!`;
- aa diventerà `hello world!hello world!`.

Possiamo notare anche un'altra cosa molto interessante: solitamente, per poter utilizzare lo standard output nei programmi C, è necessario includere all'interno del file la libreria `stdio.h`, ma nel `file0.l` non vi è nemmeno l'ombra di un inlcude: è lo stesso Flex a occuparsi di aggiungere in un secondo momento tutti i dettagli necessari per la gestione dell'output.

file1.l

```
1  %{
2  %}
3
4  non_white  [^ \t\n]*
5
6  %%
7
8  {non_white}  ECHO;
9  .    ;
10 [\n]    ;
11
12 %%
```

Preambolo Diversamente dall'esempio precedente, in questo `file1.l` abbiamo del contenuto nel preambolo. Andiamo a vederlo nel dettaglio:

1. il primo blocco (righe 1 e 2, in questo caso vuoto) è identificato dal simbolo di apertura `%{` e cui può seguire del codice C;
2. il secondo blocco (righe da 3 a 6), invece, è successivo al simbolo di chiusura `%}` e contiene quelli che sono gli shorthands per i pattern, vale a dire degli alias per delle espressioni regolari.

Rifacendoci sempre alla sintassi di Flex, possiamo analizzare l'espressione regolare definita nel preambolo e arrivare alla conclusione che faccia riferimento a quell'insieme di caratteri (`[ ]`) che si ripetono 0 o più volte (*) e che sono

diversi (^) da ' ', '\t' e '\n'; questo significa che comporteranno un match tutte quelle sequenze di caratteri che sono separate da spazi, tab o new line; più semplicemente, saranno le parole a causare un match.

Parte centrale Come descritto precedentemente, nella parte centrale i pattern sono associati alle relative azioni; in questo caso, le coppie che troviamo sono le seguenti:

- `non_white` comporta l'esecuzione della macro `echo` di Flex, che è equivalente a `printf("\%s", yytext)` in linguaggio C, e stamperà a video esattamente la sequenza di caratteri che ha causato il match;
- un qualunque carattere diverso da `\n (.)` verrà eliminato;
- new line (\n) verrà anch'esso eliminato

Complessivamente, il risultato ottenuto dall'esecuzione di un programma compilato a partire dal file discusso porterà all'eliminazione di ' ', '\t' e '\n' dal testo passato in input e ci farà ottenere ad una sequenza continua di caratteri (ad esempio, una stringa del tipo " `ab c d e f g`" diventerà "`abcdefg`").

file2.l

```
1  %{
2
3  int lineno = 1;
4
5  %}
6
7
8  %%
9
10 ^.*\n    printf ("%4d)\t%s", lineno++, yytext);
11
12 %%
```

Preambolo In questo esempio, invece, troviamo finalmente del codice C nel preambolo; in particolare, troviamo la dichiarazione di un intero chiamato `lineno` ed inizializzato al valore 1.

Parte centrale A seguire abbiamo la parte centrale del file, dove:

- all'espressione regolare `^.*\n`, che identifica tutte quelle sequenze di caratteri che iniziano per un carattere differente da new line ripetuto zero o più volte e finiscono per new line o, più semplicemente, le parole sulla stessa riga
- viene associata l'azione `printf("\%4d)\t\%s", lineno++, yytext)`, che stampa il valore dell'intero lineno seguito dal simbolo) e dal contenuto del buffer `yytext` (che contiene l'input fino ad ora riconosciuto) ed infine esegue il postincremento di lineno.

Quindi, il nostro programma si occuperà di stampare tutte le parole sulla stessa riga anteponendo il numero della riga a cui fa riferimento; in altre parole, se avessimo un testo come il seguente:

> *Autem quo ea. Voluptatum saepe porro. Quibusdam illo eum.*
> *Quia aperiam nesciunt. Qui est voluptate. Aut temporibus perspiciatis.*
> *Repudiandae delectus omnis. Modi earum doloribus. Quis eaque quidem.*

Allora otterremmo in output:

1 *Autem quo ea. Voluptatum saepe porro. Quibusdam illo eum.*
2 *Quia aperiam nesciunt. Qui est voluptate. Aut temporibus perspiciatis.*
3 *Repudiandae delectus omnis. Modi earum doloribus. Quis eaque quidem.*

file3.l

```
1  %{
2
3  int charCount = 0, wordCount = 0, lineCount = 0;
4  %}
5
6  word [^ \t\n]+
7
8  %%
9
10 {word}  {wordCount++; charCount += yyleng; }
11 [\n]    {charCount++; lineCount++; }
12 .        {charCount++; }
13
14 %%
15
```

```
16 int main() {
17   yylex();
18   printf("Characters: %d Words: %d Lines %d\n",charCount, wordCount,
     ↪  lineCount);
19   return 0;
20 }
```

In questo quarto esercizio andiamo a combinare quanto visto precedentemente, aggiungendo anche una routine custom scrivendo del codice ad hoc nell'epilogo del file.

Preambolo Nel preambolo abbiamo la dichiarazione di tre variabili che ci serviranno nelle routine come contatori; inoltre, abbiamo anche una shorthand, chiamata `word`, la cui espressione regolare include tutte quelle sequenze di uno o più caratteri (+) che non contengono ' ', '\t' e '\n.

Parte centrale Nella parte centrale del file abbiamo invece le seguenti associazioni pattern-azione:

- `word` comporta l'esecuzione del seguente pezzo di codice `{wordCount++; charCount += yyleng; }`, dove la variabile `wordCount` viene postincrementata, mentre alla variabile `charCount` viene sommato un numero pari a `yyleng`, che contiene la lunghezza della stringa riconosciuta e, per via dell'espressione regolare alla base della shorthand, equivalente a quella della parola;
- il secondo pattern è `\n`, cioè nel caso in cui si trovi una sequenza di caratteri composta solamente da una new line; l'azione legata è il blocco `{charCount++; lineCount++; }`, dove si postincrementano sia `charCount` che `lineCount`;
- infine, l'ultimo pattern, il punto (`.`), comporta solamente il postincremento della variabile `charCount`.

In sostanza, il programma che stiamo descrivendo con questi pattern si occupa di contare complessivamente quanti caratteri, parole e righe sono state individuate in un dato testo; qualora avessimo un match con una parola, allora avviene un incremento proporzionale al numero di caratteri che costituiscono quella stessa parola e, inoltre, si aggiorna il numero di parole totali che sono contenute nel testo; ogni volta che si trova una new line si incrementano di un'unità il numero di righe e di caratteri e, infine, ogni volta che si trova uno spazio o un tab viene incrementato solamente il numero di caratteri.

file4.l

```
%{

int comma = 0;  /* to check empty entries, rendered by consecutive
 ↪  commas */

%}

%%

[a-zA-Z0-9]+    {printf("<td> "); ECHO; comma = 0;}
","             {if (comma) { printf("<td> </td>"); } else {
 ↪  printf("</td>"); }
                 comma = 1;
                }
\n              {printf("</td> </tr> \n <tr>"); comma = 0;}

%%
int main() {
  printf("<html><body><table border=\"\" cellpadding=\"3\"
   ↪  cellspacing=\"0\"><tbody> \n <tr>");
  yylex();
  printf("</tr>\n</tbody></table>\n");
  printf("</body></html>\n");
  return 0;
}
```

L'ultimo esempio proposto fa uso di quanto discusso fino ad ora per analizzare un `csv` e restituire i dati sotto forma di una tabella in formato `html`; inoltre, potremo osservare un'implementazione del longest match.

Preambolo Nel preambolo viene dichiarata una variabile intera `comma`, che è inizializzata a 0; servirà per controllare le entry vuote nel `csv` passato in input date da virgole consecutive.

Parte centrale Andiamo a vedere da vicino le coppie pattern-azione che troviamo nella parte centrale.

- L'espressione regolare `[a-zA-Z0-9]+`, che indica la sequenza di uno o più caratteri alfanumerici, comporta l'esecuzione di `printf("<td>"); ECHO; comma = 0;`, dove viene creata una nuova colonna per la tabella, viene

inserita la sequenza appena letta e viene impostato il valore dell'intero `comma` a 0.

- La seconda espressione è `","`, che vuol dire che bisogna trovare il carattere che corrisponde alla virgola e che nel `csv` serve per separare i dati, comporta l'esecuzione del blocco di codice

```
{
if (comma) {
    printf("<td> </td>");
} else {
    printf("</td>");
}
    comma = 1;
}
```

  dove viene verificato il numero di virgole incontrate fino a quel punto: se si sono incontrate 0 virgole, allora si chiude semplicemente la colonna precedentemente aperta; se invece si è già incontrata 1 virgola e, consecutivamente, se ne legge un'altra (o comunque le due occorrenze risultano essere separate da sequenze non alfanumeriche), viene creata una colonna vuota, in quanto nel `csv` non vi sono dati rilevanti. In ogni caso alla fine del blocco viene impostato il valore di comma ad 1.

- In ultimo, `\n`, cioè per ogni new line viene eseguito il blocco di codice `{printf("</tr> \n <tr>"); comma = 0;}` dove viene chiusa la riga precedente ed aperta quella successiva.

Epilogo Nell'epilogo troviamo una routine custom dove viene creato l'html di base per creare la tabella e, tra le varie cose, viene anche aperta la prima riga. Dopo aver stampato lo scheletro della struttura viene invocata la procedura `yylex()`, la quale cerca i pattern nel testo; infine, la tabella viene chiusa.

6.3.1 Ulteriori informazioni su Flex

Quando abbiamo discusso del funzionamento dell'analisi lessicale basata sull'utilizzo di NFA o DFA abbiamo anche parlato del fatto che, quando si tenta di riconoscere il pattern di una determinata parola, è possibile che avvengano dei match su più stati finali dell'automa e che quindi si possa avere il dubbio su quale azione eseguire: come si riflette tutto ciò in Flex?

Per risolvere a tale problematica si utilizza la regola del *longest match*: dalla lista di pattern si va a recuperare quello più lungo che ha portato ad un match.

Cosa succede invece se ci sono più pattern che a parità di lunghezza comportano un match (ovvero i due pattern hanno la stessa lunghezza)? In questo caso si sceglie sempre il primo della lista. Da qui si intuisce dunque che anche l'ordine in cui si scrive la sequenza dei pattern ha una sua importanza: un medesimo `file.l` potrebbe non generare lo stesso risultato se si eseguisse uno shuffle dei pattern.

Capitolo 7

Analisi sintattica: parsing top-down

7.1 Il parsing

Il parsing (o analisi sintattica) è quel processo che, data una grammatica $\mathcal{G} = (V, T, S, \mathcal{P})$ e una parola w, ci permette di dire se $w \in \mathcal{L}(\mathcal{G})$ e, se questo è vero, fornire il suo albero di derivazione. Solitamente gli approcci al parsing che vengono presi in considerazione nell'ambito dei linguaggi di programmazione sono due:

- **top-down**: consiste nella costruzione di una derivazione leftmost da uno start symbol della grammatica e quindi procede dalla radice verso le foglie dell'albero di derivazione; a prima vista, si direbbe che sia l'approccio più intuitivo;
- **bottom-up**: consiste invece nella costruzione di una derivazione rightmost (in ordine inverso) della stringa dalle foglie alla radice.

A questo punto è necessario aggiungere che il parsing non si limita a questi due approcci, ma esiste anche in forma più generale utilizzando delle tattiche che vengono impiegate nel caso dei linguaggi naturali. Gli approcci descritti non permettono di considerare tutti i possibili linguaggi liberi, ma solamente delle sottoclassi: per questo si ha un'analisi sintattica estremamente efficiente dal punto di vista computazionale.

7.1.1 Top-Down Parsing

Esempio 1

Sia $w = bd$ e sia $\mathcal{G}$:

$$\begin{aligned} \mathcal{G} : S &\to Ad \mid Bd \\ A &\to a \\ B &\to b \end{aligned}$$

Per verificare se $w \in \mathcal{L}(\mathcal{G})$ con un approccio top-down dobbiamo ottenere una derivazione leftmost a partire dallo start symbol. Ovviamente, si noterà subito che non è possibile scegliere Ad come derivazione iniziale dello start symbol, perché a quel punto l'unica parola ottenibile sarebbe parola ad; dobbiamo invece optare per la seconda derivazione. La derivazione completa ci porta a

$$S \Rightarrow Bd \Rightarrow bd$$

Visto che abbiamo dimostrato che $w \in \mathcal{L}(\mathcal{G})$ e che tale derivazione esiste, allora possiamo fornire il suo derivation tree che, per questo esempio, risulta davvero molto semplice.

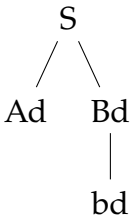


Figura 7.1: Albero di derivazione per esercizio 1

Esempio 2

Sia $w = id + id * id$ e sia $\mathcal{G}$:

$$\begin{aligned} \mathcal{G} : E &\to TE' \\ E' &\to +TE' \mid \varepsilon \\ T &\to FT' \\ T' &\to *FT' \mid \varepsilon \\ F &\to (E) \mid id \end{aligned}$$

Qui non è così intuitivo riuscire a dire se esiste una derivazione leftmost: che cosa mi conviene espandere? Avremo modo di parlare meglio di questo esempio non appena introdurremo il **predictive top-down parsing**.

Esempio 3

Sia $w = cad$ e sia $\mathcal{G}$:

$$S \to cAd$$
$$A \to ab \mid a$$

Nonostante questo esempio sia molto intuitivo, dobbiamo sforzarci di ragionare nei panni dell'algoritmo di parsing: dopo la prima derivazione, infatti, entrambe le opzioni che vengono proposte per poter derivare il non-terminale A sono apparentemente valide in quanto iniziano entrambe per a. Quale delle due dovrei dunque scegliere? Quanto devo continuare l'esecuzione prima di accorgermi se ho fatto una scelta corretta oppure no? Il nostro algoritmo potrebbe anche trovarsi nel caso di dover fare *backtrack*, perché semplicemente non c'era un modo semplice e generale per capire cosa dover scegliere: ovviamente questo tipo di tecnica funziona, ma vogliamo un approccio efficiente e il backtrack, come sappiamo, in questo non ci aiuta.

7.1.2 Predictive Top-Down Parsing

Nel caso del predictive top-down parsing non è mai necessario applicare la tecnica del backtrack, poiché questo fa riferimento a una classe particolare di grammatiche, le quali vengono definite **LL(1) grammars**; vengono così chiamate per via della procedura impiegata per analizzarle, in cui:

- guardiamo le parole da sinistra a destra;
- eseguiamo una produzione leftmost;
- guardiamo un solo simbolo (non-terminale).

Tali grammatiche prevedono una tipologia di parsing per cui non è necessario backtrack, e per di più è **completamente deterministica**.

Queste grammatiche vengono classificate a seconda del grado di determinismo che consentono; all'inizio del corso abbiamo visto la differenza tra le grammatiche senza contesto e contestuali, ma adesso le classi che incontreremo da oggi in poi si differenzieranno esclusivamente per il tipo di analizzatore con cui possiamo riconoscere le parole generate da queste grammatiche.

In questo caso il parsing si basa sul fatto che possiamo costruire una tabella di parsing che ci guida nell'analisi della parola che ci viene data in input; questa strategia ci permette molto efficacemente di dire se la parola appartenga oppure no al linguaggio e, in caso di esito positivo, di costruire la derivazione leftmost richiesta e il conseguente albero di derivazione.

Prendiamo come esempio il caso che abbiamo lasciato in sospeso precedentemente:

$$\begin{aligned}\mathcal{G} : E &\to TE' \\ E' &\to +TE' \mid \varepsilon \\ T &\to FT' \\ T' &\to *FT' \mid \varepsilon \\ F &\to (E) \mid id\end{aligned}$$

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

Tabella 7.1: Tabella del parsing top-down

La tabella è così costruita:

- si ha una **riga** per ogni non-terminale della grammatica;
- ed una **colonna** per ogni terminale della grammatica, a cui si aggiunge il simbolo $ alla parola fornita in input e utilizzato come terminatore
- le entry vuote all'interno della tabella identificano i casi di errore.

Avendo a disposizione la tabella di parsing, la cui costruzione verrà trattata successivamente, è possibile utilizzarla per il parsing top-down predittivo.

Algoritmo per il Predictive Top-Down Parsing

- Input: una stringa w, una tabella M di parsing top-down per la grammatica $\mathcal{G} = (V, T, S, \mathcal{P})$;
- Output:
  - la derivazione leftmost della stringa $w \iff w \in \mathcal{L}(\mathcal{G})$;

– altrimenti *error()*.

Per questo algoritmo utilizzeremo due strutture:

- un buffer che conterrà la parola e che terminerà con un terminatore \$;
- una pila che andrà a contenere simboli terminali e non; viene inizializzato con lo start symbol S e il terminatore \$.

L'obiettivo del nostro gioco è quello di raggiungere il \$ andando a eseguire dei `pop()` dalla pila senza mai incappare in degli `error()`.

Algorithm 8: predictiveTopDownParsing(WORD w, SYMBOL[][] M)

```
1  STACK tmp = Stack()
2  tmp.push($)
3  tmp.push(S)                                  // S è starting symbol di G
4  SYMBOL b = w$.firstSymbol()
5  SYMBOL X = tmp.top()
6  while X ≠ $ do
7      if X == b then
8          tmp.pop()
9          b = w$.nextSymbol()
10     else if isTerminal(X) then
11         error()
12     else if M[X, b] == error then
13         error()
14     else if M[X, b] == X → Y_1, ..., Y_k then
15         output(X → Y_1, ..., Y_k)
16         tmp.pop()
17         for Y_i ∈ Y_k, ..., Y_1 do
18             tmp.push(Y_i)
19     X = tmp.top()
```

Nell'algoritmo di parsing si utilizza la variabile b come primo simbolo delle parola $w\$$ e si inizializza la variabile X con la cima dello stack (il caso base corrisponde ad avere $X = S$). Finché $X \neq \$$ (sostanzialmente finché non ho svuotato completamente la pila), sono dati i seguenti casi:

1. se $X = b$, allora tolgo l'elemento dalla cima della pila e imposto b al simbolo successivo nell'input buffer. Questo corrisponde al caso in cui nella derivazione della costruzione parziale ho ottenuto un match con un terminale nella parola;

2. se invece X è comunque un terminale, allora deve essere che $X \neq b$, ma ciò produce un errore;

3. se invece X è un non-terminale, allora è necessario verificare la tabella di top-down parsing in posizione $M[X, b]$ e possono valere le seguenti:

   - $M[X, b]$ = `error` e allora viene restituito `error();`
   - $M[X, b] = X \rightarrow Y_1 ... Y_k$ e quindi è una produzione che verrà utilizzata per continuare la derivazione: questa viene stampata in output, viene rimosso l'elemento X dalla testa della pila e infine viene inserito il body della produzione in ordine inverso (cioè in modo che Y_1 sia l'elemento in cima allo stack).

Infine, come ultima operazione X viene assegnato all'elemento in cima alla pila e si ripete. In Tab.7.2 possiamo vedere un esempio che fa uso dell'algoritmo appena descritto.

stack	input	output
$\$E$	$id + id * id\$$	$E \to TE'$
$\$E'T$	$id + id * id\$$	$T \to FT'$
$\$E'T'F$	$id + id * id\$$	$F \to id$
$\$E'T'id$	$id + id * id\$$	
$\$E'T'$	$+id * id\$$	$T' \to \varepsilon$
$\$E'$	$+id * id\$$	$E' \to TE'$
$\$E'T+$	$+id * id\$$	
$\$E'T$	$id * id\$$	$T \to FT'$
$\$E'T'F$	$id * id\$$	$F \to id$
$\$E'T'id$	$id * id\$$	
$\$E'T'$	$*id\$$	$T' \to *FT'$
$\$E'T'F*$	$*id\$$	
$\$E'T'F$	$id\$$	$F \to id$
$\$E'T'id$	$id\$$	
$\$E'T'$	$\$$	$T' \to \varepsilon$
$\$E'$	$\$$	$E' \to \varepsilon$
$\$$	$\$$	

Tabella 7.2: Tabella delle strutture a ogni passo

7.1.3 Tabella di parsing

L'algoritmo di Parsing predittivo ha una complessità lineare. Ci chiediamo quindi: come costruiamo le Parsing Table? In quale posizione dobbiamo mettere le produzioni della grammatica per fare sì che l'algoritmo funzioni?

Ricordiamo che la cella $M[A, b]$ della parsing table viene consultata per espandere il non-terminale A sapendo che il prossimo terminale nell'input buffer è b. Andiamo dunque a valorizzare la cella $M[A, b] = A \to \alpha$ se:

- il body della nostra produzione con driver A, è tale per cui esiste una derivazione del tipo $\alpha \Rightarrow^* b\beta$, ossia partendo da α si riesce, con zero o più passi, a ottenere una stringa che inizia per b;
- oppure $\alpha \Rightarrow^* \varepsilon$ ed è possibile avere $S \Rightarrow^* wA\gamma$ (ottenuta da una derivazione di tipo leftmost) con $\gamma \Rightarrow^* b\beta$

Le celle per cui non è possibile inserire una produzione (cioè quelle vuote) verranno valorizzate a `error()`.

Esercizio 1

Sia data la seguente grammatica:

$$\begin{aligned} \mathcal{G} : S &\to aA \mid bB \\ A &\to c \\ B &\to c \end{aligned}$$

Andiamo a vedere come costruire la tabella, aiutandoci dal fatto che è molto semplice ricavare a vista il linguaggio denotato dalla seguente grammatica, che include semplicemente le due parole ac oppure bc. Partiamo dalle produzioni di S:

- se applichiamo la definizione precedente abbiamo che $M[S, a] = S \to aA$, in quanto solamente utilizzando quella produzione riusciremmo ad avere una stringa che comincia per a ($aA \Rightarrow ac$);
- lo stesso ragionamento può essere applicato per $M[S, b] = S \to bB$;
- dal momento che non è possibile creare delle stringhe che cominciano per il terminale c, allora avremo che $M[S, c] = error()$;

A questo punto è possibile passare agli altri due non-terminali della grammatica:

- nel caso di A è possibile notare che esiste una sola produzione che permette di ottenere soltanto c, quindi avremo che $M[A, c] = A \to c$;
- nel secondo caso è possibile applicare la stessa logica, per cui avremo che $M[B, c] = B \to c$.

La tabella finale sarà quindi costruita in questo modo:

	a	b	c	$
S	$S \to aA$	$S \to bB$		
A			$A \to c$	
B			$B \to c$	

Tabella 7.3: Parsing table per esercizio 1

Ricordiamoci che non è possibile che più produzioni possano essere inserite all'interno della stessa cella, in quanto ci stiamo occupando delle grammatiche $LL(1)$, per cui è possibile applicare un parsing di tipo deterministico; se incrociamo una cella che contiene più di una entry, allora non è possibile fare un parsing deterministico.

Quando si arriva ad una error entry vuol dire che sulla testa della pila c'è un non-terminale che, indipendentemente da come decida di espanderlo, non mi porterà mai alla stringa che sto cercando di ottenere.

Esercizio 2

Sia data la seguente grammatica:

$$\begin{aligned} \mathcal{G} : S &\to aAb \\ A &\to \varepsilon \end{aligned}$$

In questo esempio, invece, il linguaggio denotato dalla nostra grammatica è composto solamente dalla parola ab. Per la definizione di α, l'unica cella che ha come non-terminale (nonché start symbol) S è la cella $M[S, a] = S \to aAb$, perché l'unica stringa che è possibile ottenere inizia per a. Da qui è possibile intuire anche che, per ottenere la parola ab, è necessario che $M[A, b] = S \to \varepsilon$; in tutti gli altri casi verrà ritornato un errore, in quanto tale parola non apparterrebbe al linguaggio denotato dalla grammatica.

	a	b	$
S	$S \to aAb$		
A		$A \to \varepsilon$	

Tabella 7.4: Parsing table per esercizio 2

7.2 First(α)

7.2.1 Definizione

Definizione 7.2.1. *Chiamiamo $first(\alpha)$ quell'insieme di terminali che sono posti all'inizio delle stringhe derivate da α.*

Inoltre, se $\alpha \Rightarrow^* \varepsilon$, allora $\varepsilon \in$ first(α): ciò vuol dire che α è un non-terminale annullabile (nullable) e quindi, dopo una serie di passi, sarà pari a ε.

Il concetto di $first$ può essere definito ricorsivamente come segue:

Base casi base:

- $first(\varepsilon) = \{\varepsilon\}$
- $first(a) = \{a\}$

Step $first(A) = \bigcup_{A \to \alpha} first(\alpha)$

Nell'ultimo caso, quello ricorsivo, si ha che $first(A)$ è dato dall'unione di tutti i $first(\alpha)$, dove α è il body di tutte quelle produzioni della grammatica che hanno A come driver.

7.2.2 Algoritmo di calcolo di first(α)

Mostriamo ora un algoritmo che ci permette di calcolare $first(Y_1 \dots Y_n)$ (con $Y_i \in V$), cioè l'insieme dei first per una certa parola considerata.

Algorithm 9: SET firstComputation(WORD α)

```
1  SET first(Y_1 ... Y_n) = ∅
2  j = 1
3  while j ≤ n do
4  |  first(Y_1 ... Y_n).add(first(Y_j) \{ε})
5  |  if ε ∈ first(Y_j) then
6  |  |  j = j +1
7  |  else
8  |  |  break
9  if j = n + 1 then  first(Y_1 ... Y_n).add(ε)
10
```

Andiamo a vedere più da vicino quale idea stiamo seguendo in questo algoritmo. Dopo aver inizializzato l'insieme dei $first(Y_1 \dots Y_n)$, iteriamo su tutti gli elementi $Y_j \in V$ della parola:

- finché non abbiamo esaminato tutti gli $Y_j : 1 \leq j \leq n$, si aggiunge $first(Y_j) \setminus \{\varepsilon\}$ ai $first(Y_1 \dots Y_n)$ e poi si controlla se $\varepsilon \in first(Y_j)$;
- se il controllo restituisce `true`, allora vuol dire che è necessario continuare la ricerca dei first, perché è possibile che in almeno un caso $Y_j = \varepsilon$ e quindi è possibile che nessuno dei $first(Y_j) \in first(Y_1, \dots Y_n)$;
- se il controllo restituisce invece `false`, allora possiamo fermarci, in quanto abbiamo trovato tutti i $first(Y_1 \dots Y_n)$.

L'ultimo controllo serve per verificare se, $\forall\ Y_j \in (Y_1 \dots Y_n)$, non sono in realtà tutti annullabili; in tal caso, sarebbe necessario aggiungere anche ε ai $first(Y_1 \dots Y_n)$, in quanto $(Y_1 \dots Y_n)$ sarebbe annullabile.

7.2.3 Training

Sia data la seguente grammatica:

$$\begin{aligned} \mathcal{G} : E &\to TE' \\ E' &\to +TE' \mid \varepsilon \\ T &\to FT' \\ T' &\to *FT' \mid \varepsilon \\ F &\to (E) \mid id \end{aligned}$$

Osservando la grammatica è possibile, per la definizione espressa precedentemente, affermare con abbastanza semplicità che:

$$\begin{aligned} \{\varepsilon\} \in first(E'), & \quad \text{da } E' \to \varepsilon \\ \{\varepsilon\} \in first(T'), & \quad \text{da } T' \to \varepsilon \\ \{id\} \in first(F), & \quad \text{da } F \ \to id \end{aligned}$$

Tuttavia, l'idea generale è quella di partire da quelle produzioni che necessitano di meno passaggi possibili, cioè quelle che hanno un terminale (o ε) come body, e il cui driver può essere utilizzato per risolvere i $first$ di un'altra produzione.

Per calcolare i $first$ della grammatica proposta ci conviene partire da $first(F)$: per il passo induttivo dobbiamo calcolare $\bigcup_{F \to \alpha} first(\alpha)$, cioè $first((E)) \cup first(id)$. Separiamo dunque il calcolo nelle due componenti:

- nel primo caso analizziamo $first((E))$, per cui dobbiamo procedere al calcolo di una parola composta da tre elementi ($Y_1 =' (', Y_2 = E, Y_3 =')'$, il lettore perdoni l'abuso di notazione nell'utilizzo degli apici) e dunque per prima cosa dobbiamo considerare $first(Y_1)$; tuttavia, $Y_1 =' ($' è un terminale, per cui avremo che $first(Y_1) = \{Y_1\}$: questo significa che $\{Y_1\} \neq \varepsilon$ e che possiamo aggiungerlo a $first(F)$ e interrompere subito l'algoritmo;
- nel secondo caso, invece, ci troviamo di fronte al caso base $first(id) = \{id\}$ e dunque possiamo subito aggiungerlo a $first(F)$.

In questo caso il risultato finale è dunque $first(F) = \{(\} \cup \{id\} = \{(, id\}$; in modo analogo è possibile arrivare alla soluzione per $first(T')$ e $first(E')$.

A questo punto, il miglior modo per procedere con l'algoritmo è, in mancanza di casi base, quello di trovare una produzione del tipo $A \to F\alpha$; in quel modo potremmo sfruttare le conoscenze appena ottenute relativamente a $first(F)$. Applicando dunque l'algoritmo per $first(T)$ ci si accorge che è necessario analizzare la produzione $T \to FT' = Y_1 Y_2$: iniziamo da $first(Y_1)$. Dal momento che che abbiamo già calcolato $first((F)$ e, poiché $\varepsilon \notin first(F)$, possiamo arrestare l'algoritmo e affermare che $first(T) = first(FT') = first(F) = \{(, id\}$. Svolgendo dunque l'esercizio nella sua interezza è possibile ottenere il seguente risultato:

	first
E	{id, (}
E′	$\{\varepsilon, +\}$
T	{id, (}
T′	$\{\varepsilon, *\}$
F	{id, (}

Tabella 7.5: Esercizio sui first

Cosa succede se invece, al posto della produzione $T \to FT'$, diventa $T \to T'F$? Lasciando il resto dell'esercizio invariato, proviamo a calcolare $first(T)$. In questo caso avremmo dovuto analizzare prima i $first(T') = \{\varepsilon, *\}$ e aggiungere $first(T') \setminus \{\varepsilon\} = \{*\}$ a $first(T)$; tuttavia, dal momento che $\varepsilon \in first(T')$, l'algoritmo non può arrestarsi (perchè potrebbe darsi che T' sia nullo), ma è necessario considerare anche i $first(F)$. Come svolto precedenetemente aggiungiamo $first(F) \setminus \{\varepsilon\} = \{id, (\}$ a $first(T)$; essendo che $\varepsilon \notin first(T')$ posso interrompere l'algoritmo e segnare che $first(T) = \{*, id, (\}$

7.3 Follow(A)

7.3.1 Definizione

A differenza dei *first*, che sono definiti per stringhe generiche di terminali e non-terminali, i *follow* sono computati solamente per i non-terminali: scriveremo dunque $follow(A)$.

Definizione 7.3.1. *Con $follow(A)$ indichiamo l'insieme dei terminali che possono seguire A in qualche derivazione.*

I *first* evidenziano quali sono i terminali per cui iniziano le stringhe derivabili da certi elementi (stringhe o non-terminali), i *follow* invece indicano quali sono i terminali che possono seguire.

7.3.2 Algoritmo per il calcolo dei follow(A)

Algorithm 10: SET followComputation(WORD A)

```
1  SET follow(S) = {$}
2  foreach A ≠ S do
3      follow(A) = ∅
4  repeat
5      foreach B → αAβ do
6          if β ≠ ε then
7              follow(A).add(first(B) \{ε})
8          if β = ε ∨ ε ∈ first(β) then
9              follow(A).add(follow(B))
10 until saturation
```

L'algoritmo comincia inizializzando $follow(S) = \$$: dal momento che dallo start symbol si possono generare tutte le possibili parole appartenenti al linguaggio della grammatica, allora possiamo aspettarci di trovare il terminatore di stringa \$ dopo una qualsiasi parola. Negli altri casi, invece, $\forall A$ tale che A è un non-terminale, inizializziamo $follow(A) = \emptyset$.

A questo punto, è necessario sottolineare che, visto che siamo interessati a quei terminali che seguono un particolare non terminale, ci interessano solamente le produzioni della grammatica per cui il generico non terminale A non compare nel driver della produzione, bensì nel body. Per ogni $B \to \alpha A \beta$, si eseguono le sequenti operazioni:

- se $\beta \neq \varepsilon$, allora aggiungiamo $first(\beta) \setminus \{\varepsilon\}$ a $follow(A)$;
- se $\beta = \varepsilon$ or $\varepsilon \in first(\beta)$, allora aggiungiamo $follow(B)$ a $follow(A)$; questo perché ciò che segue B potrà seguire anche A (si tenga presente che se $\beta = \varepsilon$, allora A è la radice dell'ultimo sottoalbero generato da B.

7.3.3 Training: esercizi su first/follow

Esercizio first/follow 1

Utilizziamo sempre la grammatica dell'esempio precedente, che qui riportiamo per comodità del lettore, e andiamone a definire i follow.

$$\begin{aligned}\mathcal{G} : E &\to TE' \\ E' &\to +TE' \mid \varepsilon \\ T &\to FT' \\ T' &\to *FT' \mid \varepsilon \\ F &\to (E) \mid id\end{aligned}$$

1. Dall'algoritmo sappiamo di dover inizializzare $follow(E) = \$$.
2. Iniziamo guardando la prima produzione, cioè $E \to TE'$: se poniamo $A = E'$, allora $\beta = \varepsilon$ e quindi dobbiamo aggiungere i $follow(E)$ (quelli del driver) a $follow(E')$ (quelli del body).
3. Sempre soffermandoci sulla stessa produzione poniamo $A = T$: questo vuol dire che $\beta = E' \neq \varepsilon$ e quindi, ricadendo nel primo caso, aggiungiamo $first(E) \setminus \{\varepsilon\} = \{+\}$ a $follow(T)$; tuttavia, essendo che $\varepsilon \in first((E')$, allora ricadiamo anche nel secondo caso, per cui aggiungiamo $follow(E)$ a $follow(T)$.
4. Passiamo ora alla produzione $E' \to +TE'$: poniamo quindi $A = E'$ e $\beta = \varepsilon$; ricadendo nel secondo caso dovremmo aggiungere $follow(E')$ a $follow(E')$ ma, dato che questo non fornisce informazioni aggiuntive, possiamo scartare questa informazione.
5. Facendo sempre riferimento ad $E' \to +TE'$, esaminiamo il caso per cui $A = T$: visto che $\beta = E'$, tale caso è identico a quello esaminato al punto 3 e porterà all'aggiunta dei medesimi terminali.
6. L'ultimo elemento che non abbiamo preso in considerazione per la produzione corrente è terminale $+$ ma, in quanto terminale, non possiamo calcolarne i follow.
7. Continuiamo ad applicare l'algoritmo come visto nei punti precedenti per calcolare i follow delle produzioni rimanenti.

Un caso che potrebbe risultare interessante riguarda la produzione $F \to (E)$. Poniamo come fatto precedentemente $A = E$: essendo che $\beta = \ ) \neq \varepsilon$, è necessario

aggiungere $first(\beta) \setminus \{\varepsilon\} = \{)\}$ a $follow(E)$; visto che la seconda condizione non è vera, possiamo fermarci ed affermare che $follow(E)) = \{\$,)\}$.

Inoltre, nei punti in cui si deovno aggiungere ai follow fi un non-terminale i follow di un altro si lasciano delle annotazioni che si andranno a risolvere una volta terminata l'analisi di tutte le produzioni. Il risultato finale verrà rappresentato come segue:

	first	computation of follow
E	{id, (}	\$,)
E'	$\{\varepsilon, +\}$	add follow(E)
T	{id, (}	$+$, add follow(E), $+$, add follow(E')
T'	$\{\varepsilon, *\}$	add follow(T)
F	{id, (}	$*$, add follow(T), $*$, add follow(T')

Tabella 7.6: Esercizio sui follow, step intermedio

A questo punto è possibile eliminare mano a mano le dipendenze e le varie ripetizioni presenti all'interno delle computazioni dei *follow*, ottenendo il seguente risultato:

	first	follow
E	{id, (}	$\{\$,)\}$
E'	$\{\varepsilon, +\}$	$\{\$,)\}$
T	{id, (}	$\{\$,), +\}$
T'	$\{\varepsilon, *\}$	$\{\$,), +\}$
F	{id, (}	$\{\$,), +, *\}$

Tabella 7.7: Esercizio sui follow, risultato finale

Come è possibile osservare dalla tabella conclusiva, alcuni passaggi potevano essere tranquillamente evitati (ad esempio l'aggiunta più volte degli stessi terminali): la scelta di evitare tali passaggi oppure di eseguirli per una maggiore sicurezza è a discrezione del lettore.

Esercizio first/follow 2

Prendiamo ora in analisi la seguente grammatica:

$$\begin{aligned}\mathcal{G} : S &\to aABb \\ A &\to Ac \mid d \\ B &\to CD \\ C &\to e \mid \varepsilon \\ D &\to f \mid \varepsilon\end{aligned}$$

Questa volta ripassiamo anche il calcolo dei first.

1. Partiamo da S: ricordiamoci che per calcolare i *first* di un non-terminale dobbiamo vedere i *first* di tutte le sue produzioni; in particolare, so che qualunque stringa derivata da S inizierà con a (da $S \to aABb$), e siccome a è un terminale ed è diverso da ε, non proseguo oltre nella ricerca di *first* per S;

2. per calcolare i *first* di A, invece, ho due produzioni da vagliare: la prima (che derivo da $A \to Ac$) mi dice che i *first* di A contengono anche i *first* dello stesso A (non aggiunge informazioni), la seconda mi dice che d può essere un *first* di A, quindi scrivo $\{d\}$;

3. per calcolare i *first* di B devo conoscere quelli di C e D, e dunque parto da C: in questo caso ottengo semplicemente che i *first* sono $\{e\}$ e possibilmente $\{\varepsilon\}$, quindi per C scrivo $\{e, \varepsilon\}$;

4. in modo simile a C posso facilmente ricavare che i *first* di D sono $\{f, \varepsilon\}$;

5. tornando ora ad analizzare B posso dire che i suoi *first* sono $first(C) \setminus \{\varepsilon\}$, ma siccome C contiene ε tra i suoi *first*, allora devo aggiungere $first(D)$ a $first(B)$;

6. infine, notando che anche $first(D)$ contiene ε, posso annoverare in $first(B)$ anche ε stesso.

Ora che abbiamo terminato la nostra ricerca dei *first* possiamo ricavare la tabella risolutiva dell'esercizio, che si presenta come in Tab. 7.8.

	first	follow
S	{a}	
A	{d}	
B	{e, f, ε}	
C	{e, ε}	
D	{f, ε}	

Tabella 7.8: Esercizio 7.3.3 su first/follow, step 1

Passiamo ora gaiamente al prossimo step, ovvero il calcolo dei vari *follow* seguendo l'algoritmo indicato in Alg. 10.

Per comodità notazionale, dato che nelle produzioni dell'esercizio compare il non-terminale A, scriveremo X per indicare la A dell'algoritmo per il calcolo dei *follow* (assumiamo quindi che ogni A nell'algoritmo venga sostituita da X; quindi, ad esempio, $\alpha A \beta$ diventa $\alpha X \beta$).

1. la fase di inizializzazione del calcolo dei *follow* vuole che lo start symbol abbia come *follow* il terminatore di stringa , per le ragioni che abbiamo già visto, quindi poniamo $follow(S) = \$$;
2. cominciamo l'analisi da $S \rightarrow aABb$:
   (a) osservando l'algoritmo, cerchiamo un match possibile per $\alpha X \beta$; partiamo con il considerare $X = A$ e di conseguenza $\beta = Bb$;
   (b) in questo caso abbiamo che $\beta \neq \varepsilon$, quindi dobbiamo aggiungere i $first(\beta)$ ai $follow(A)$;
   (c) i *first* di β in questo caso sono i *first* di Bb, quindi $\{e, f, b\}$ (notare che non sono $first(B)$ ma $first(Bb)$); come indicato nella procedura, li aggiungo a $follow(A)$;
   (d) β è diverso da ε ed il suo first non contiene ε, quindi il secondo `if` non si applica e terminiamo questo branch;
   (e) ora dobbiamo analizzare il caso in cui, studiando la produzione di S, assumiamo $X = B$ e quindi $\beta = b$;
   (f) in questo caso ho che devo aggiungere a $follow(B)$ i $first(b)$, ovvero $\{b\}$;
   (g) β è diverso da ε ed inoltre $\varepsilon \notin first(\beta)$, quindi termino anche questo branch.

3. Ho concluso quindi l'analisi delle produzioni di S e passo quindi ad analizzare la produzione di A:

   (a) parto da $A \to Ac$: in questo caso non ho altra scelta che considerare $X = A$, quindi $\beta = c$: ricadentdo nel primo `if` dell'algoritmo, aggiungo $first(c) \setminus \{\varepsilon\}$ (ovvero $\{c\}$) ai $follow(A)$;

   (b) chiudo quindi il branch dato che il secondo `if` non si applica;

   (c) considero ora la seconda produzione di A, ovvero $A \to d$; tuttavia, questa non è nella forma $\alpha X \beta$ e quindi non ci dà informazioni, per cui posso passare oltre; ho terminato le produzioni di A.

4. Passiamo all'analisi della produzione $B \to CD$:

   (a) considero $X = C$ e ottengo quindi $\beta = D$; dato che $\beta \neq \varepsilon$ aggiungo ai $follow(C)$ i $first(D) \setminus \{\varepsilon\} = \{f\}$;

   (b) però, dal momento che $\varepsilon \in first(\beta)$, entro anche nel secondo `if` e aggiungo i $follow(B)$ ai $follow(C)$; visto che non posso eseguire questa operazione fino a quando non avrò esaminato tutte le altre produzioni, lascio un placeholder nella tabella. Avendo raggiunto la fine del ciclo `while` chiudo qui il branch;

   (c) ora considero la seconda opzione, ovvero $X = D$, quindi $\beta = \varepsilon$;

   (d) in questo caso andiamo direttamente nel secondo `if` che ci dice di aggiungere i $follow(B)$ ai $follow(D)$; nella tabella, quindi, indico un altro placeholder;

   (e) abbiamo terminato le possibili interpretazioni delle produzioni per la derivazione $B \to CD$.

5. Le produzioni di C, come quelle di D, non sono del tipo $\alpha X \beta$, dato che non contengono caratteri non-terminali; queste produzioni non ci danno informazioni sui *follow* e l'algoritmo ci dice pertanto di ignorarle;

6. a questo punto abbiamo terminato le produzioni da analizzare, non ci resta altro che risolvere i placeholder che abbiamo lasciato nella tabella durante i passi precedenti.

Osserviamo in Tab.7.9 come risulta la tabella prima della sostituzione dei placeholder

	first	follow
S	{a}	{$}
A	{d}	{e, f, b, c}
B	{e, f, ε}	{b}
C	{e, ε}	{f, follow(B)}
D	{f, ε}	{follow(B)}

Tabella 7.9: Esercizio 7.3.3 su first/follow con i placeholder

Mentre in Tab.7.10 si può osservare il risultato finale, con sostituzione dei placeholder.

	first	follow
S	{a}	{$}
A	{d}	{e, f, b, c}
B	{e, f, ε}	{b}
C	{e, ε}	{f, b}
D	{f, ε}	{b}

Tabella 7.10: Esercizio 7.3.3 su first/follow una volta sostituiti i placeholder

Esercizio first/follow 3

Passiamo ora ad un'altro esercizio dello stesso tipo, sviluppato a partire dalla seguente grammatica:

$$\begin{aligned} \mathcal{G} : S &\rightarrow aA \mid bBc \\ A &\rightarrow Bd \mid Cc \\ B &\rightarrow e \mid \varepsilon \\ C &\rightarrow f \mid \varepsilon \end{aligned}$$

Questa volta saremo più spigliati con la risoluzione dei *first*, ma li scriveremo comunque, dato che dobbiamo tener ben presente questa massima:

> "I *first* ed i *follow* li dovete sapere bene perché noi, con questi, ci faremo gli spaghetti." Paola Quaglia

Andiamo quindi, senza ulteriori indugi, a risolvere i *first* dei vari non-terminali.

1. Per S inseriamo $\{a, b\}$ e terminiamo subito, dato che entrambi sono terminali;
2. per A dobbiamo conoscere i *first* di B e C;
3. per B abbiamo che i *first* sono $\{e, \varepsilon\}$;
4. per C abbiamo che i *first* sono $\{f, \varepsilon\}$;
5. infine torniamo a risolvere A:
   - analizziamo $A \to Bd$: inseriamo $first(B) \setminus \{\varepsilon\}$, ovvero inseriamo $\{e\}$;
   - siccome $\varepsilon \in first(B)$, aggiungiamo anche i *first* di b, che sono proprio $\{b\}$;
   - analizziam poi $A \to Cc$;
   - esattamente come per $A \to Bd$, abbiamo che i *first* sono $\{f, c\}$;

Una volta inseriti i first nella tabella ci troveremo nella situazione rappresentata in Tab.7.11.

	first	follow
S	{a, b}	
A	{e, d, f, c}	
B	{e, ε}	
C	{f, ε}	

Tabella 7.11: Esercizio 7.3.3 su first/follow, step 1

Ora è il momento di tuffarci a capofitto nel calcolo dei *follow*; anche in questo caso tenterò di matentere la spiegazione concisa ma completa e, per comodità notazionale, dato che nelle produzioni dell'esercizio compare il non-terminale A, anche in quest'esercizio scriveremo X per indicare la A dell'algoritmo per il calcolo del *follow* (come già fatto per il caso precedente).

Prima di affogare nei calcoli, è interessante osservare più da vicino qual è l'idea sottesa all'inserimento di \$ in follow(S) come primo passo.

Questa azione risulta intuitiva se si pensa che $follow(Z)$ rappresenta quello che io mi aspetto di poter trovare dopo quello che derivo da un certo simbolo non-terminale Z, con un certo numero di passi; quindi si capisce che, essendo S l'origine di tutte le parole generate da una certa grammatica, mi aspetto che, una volta analizzato tutto ciò che è generato da S, troverò il terminatore di stringa, ovvero proprio \$.
Bene, ora siamo pronti a fare a pugni coi calcoli.

1. Partiamo con l'analizzare le produzioni di S, iniziando con $S \to aA$;
   (a) possiamo considerare solo $X = A$, quindi per forza di cose avremo che $\beta = \varepsilon$;
   (b) passiamo subito al secondo `if`, che ci dice di aggiungere i $follow(S)$ ai $follow(A)$; lasciamo quindi un placeholder nella tabella e proseguiamo.
2. Analizziamo ora $S \to bBc$;
   (a) in questo caso siamo obbligati a scegliere $X = B$ e quindi avremo che $\beta = c$;
   (b) passiamo dentro al primo `if` e aggiungiamo $first(c) = \{c\}$ a $follow(B)$;
   (c) il secondo `if` non si applica, quindi terminiamo.
3. Analizziamo ora $A \to Bb$;
   (a) dobbiamo scegliere $X = B$, quindi abbiamo che $\beta = d$;
   (b) anche questa volta soddisfiamo la condizione del primo `if` ed aggiungiamo $first(d) = \{d\}$ a $follow(B)$;
   (c) come nel caso precedente, il secondo `if` non si applica: terminiamo il branch.
4. Analizziamo $A \to Cc$;
   (a) in questo caso $X = C$, quindi troviamo che $\beta = c$;
   (b) come per il caso appena visto, aggiungiamo $first(c) = \{c\}$ a $follow(C)$ e chiudiamo il branch.
5. Infine, tutte le rimanenti produzioni non sono nella forma $\alpha X \beta$, quindi possiamo ragionevolmente dire di aver terminato l'analisi;
6. l'ultimo passo che rimane da svolgere è la sostituzione dei placeholder.

Osserviamo quindi in Tab.7.12, come risulta la tabella dell' Es.7.3.3 prima della sostituzione dei placeholder.

	first	follow
S	{a, b}	{$}
A	{e, d, f, c}	{follow(S)}
B	{e, ε}	{c, d}
C	{f, ε}	{c}

Tabella 7.12: Esercizio 7.3.3 su first/follow con i placeholder

Quindi, in Tab.7.13, ammiriamo la soluzione finale dell' Es.7.3.3, con sostituzione dei placeholder.

	first	follow
S	{a, b}	{$}
A	{e, d, f, c}	{$}
B	{e, ε}	{c, d}
C	{f, ε}	{c}

Tabella 7.13: Esercizio 7.3.3 su first/follow una volta sostituiti i placeholder

7.4 Costruzione una tabella di parsing top-down

Ora che ci siamo esercitati con i meccanismi necessari, possiamo tornare ad occuparci della nostra preoccupazione principale, ovvero come costruire una tabella di parsing.

7.4.1 Algoritmo per la costruzione di una parsing table

Rappresentato in Alg. 11 si può osservare l'algoritmo che, data in input una grammatica $\mathcal{G}$, ritorna la parsing table propria di quella grammatica.

Algorithm 11: TABLE ParsingTableComputation(GRAMMAR $\mathcal{G}$)

```
1 foreach A → α ∈ P do
2     foreach b ∈ (first(α)\{ε}) do
3         add A → α to M[A, b]
4     if ε ∈ first(α) then
5         foreach x ∈ follow(A) do
6             add A → α to M[A, x]
7 set to error() all the empty entries
```

Ricordiamo brevemente che le tabelle di parsing top-down ci servono per verificare se, data una certa parola, questa può essere derivata tramite derivazione leftmost da una data grammatica.

L'algoritmo prevede di scorrere tutte le produzioni $A \to \alpha$ presenti nella grammatica e, per ognuna di queste:

1. aggiungere $A \to \alpha$ in $M[A, b]$ per ogni $b \in (first(\alpha) \setminus \{\varepsilon\})$;

2. se $\varepsilon \in first(\alpha)$, aggiungere $A \to \alpha$ a $M[A, x]$ per tutti gli $x \in follow(A)$.

Una volta terminato questo ciclo, si va a settare il valore $error()$ in tutte le entry della tabella ancora vuote. Alcune note interessanti:

- si osservi che $follow(A)$ può contenere il simbolo \$, ed è questo il motivo per cui nel punto due usiamo x invece che b; infatti, mentre b indica terminali, x serve proprio a far notare che potrebbe esserci \$, che non è un terminale della grammatica, ma il carattere segnalatore della terminazione di una parola;
- a livello grafico caselle di errore saranno lasciate vuote, mentre nell'implementazione vera e propria queste vengono fatte puntare tutte a una routine di errore;
- è possibile finire con l'avere con due entry in una stessa cella; tuttavia, le grammatiche che compongono tabelle con questa forma non sono deterministiche e non appartengono alla classe $LL(1)$, di cui ci stiamo attualmente occupando;
- le caselle con entry multiple nella tabella di parsing si dicono *entry multiple-defined*.

7.4.2 Applicazioni

Viene proposta ora come esempio al lettore questa grammatica.

$$\begin{aligned} \mathcal{G} : E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid id \end{aligned} \tag{7.1}$$

Saprebbe dire il lettore se la grammatica qui presentata appartiene alla classe LL(1)?

Mentre il lettore riflette su questo quesito può leggere un'altra importante citazione del nostro punto di riferimento, Paola Quaglia, la quale riflette sul senso di incomunicabilità che attraversa i tempi moderni, quel conflitto generazionale di cui tutti, in certo momento della nostra vita e in un certo schieramento, siamo stati attori.

> "NWY5YTliMmVjMmRhNiwyOS8xMC8yMDIwIDExOjUxLGthbHQxMA==
> kaltura ci parla cosi' :-("

Torniamo a noi: la risposta alla domanda posta in precedenza è che la grammatica in 7.1 non appartiene alla classe $LL(1)$; ciò si può dimostrare creandone la parsing table per tale grammatica.

Si può notare di fatto come $first(E) = first(T) = first(F) = \{(, id\}$. Di conseguenza, se seguiamo i passi dell'Alg. 11 per la creazione di una parsing table, otteniamo una situazione in cui la casella $M[E, id]$ contiene sia $E \to E + T$ che $E \to T$: ci troviamo nel caso di una *entry multiply-defined*. Ma su questa grammatica abbiamo anche altre cose da dire.

7.5 Grammatiche con ricorsione sinistra

La grammatica vista sopra 7.1 presenta anche una proprietà (o difetto) molto interessante, chiamata ricorsione sinistra:

Definizione 7.5.1. *Una grammatica presenta ricorsione sinistra (left recursive grammar) se, per qualche A e α, $A \Rightarrow^* A\alpha$*

Ad esempio, consideriamo la seguente grammatica:

$$S \to B \mid a$$
$$B \to Sa \mid b$$

Questa grammatica è ricorsiva a sinistra, perché possiamo ottenere una derivazione del tipo: $S \Rightarrow B \Rightarrow Sa$.

7.5.1 Grammatiche con ricorsione sinistra immediata

Se osserviamo la grammatica 7.1, ci rendiamo conto immediatamente che presenta ricorsione sinistra, dato che possiamo espandere un numero indefinito di volte la produzione $E \to E + T$ ed ottenere una sequenza di derivazioni come la seguente:

$$E \Rightarrow E + T \Rightarrow E + T + T \Rightarrow E + T + T + T \Rightarrow \dots$$

Ma non finisce qui: infatti, questa grammatica presenta, per la precisione, la caratteristica di ricorsione immediata sinistra (*immediately left recursive grammar*), ovvero presenta una produzione in forma $A \to A\alpha$.

Presentiamo ora un lemma sulle grammatiche con ricorsione sinistra.

Lemma 7.5.1. *Se una grammatica $\mathcal{G}$ presenta ricorsione sinistra, immediata o no che sia, allora la tal grammatica $\mathcal{G}$ <u>non</u> appartiene alla classe LL(1).*

Una volta venuti a conoscenza di questo lemma, è lecito chiedersi se tali grammatiche proprio non siano riducibili a grammatiche di tipo $LL(1)$ per fare in modo da poterle analizzare in maniera deterministica (e di conseguenza più efficiente).

7.5.2 Eliminazione della ricorsione sinistra immediata

In molti casi è effettivamente possibile eliminare la ricorsione sinistra. Proviamo a capire l'intuizione da seguire per ottenere una grammatica $LL(1)$ da una che presenta ricorsione sinistra; per farlo, aiutiamoci tentando di risolvere tale problema per la seguente grammatica d'esempio:

$$A \to A\alpha \mid \beta \text{ con } \alpha \neq \varepsilon \wedge \beta \neq A\gamma \tag{7.2}$$

Di fatto, se valesse $\alpha = \varepsilon$, allora la grammatica sarebbe solo scritta male e non sarebbe quindi left recursive. Il nostro obbiettivo, ricordiamolo, è quello di ottenere una grammatica che ci consente di avere lo stesso linguaggio della precedente, ma che tuttavia non presenta ricorsione sinistra.

Intuizione

Per aiutarci a trovare un'idea, possiamo tracciare un albero piuttosto grezzo che rappresenti, più o meno, quello che sta succedendo se cerchiamo di derivare lo schema 7.2:

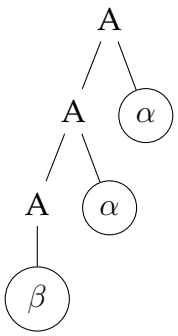


Figura 7.2: Rappresentazione ad albero del problema

Questa figura non è un vero e proprio albero di derivazione, ma ci permette di capire una cosa: quello che noi vogliamo fare è modificare la struttura dell'albero, quindi le produzioni della grammatica, e al contempo mantenere inalterata la frontiera dell'albero. Potremmo ottenere questo risultato aggiungendo un nuovo non-terminale alla grammatica, in modo da eliminare l'elemento ricorsivo della produzione; idealmente, vorremmo ottenere un risultato di questo tipo:

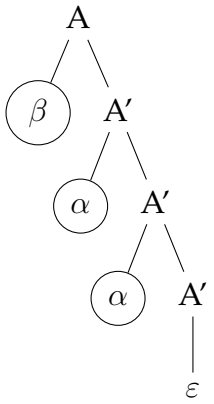


Figura 7.3: Rappresentazione della nostra intuizione

Soluzione formale

A questo punto possiamo scrivere formalmente la nostra strategia per l'eliminazione della ricorsione sinistra immediata: data una produzione del tipo:

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

dove $\alpha \neq \varepsilon$ e $\beta \neq A\gamma$, possiamo riscriverla equivalentemente come segue:

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

dove A' è un non-terminale *fresh* per la grammatica di riferimento, questo perché altrimenti potrebbe interferire con la derivazione di alcune parole nel linguaggio generato dalla stessa grammatica.

Formulazione generale

Riformuliamo la precedente strategia, in modo da astrarla al caso più generale possibile rispetto ai body di una certa produzione.

Per eliminare la ricorsione sinistra immediata di una produzione e mantenerne inalterate le eventuali derivazioni, devo considerare la sua forma:

$$A \to A\alpha_1 \mid \cdots \mid A\alpha_n \mid \beta_1 \mid \cdots \mid \beta_k \tag{7.3}$$

dove $\alpha_j \neq \varepsilon \quad \forall j : 1 \leq j \leq n$ e anche $\beta_i \neq A\gamma_i \quad \forall i : 1 \leq i \leq k$, con la seguente forma:

$$A \to \beta_1 A' \mid \cdots \mid \beta_k A' \tag{7.4}$$
$$A' \to \alpha_1 A' \mid \cdots \mid \alpha_n A' \mid \varepsilon$$

dove $A' \notin \mathcal{A} \setminus T$, cioè è un non-terminale *fresh*.

Andiamo adesso a vedere come (e se) è possibile elminare la ricorsione sinistra anche quando non è immediata.

7.5.3 Eliminazione della ricorsione sinistra

Qui il problema si fa più spinoso: possiamo già anticipare che non sarà un'operazione possibile in tutte le grammatiche e per cui c'è addirittura il rischio che, l'eliminazione della ricorsione sinistra per un certo non-terminale, causi la sua comparsa per un altro non-terminale.

In ogni caso, l'idea è di ridurre gli steps della derivazione $A \Rightarrow^* A\alpha$, in modo da ottenere una produzione che presenta ricorsione sinistra immediata ed applicare la tecnica vista in 7.5.2.

Consideriamo la seguente grammatica:

$$A \to Ba \mid b$$
$$B \to Bc \mid Ad \mid b$$

Notiamo subito che abbiamo una ricorsione sinistra immediata su B, ma abbiamo anche una ricorsione sinistra su A, attraverso $A \Rightarrow Ba \Rightarrow Ada$.

Ci eravamo detti di tentare di ridurre gli steps di derivazione; per cui, quello che facciamo adesso è sostituire i non-terminali coi body delle loro produzioni, ad esempio:

$B \to Ad$ diventerà $B \to Bad \mid bd$

Mantenendo inalterate le altre produzioni, ottengo la seguente grammatica:

$$A \to Ba \mid b$$
$$B \to Bc \mid Bad \mid bd \mid b$$

A questo punto diventa molto semplice eliminare la ricorsione immediata di B con lo stesso metodo visto in precedenza:

$$A \to Ba \mid b$$
$$B \to bdB' \mid bB'$$
$$B' \to cB' \mid adB' \mid \varepsilon$$

7.5.4 Eliminazione di ricorsione sinistra: efficacia

Fermiamoci un attimo e facciamo il punto della situazione. Prima di questa ampia digressione sulle grammatiche con ricorsione sinistra stavamo parlando di top-down parsing, giusto? Bramavamo una tabella di parsing senza entries con definizioni multiple, perché questo avrebbe reso il nostro parsing deterministico. Sapevamo che, fintantoché andavamo a considerare delle grammatiche $LL(1)$, avremmo potuto dormire sogni tranquilli; ma il lemma 7.5.1 ci ha detto che le grammatiche con ricorsioni sinistra (sia immediata, sia non immediata) non sono $LL(1)$, per cui abbiamo iniziato a chiederci se fosse possibile eliminare in qualche modo questa proprietà. Ebbene, dopo tanto tempo speso nell'impresa, ci chiediamo: dopo aver eliminato tutte le ricorsioni, abbiamo certezza che la grammatica ottenuta sia $LL(1)$?

Consideriamo la grammatica delle espressioni aritmetiche, visto che siamo partiti da quella; proviamo ad applicare le nostre regole di eliminazione e vediamo cosa succede.

$$\begin{aligned} \mathcal{G} : E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid id \end{aligned}$$

E il risultato finale è proprio lei, la grammatica che abbiamo visto a 7.3.3 nel calcolo dei *first/follow*! Contenti? Fossi in voi, me la tatuerei da qualche parte per evitare che, dopo averla trovata all'esame senza saper rispondere, infesti i vostri sogni per il resto della vostra vita.

$$\begin{aligned} \mathcal{G}' : E &\to TE' \\ E' &\to +TE' \mid \varepsilon \\ T &\to FT' \\ T' &\to *FT' \mid \varepsilon \\ F &\to (E) \mid id \end{aligned}$$

Per di più sappiamo questa grammatica è $LL(1)$, possiamo dunque ritenerci soddisfatti? L'algoritmo di eliminazione della ricorsione sinistra garantisce effettivamente di ottenere grammatiche $LL(1)$?

Per evitare di cadere in ragionamenti induttivi fallaci del tipo

$$\text{uh, in questo caso funziona} \implies \text{funzionerà in qualsiasi caso, no?}$$

andiamo a vedere un altro caso, considerando una grammatica che genera comunque il linguaggio delle espressioni aritmetiche ma, a differenza della precedente, è ambigua:

$$\mathcal{G} : E \to E + E \mid E * E \mid (E) \mid id$$

	$($	$)$	id	$+$	$*$	\$
E	$E \to (E)E'$		$E \to id\ E'$			
E'		$E' \to \varepsilon$		$E' \to +EE'$, $E' \to \varepsilon$	$E' \to *EE'$, $E' \to \varepsilon$	$E' \to \varepsilon$

Tabella 7.14: Tabella di parsing LL(1) per $\mathcal{G}'$

Eseguendo la nostra procedura otteniamo la seguente grammatica:

$$\begin{aligned} \mathcal{G}' : E &\to (E)E' \mid idE' \\ E' &\to +EE' \mid *EE' \mid \varepsilon \end{aligned}$$

Questa grammatica è $LL(1)$? Sfortunatamente, se costruiamo la tabella di parsing (Tab.7.14) per questa grammatica, troviamo che le celle $[E', +]$ e $[E', *]$ contengono due produzioni:

- $E' \to +EE'$ (o $E' \to *EE'$);
- $E' \to \varepsilon$.

Ne abbiamo un ulteriore conferma se andiamo a calcolare i first/follow per E e E':

	first	follow
E	$\{(, id\}$	$\{\$, +, *,)\}$
E′	$\{+, *, \varepsilon\}$	$\{\$, +, *,)\}$

Tabella 7.15: Tabella con i first/follow per E e E'

Conclusione Nota bene: non dobbiamo cadere nell'errore di supporre che, magari, la nostra procedura funzioni a meno che non sia lanciata su grammatiche ambigue, perché non ne abbiamo fornito alcuna prova. Da questo esempio possiamo solamente concludere, a malincuore, che la procedura di eliminazione della ricorsione sinistra *non garantisce* di ottenere delle grammatiche $LL(1)$.

7.5.5 Eliminare la ricorsione sinistra elimina l'ambiguità?

Possiamo però chiederci: la grammatica $\mathcal{G}'$ che abbiamo ottenuto è ancora ambigua, oppure applicando la nostra procedura ne abbiamo eliminato l'ambiguità?

Ricordiamo che l'ambiguità della grammatica di partenza risiedeva nell'impossibilità di esprimere l'associatività degli operatori e la precedenza dell'uno sull'altro [1].

Sfortunatamente, riusciamo a trovare senza troppi problemi due derivazioni rightmost che ci portano a ottenere la medesima stringa $id + id * id$:

$$\begin{aligned} E &\Rightarrow idE' \Rightarrow id + EE' \Rightarrow id + E * EE' \Rightarrow id + E * E \\ &\Rightarrow id + E * idE' \Rightarrow id + E * id \Rightarrow id + idE' * id \\ &\Rightarrow id + id * id \end{aligned}$$

$$\begin{aligned} E &\Rightarrow idE' \Rightarrow id + EE' \Rightarrow id + E \Rightarrow id + idE' \\ &\Rightarrow id + id * EE' \Rightarrow id + id * E \Rightarrow id + id * idE' \\ &\Rightarrow id + id * id \end{aligned}$$

Possiamo convincerci di questo risultato anche seguendo un altro procedimento, ossia tracciando un albero di derivazione parziale per la nostra grammatica:

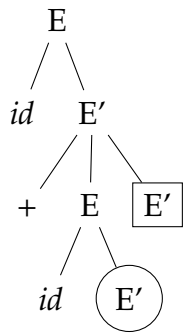


Figura 7.4: Albero di derivazione parziale per $\mathcal{G}'$

A questo punto notiamo subito che, per ottenere la parola $id + id * id$, possiamo completare questo albero parziale sostituendo i nodi marcati con i seguenti sottoalberi:

[1] Si riveda a 2.3.2 in caso servisse un ripasso.

Figura 7.5: Completamento di 7.4, versione 1

Ma, allo stesso identico modo, potremmo completare lo stesso albero parzial e ottenere la stessa identica parola sostituendo gli stessi nodi con questi due diversi sottoalberi:

Figura 7.6: Completamento di 7.4, versione 2

Conclusione Anche qui, non possiamo fare altro che concludere che applicare la procedura di eliminazione della ricorsione sinistra su una grammatica ambigua *non* ne elimina l'ambguità.

7.6 Fattorizzabilità Sinistra

Consideriamo ora la grammatica $\mathcal{G} : S \to aSb \mid ab$, che ormai ben conosciamo: questa grammatica ci permette di denotare, ad esempio, un linguaggio composto da parentesi annidate. Quello che non sapevamo è che non è una grammatica $LL(1)$: ce ne possiamo accorgere calcolandone i $first(S)$. Dal momento che

l'unico elemento appartenente a questo insieme, nel caso delle due produzioni, risulta infatti essere il non-terminale a, questo vuol dire che, applicando l'algoritmo per la generazione della tabella di parsing predittivo, ci troveremo con due produzioni nella posizione $M[S, a]$.

Il punto è che $\mathcal{G}$ può essere **fattorizzata a sinistra**; in generale, una grammatica è fattorizzabile a sinistra quando:

- ci sono almeno due produzioni che hanno lo stesso non-terminale come driver;
- possiedono un prefisso comune nel body.

Nel caso precedente, ad esempio, vi sono due produzioni che hanno S come driver e a come prefisso. Rispetto a tutto ciò, analogmente alle grammatiche con ricorsione a sinsitra, abbiamo un lemma.

Lemma 7.6.1. *Se la grammatica $\mathcal{G}$ può essere fattorizzata a sinistra, allora sicuramente $\mathcal{G}$ non è $LL(1)$.*

7.6.1 Strategia

Non possiamo quindi fare a meno di chiederci, di nuovo, se possiamo modificare le produzioni della grammatica in modo da non avere più produzioni per un singolo prefisso, ed evitare quindi di trovarci una tabella di parsing con entries multiple defined.

L'idea che sta alla base della strategia che si utilizza per la fattorizzazione sinistra è quella di rimandare il più possibile la scelta delle produzioni con lo stesso prefisso. Quindi, data una grammatica $\mathcal{G}$ che ha due produzioni con lo stesso driver e con un prefisso comune, quello che si fa è rimpiazzare la produzione iniziale:

$$A \to \alpha\beta_1 \mid \alpha\beta_2$$

andando a sostituirla con due produzioni di questo tipo:

$$A \to \alpha A'$$
$$A' \to \beta_1 \mid \beta_2$$

dove $A' \notin \mathcal{A} \setminus T$, cioè è un non-terminale *fresh*.

Nel caso della prima grammatica non vi può essere determinismo nell'espansione data da A, perché sceglieremo l'espansione indipendentemente dalla lettera successiva a quella considerata correntemente; ad esempio, se potessi sapere che il prossimo simbolo sarà una b, allora non avrei dubbi su cosa scegliere.

7.6.2 Algoritmo di fattorizzazione a sinistra

Il tipo di trasformazione che possiamo fare ad una grammatica generica in modo da rimuovere questi prefissi comuni mantenendo il linguaggio è espresso dal seguente algoritmo:

Algorithm 12: GRAMMAR leftFactorization(GRAMMAR $\mathcal{G}$)

```
1 repeat
2     foreach A do
3         find the largest prefix α common to 2 or more productions for A
4         if α ≠ ε then
5             choose a fresh non-terminal A' and replace
6             A → αβ1 | ... | αβn | γ1 | ... | γk by
7             A → αA' | γ1 | ... | γk
8             A' → β1 | ... | βn
9 until no pair of productions for any A has common prefix
```

L'idea è che, data in input una grammatica che può essere fattorizzata a sinistra, per ogni non-terminale della grammatica

1. si trova il più lungo prefisso (α) comune a due o più produzioni aventi lo stesso driver
2. se tale prefisso α esiste, allora
   (a) viene scelto un non-terminale $A' : A' \notin \mathcal{A} \setminus T$
   (b) e si sostituiscono le produzioni di A nella forma

   $$A \to \alpha\beta_1 \mid ... \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_n$$

   con le seguenti produzioni:

   $$A \to \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_n$$
   $$A' \to \beta_1 \mid \ldots \mid \beta_n$$
3. si ripete da 1 fino a quando non è più possibile trovare produzioni con un prefisso comune (α)

L'ultimo punto ci fa intuire che non basta una semplice iterazione di tipo `foreach` per eliminare i prefissi comuni, ma è necessario esaminare tutte le produzioni, anche dopo che sono stati rimossi dei prefissi; questo perché, naturalmente, è possibile che si generino dei nuovi prefissi a seguito della rimozione dei primi, per cui è opportuno ripetere la procedura finché nessun nuovo prefisso viene trovato.

Considerazioni sull'efficacia Vediamo se tale algoritmo risolve il problema che avevamo precedentemente sollevato, ossia se questo impedisca la generazione di entries a definizione multipla nella tabella di parsing. Scopriamolo eseguendo la fattorizzazione sulla nostra fida $\mathcal{G} : S \to aSb \mid ab$, da cui otteniamo:

$$\begin{aligned} \mathcal{G}' : S &\to aS' \\ S' &\to Sb \mid b \end{aligned}$$

Ma quindi, questa $\mathcal{G}'$ che abbiamo appena ottenuto è una grammatica $LL(1)$? Se andiamo a calcolarne i *first/follow*, dovremmo trovare una qualcosa di simile a questo:

- $first(S) = \{a\}$ e $follow(S) = \{\$, b\}$;
- $first(S') = \{a, b\}$ e $follow(S') = \{\$, b\}$.

Per cui la tabella di parsing viene costruita come segue:

	a	b	$
S	$S \to aS'$		
S'	$S' \to Sb$	$S' \to b$	

Tabella 7.16: Top Down Parsing Table - Dopo fattorizzazione sinistra

Essendo che non abbiamo entries multiply-defined non possiamo fare altro che affermare che la grammatica così ottenuta è $LL(1)$; come sempre però un esempio non è sufficiente a dimostrare che questo valga in tutti i casi.

Dangling Else

Passiamo ora ad esaminare un caso di fattorizzazione sinistra molto famoso nei linguaggi di programmazione e che abbiamo già introdotto, ossia la grammatica ambigua degli `if` e degli `else`:

$$S \to \texttt{if}\ b\ \texttt{then}\ S \mid \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ S \mid c$$

Questa grammatica, una volta fatta passare nell'algoritmo di fattorizzazione sinistra, diventa:

$$\begin{aligned} S &\to \texttt{if}\ b\ \texttt{then}\ SS' \mid c \\ S' &\to \texttt{else}\ S \mid \varepsilon \end{aligned}$$

Per il risultato ottenuto in precedenza tale grammatica dovrebbe ora essere $LL(1)$, giusto? Calcoliamone i *first/follow*:

- $first(S) = \{\texttt{if}\ b\ \texttt{then}, c\}$ e $follow(S) = \{\$, \texttt{else}\}$
- $first(S') = \{\texttt{else}, \varepsilon\}$ e $follow(S') = \{\$, \texttt{else}\}$

In questo caso la grammatica risultante non è $LL(1)$, perché in $M[S', \texttt{else}]$ ci sono due produzioni: si può infatti osservare che

$$M[S', \texttt{else}] = S' \to \texttt{else}\ S, \text{ in quanto } \texttt{else} \in \text{first}(\texttt{else}\ S')$$
$$M[S', \texttt{else}] = S' \to \varepsilon, \text{ in quanto } \texttt{else} \in \text{follow}(S')$$

Abbiamo anche un altro modo per constatare che la nostra grammatica rimane ambigua, ossia andando a vedere il seguente albero di derivazione:

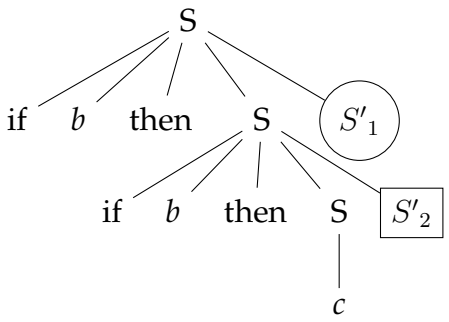


Figura 7.7: leftFactorizationAmbiguity

Si può infatti notare che

- in un caso è possibile scegliere di sostituire S'_1 con ε e S'_2 con $\texttt{else}\ c$
- nell'altro invece è possibile sostituire S'_1 con $\texttt{else}\ c$ e S'_2 con ε

In entrambi i casi si arrivare di fatto alla stessa stringa da due alberi di derivazione differenti (ottenuti impiegando due derivazioni leftmost).

Questo problema viene definito *dangling else* e corrisponde a non sapere identificare in maniera certa a quale `then` appartenga un determinato `else`. Per poter risolvere tale problematica vi sono due differenti soluzioni:

- proibire il costrutto `if-then` e sostituirlo con un `if-then-else`, tecnica che viene impiegata nel caso dei linguaggi funzionali: l'idea è che bisogna avere un ramo in cui il booleano b sia valutato a true e un altro nel caso in cui sia valutato a false; nel caso in cui l'`else` dovesse rivelarsi completamente inutile, allora si crea un ramo fittizio;

- imporre l'**innermost binding**, dove si fa corrispondere l'`else` al `then` più vicino e non ancora matchato: ovviamente tale soluzione ha senso solo nel caso in cui non si vogliono utilizzare parentesizzazioni.

L'innermost binding può anche essere implementato attraverso la definizioni di policy per definire particolari direttive al parser (approfondiremo il discorso quando parleremo di Byson, niente paura); oppure specializzando la grammatica, permettendo solo a coppie di `then-else` che matchano tra le occorrenze di `then` e `else`: in particolare, il secondo risultato può essere ottenuto utilizzando la grammatica seguente:

$$
\begin{aligned}
S &\to M \mid U \\
M &\to \texttt{if}\ b\ \texttt{then}\ M\ \texttt{else}\ M \mid c \\
U &\to \texttt{if}\ b\ \texttt{then}\ S \mid \texttt{if}\ b\ \texttt{then}\ M\ \texttt{else}\ U
\end{aligned}
$$

7.7 Riepilogo sulle grammatiche LL(1)

A questo punto dovrebbe essere chiaro che, se una grammatica è

- ricorsiva a sinistra $\vee$
- fattorizzabile a sinistra $\vee$
- ambigua,

allora non è possibile che sia una grammatica $LL(1)$. Abbiamo quindi un lemma, che fa più o meno così:

Lemma 7.7.1. *$\mathcal{G}$ è una grammatica $LL(1)$ se e solo se, nel caso in cui $\mathcal{G}$ avesse delle produzioni del tipo $A \to \alpha \mid \beta$, allora:*

- *first(α) $\cap$ first(β) = $\emptyset$;*
- *se $\varepsilon \in$ first(α), allora first(β) $\cap$ follow(A) = $\emptyset$ e, viceversa, se $\varepsilon \in$ first(β), allora first(α) $\cap$ follow(A) = $\emptyset$*

Questo termina la discussione sulle grammatiche $LL(1)$. Va però detto che tali proprietà sono estensibili alle grammatiche $LL(K)$; ricordiamo infatti che l'acronimo LL si riferisce al fatto che la stringa viene letta da sinistra verso destra e che viene applicata un tipo di derivazione leftmost, mentre il numero 1 messo tra parentesi ci dice che il nostro algoritmo di parsing analizzerà un elemento di input alla volta. L'algoritmo si può dunque estendere se scegliamo di controllare

un numero arbitrario K di elementi alla volta: nel complesso, la strategia è analoga; tuttavia, la tabella di parsing prevedrà delle entries in più.

Ad esempio, nel caso di una grammatica $LL(2)$ vanno segnalate sulle colonne coppie di simboli terminali; se prendiamo come riferimento la grammatica $S \to aSb \mid ab$, avremo sulle colonne le coppie di terminali aa, ab, bb, ba, $b\$$, $a\$$, $\$\$$. In questo caso, avendo la possibilità di vedere due simboli alla volta riuscirei a fare un parsing predittivo deterministico? Sì, perché so per certo che se leggessi ab dovrei utilizzare la produzione $S \to ab$, mentre in tutti gli altri casi $S \to aSb$.

7.8 Esercizi riassuntivi sulle grammatiche LL(1)

Esercizio 1

Sia data la grammatica $S \to aSb \mid \varepsilon$ che denota il linguaggio $\{a^n b^n \mid n \geq 0\}$; questa grammatica è $LL(1)$?

Possiamo facilmente calcolare che first(S) = $\{a, \varepsilon\}$, così come follow(S) = $\{\$, b\}$.

Per il lemma precedente visto che first(aSb) $\cap$ first(ε) = $\emptyset$ e che, nonostante $\varepsilon \in$ first(ε), abbiamo che first(aSb) $\cap$ follow(S) = $\emptyset$; ciò può essere osservato anche dalla tabella di parsing predittivo:

	a	b	$
S	$S \to aSb$	$S \to \varepsilon$	$S \to \varepsilon$

Tabella 7.17: Es 1 - Training LL(1)

Sorprendentemente dunque la grammatica è $LL(1)$.

Esercizio 2

Consideriamo adesso la seguente grammatica:

$$S \to AbB \mid B$$
$$A \to cB \mid a$$
$$B \to A$$

La grammatica sopra citata è $LL(1)$? Il sospetto è di no, perché a prima impressione i first(A) sono uguali ai first(B) e, essendo che nelle produzioni di S compaiono in prima posizione entrambi i non-terminali discussi, è possibile che

si vadano a collocare nella stessa cella. Procediamo per gradi calcolando i first e, se necessario, i follow:

	first
S	$\{a, c\}$
A	$\{a, c\}$
B	$\{a, c\}$

Tabella 7.18: Es 2: Calcolo First - Training LL(1)

Questa grammatica non è dunque $LL(1)$, perché nella tabella di parsing avrò le produzioni $S \to AbB$ e $S \to B$ sia nella cella $M[S, a]$ che in $M[S, c]$.

	a	b	c	$
S	$S \to AbB$ $S \to B$		$S \to AbB$ $S \to B$	
A	$A \to a$		$A \to cB$	
B	$B \to A$		$B \to A$	

Tabella 7.19: Es 2: Training LL(1)

Esercizio 3

Esaminiamo infine questa grammatica:

$$S \to Aa$$
$$A \to bB \mid c$$
$$B \to aA \mid \varepsilon$$

	first	follow
S	$\{b, c\}$	$
A	$\{b, c\}$	a
B	$\{a, \varepsilon\}$	a

Tabella 7.20: Es 3: Calcolo First & follow - Training LL(1)

Si può notare che mi ritroverò con produzioni in $M[B, a]$, e ciò è osservabile anche per via del lemma enunciato precedentemente: infatti, si ha che $\varepsilon \in$ first(α) e first(aA) $\cap$ follow(B) = $\{a\} \neq \emptyset$.

la tabella risultante è la seguente, e possiamo dedurne che la grammatica non è $LL(1)$.

	a	b	c	$
S		$S \to Aa$	$S \to Aa$	
A		$A \to bB$	$A \to c$	
B	$B \to aA$ $B \to \varepsilon$			

Tabella 7.21: Es 3: Training LL(1)

Capitolo 8

Analisi sintattica: Introduzione al bottom-up parsing e parsing SLR(1)

Iniziamo la trattazione del parsing di tipo *bottom-up*: come suggerisce il nome stesso, consiste nel ricostruire le derivazioni di una parola in ordine inverso, partendo dall'ultima produzione usata per derivare la parola considerata, fino ad arrivare allo start symbol; a livello visivo, possiamo pensare che la nostra intenzione sia quella di costruire un albero di derivazione partendo dalle foglie e arrivando alla radice.

Prima di cominciare, un'avvertenza: questo argomento è molto complesso e la teoria sottesa è piuttosto criptica e pesante, per cui suggeriamo al lettore di dedicare molta attenzione agli esempi ed esercizi che presenteremo, perché a nostro avviso sono spesso fondamentali alla comprensione dello stesso impianto teorico su cui poggiano.

8.1 Introduzione

8.1.1 Schema del processo

Durante questo e durante i prossimi capitoli andremo a conoscere diverse varianti del bottom-up parsing, che però condividono tutte lo stesso schema di operazioni da compiere. È uno schema diviso in quattro fasi, che andremo ad analizzare una ad una per ciascuna delle diverse varianti che incontreremo, per cui è bene conoscerlo subito e cercare di tenerlo a mente.

1. All'inizio, altro non abbiamo se non una grammatica $\mathcal{G}$ e una parola w;

2. la prima operazione sarà la costruzione di un automa caratteristico per la nostra $\mathcal{G}$, che è semplicemente un automa a stati finiti simile ad altri che abbiamo già visto;

3. una volta che avremo l'automa, possiamo usarlo per calcolarne una tabella di parsing bottom-up;

4. infine, utilizzando la tabella, lanceremo un algoritmo chiamato `shift/reduce` che, con l'ausilio della tabella appena calcolata, effettuerà il parsing vero e proprio della nostra parola w rispetto alla grammatica $\mathcal{G}$.

Consigliamo al lettore di tornare spesso a rivedere questo elenco numerato, perché andremo a trattare ciascun passaggio in maniera piuttosto estesa e si rischia di perdere lo sguardo d'insieme del processo.

8.1.2 Tipologie di parsing

Le sigle Riprendiamo il discorso di quali sono le varianti di parsing bottom-up: come per il nostro caro top-down, esistono diverse modalità per svolgere questo tipo di parsing, nella nostra trattazione parleremo soprattutto di SLR(1) (detto anche SLR per gli amici), LR(1) e LALR(1).

Classi di grammatiche Non tutte queste tecniche sono applicabili su tutte le grammatiche; infatti, la formulazione di ogni grammatica determina quale tecnica di parsing potrà essere applicata, tanto che solitamente si dice che è la grammatica stessa ad appartenere ad una determinata classe (quindi il lettore non si stupisca se fra poco inizieremo a parlare di grammatiche LR(1), SLR(1) et similia). In altre parole, noi diciamo che una grammatica $\mathcal{G}$ appartiene a una certa classe di grammatiche, ad esempio LR(1), se siamo capaci di costruire una tabella di parsing di tipo LR(1) per $\mathcal{G}$ (o di qualsiasi altro tipo stiamo considerando).

Differenze tra le classi Dipendentemente dalla classe di grammatica considerata, avremo automi caratteristici che rappresentano le informazioni in maniera più o meno dettagliata. Maggiore è il livello di dettaglio dell'informazione, più diventa grande e complesso l'automa caratteristico, ma anche più potente diventa il nostro parsing, inteso come numero di diverse grammatiche che può analizzare. Tra quelli presentati, il meccanismo più potente è LR(1), complementare del LL(1) che abbiamo visto nel parsing top-down; per questo motivo noi cercheremo sempre di costruire una tabella di parsing deterministico che rientri nei vincoli di LR(1). Se questo non sarà possibile andremo a scalare in complessità con LALR(1) e SRL(1).

Ampliamento di $\mathcal{P}$ Indipendentemente da quella che è la classe che consideriamo, quando siamo nell'ambito del parsing bottom-up andremo sempre ad ampliare l'insieme $\mathcal{P}$ delle produzioni aggiungendo la produzione $S' \to S$, dove S' è un non-terminale *fresh* per la nostra grammatica $\mathcal{G}$.

8.2 Costruzione dell'automa

Tuffiamoci subito nel primo passaggio, ossia la costruzione di un automa caratteristico a partire dalla grammatica data.

8.2.1 Gli stati

Andiamo per prima cosa a indagare quale forma avranno gli stati del nostro automa caratteristico. Gli stati sono degli insiemi di *items*, dove gli items sono oggetti che avranno forma diversa, dipendentemente dalla tecnica di parsing utilizzata:

LR(0)-items $A \to \alpha \cdot \beta$

LR(1)-items $[A \to \alpha \cdot \beta, L]$, dove $L \subseteq T \cup \{\$\}$[1]

Dalla definizione possiamo intuire cosa intendevamo quando prima abbiamo detto che gli LR(1)-items sono più ricchi dei loro corrispondenti LR(0), e ci permettono quindi di riconoscere grammatiche in modo più efficace: quell'insieme L, come vedremo in seguito, contiene delle informazioni rispetto a quello che ci aspettiamo di leggere in futuro; nessuna informazine simile è presente negli LR(0)-items.

Esempio Andiamo a fare subito un esempio per aiutarci a capire di cosa stiamo parlando. Consideriamo la nostra fida grammatica:

$$\mathcal{G} : S \to aSb \mid ab \tag{8.1}$$

Come avevamo anticipato in 8.1.2, come prima cosa aggiungiamo alla grammatica una nuova produzione $S' \to \cdot S$, e partiamo proprio da quest'ultima. Consideriamo il suo LR(0)-item $S' \to \cdot S$: il significato intuitivo è che, se siamo all'inizio della procedura di parsing, siamo in una posizione in cui vogliamo conoscere quali sono le parole derivabili a partire da S (o, nel caso più generale, di qualsiasi cosa segua il marker $\cdot$), per cui è logico pensare che il nostro item $S' \to \cdot S$ debba stare nello stato iniziale dell'automa, che chiamiamo P_0.

[1] Ricordiamo che T è l'insieme dei terminali della grammatica considerata.

Ma non sarà l'unico item a risiedere in P_0. Guardiamo bene 8.1: analizzare la parola vuol dire aspettarsi qualcosa che derivi da aSb oppure ab, per cui gli LR(0)-items di S saranno:

- $S \to \cdot aSb$;
- $S \to \cdot ab$.

Questi verranno inseriti nel nostro P_0, assieme all'item $S' \to \cdot S$ che avevamo identificato prima.

8.2.2 Chiusura di un insieme di LR(0)-items

Nel nostro esempio precedente abbiamo cercato di capire intuitivamente quali items è logico inserire negli stati, ma è chiaro che serve ben poca complessità per mettere fuori gioco il nostro intuito. Abbiamo bisogno di un procedimento formale per determinare quali items andranno inseriti nei vari stati, e in questo ci viene in aiuto il concetto di chiusura di un insieme di LR(0)-items:

Definizione 8.2.1. *Sia P un insieme di LR(0)-items; allora, $closure_0(P)$ è il più piccolo insieme che soddisfa la seguente equazione:*

$$closure_0(P) = P \cup \{B \to \cdot\gamma \mid A \to \alpha \cdot B\beta \in closure_0(P) \text{ e } B \to \gamma \in P'\} \quad (8.2)$$

La chiusura consiste sostanzialmente nell'aggiungere, per tutti quegli items che hanno un punto davanti ad un non-terminale, tutte le derivazioni possibili di quel non-terminale; questo va applicato ricorsivamente fino a che non sono presenti tutte le chiusure delle produzioni con un punto davanti ad un non-terminale. Più facile a farsi che a dirsi, credetemi, come del resto qualsiasi altro procedimento che vedremo in questo capitolo.

Esempio di calcolo della chiusura

Prendiamo ora ad esempio la seguente grammatica:

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

Come primo passo, consideriamo l'item $E' \to \cdot E$ e calcoliamone la chiusura $closure_0(\{E' \to \cdot E\})$. Ecco il procedimento passo per passo:

1. inizializziamo $closure_0(\{E' \to \cdot E\})$ = $\{E' \to \cdot E\}$;

2. andiamo a vedere se questo insieme contiene qualche marker davanti ad un non-terminale: effettivamente, c'è un marker prima di E;
3. aggiungiamo quindi le due produzioni di E alla chiusura $\{E \to \cdot E + T\}$ e $\{E \to \cdot T\}$;
4. una volta aggiunte queste produzioni, vediamo ricorsivamente se si presentano altre situazioni con $\cdot$ prima di un non-terminale;
5. nel primo caso troviamo ancora $\cdot E$, però abbiamo già analizzato la produzione E, per cui possiamo passare oltre;
6. nel secondo caso invece abbiamo $\cdot T$ e non abbiamo ancora analizzato tutte le produzioni di T, quindi andiamo ad aggiungere all'insieme le produzioni di T;
7. aggiungamo $\{T \to \cdot T * F\}$ e $\{T \to \cdot F\}$;
8. ci troviamo di nuovo in un caso in cui abbiamo due nuove derivazioni con $\cdot$ davanti a un non-terminale, ma $\cdot T$ è già analizzato, per cui analizziamo solo $\cdot F$;
9. aggiungiamo le produzioni di F: $\{F \to \cdot(E)\}$ e $\{F \to \cdot id\}$;
10. siamo arrivati alla conclusione, e qui sotto è riportato l'insieme chiusura che abbiamo trovato:

$$
\begin{aligned}
E' &\to \cdot E \\
E &\to \cdot E + T \\
E &\to \cdot T \\
T &\to \cdot T * F \\
T &\to \cdot F \\
F &\to \cdot(E) \\
F &\to \cdot id
\end{aligned}
$$

Ora che ne abbiamo visto un'applicazione, formalizziamo questa procedura-con dello pseudocodice: solo per voi, l'algoritmo per il calcolo della chiusura

Algorithm 13: SET chiusura$_0$(SET P)

```
1 foreach item ∈ P do
2     etichetta item come unmarked
3 while ∃item ∈ P : item è unmarked do
4     mark(item)
5     if item ha forma A → α · Bβ then
6         foreach B → γ ∈ P' do
7             if B → ·γ ∉ P then
8                 aggiungi B → ·γ a P come unmarked item
9 return P
```

8.2.3 Costruzione di un automa caratteristico LR(0)

Abbiamo visto cosa sono gli stati che compongono gli automi caratteristici; a questo punto siamo pronti per costruire il nostro primo automa caratteristico, in questo caso per il parsing. La tecnica di costruzione è incrementale: andiamo a popolare un set di stati definendo mano a mano la funzione di transizione, fino a saturazione; il lettore si accorgerà che tale tecnica è poi utilizzata per costruire anche altri automi LR.

Procedura

Inizio In primis, definiamo il kernel dello stato iniziale come $P_0 = \{S' \to \cdot S\}$, dove S' è un carattere inserito da noi, mentre S è lo start symbol della nostra grammatica.

Svolgimento Fino a che non esauriamo gli stati ancora da visitare andiamo a prenderli uno alla volta e li analizziamo nel segiente modo:

1. calcoliamo la chiusura del kernel dello stato, questo insieme rappresenta tutte le produzioni che si possono compiere da un certo stato per transitare verso altri stati;

2. una volta calcolata la chiusura del kernel, le produzioni (gli item) che abbiamo già collezionato avranno forma $A \to \alpha \cdot x\beta$, il che significa che nello stato in cui mi trovo, diciamo P, ho già visto α e posso fare una transizione tramite x;

3. esiste quindi una transizione da P a uno stato P' attraverso l'item $A \to \alpha x \cdot \beta$; se x è un terminale, tale transizione rappresenta un'operazione di shift (come abbiamo visto nell'esercizio della sezione scorsa). Questo significa che avrò una transizione etichettata con x che mi porta da P a P';

4. a questo punto genero il nuovo stato P' che contiene come kernel l'item $A \to \alpha x \cdot \beta$, ricordando che poi andrò a includere tra gli item di P' anche gli item che appartengono a $closure_0(\{A \to \alpha x \cdot \beta\})$, poiché se β è un non-terminale allora mi aspetto di poter trovare in P' anche tutto ciò che deriva da β.

C'è però una nota da aggiungere a questo procedimento: può accadere che quando generiamo un nuovo stato P' per transizione da uno stato P ci rendiamo conto che il kernel di questo stato corrisponde al kernel di un altro stato che abbiamo già vistisato, diciamo Q; in questo caso invece che creare un nuovo stato P', quello che facciamo è collegare (tramite una x-transizione) P a Q.

Esempio di costruzione di automa LR(0)

Per consolidare la procedura sopra illustrata, andiamo subito a mettere le mani su un esempio di costruzione di un automa caratteristico per il parsing LR(0), in particolare per la seguente grammatica:

$$S \to aABe$$
$$A \to Abc \mid b$$
$$B \to d$$

Inizio Partiamo creando lo stato iniziale, lo stato 0:

$$S' \to \cdot S$$

Ma c'è di più: nello stato iniziale va inserita anche la $closure_0(\{S' \to \cdot S\})$, quindi aggiungo le produzioni di S: $S \to \cdot aABe$. Dato che non sono presenti altre produzioni con marker prima di caratteri non-terminali, la chiusura dello stato 0 è completa.

Svolgimento A questo punto ci troviamo nello stato 0 ed abbiamo due produzioni, una con il marker prima di S ed una con il marker prima di a, per cui dobbiamo aggiungere le seguenti transizioni:

1. $\tau(0, S) = 1$[2], ovvero una transizione che sta per "sono nello stato 0 e leggo S nell'input buffer";
2. $\tau(0, a) = 2$, ovvero una transizione che sta per "sono nello stato 0 e leggo a nell'input buffer".

Questi stati però potrebbero essere già presenti! Non è questo il caso dato che sono i primi stati che troviamo, ma in seguito dovremmo ricordarci di tale controllo.

- Partiamo con l'analizzare il nuovo stato $\tau(0, S) = 1$.
  Dobbiamo calcolare il *kernel* dello stato, che si ottiene spostando il marker oltre al carattere che ci ha portati qui:

  $$S' \to S\cdot$$

  Questo è quello che viene definito kernel dello stato; non presenta ulteriori transizioni possibili dato che il marker è arrivato in fondo, quindi passiamo ad analizzare un altro stato.

- Andiamo ad analizzare lo stato che raggiungiamo da 0 dopo aver letto una a $\tau(0, a) = 2$.
  Il kernel questa volta è:

  $$S \to a \cdot ABe$$

  Dato che il kernel presenta almeno una produzione con un non-terminale alla destra del marker, dobbiamo aggiungere la chiusura del kernel a questo stato, ovvero:

  $$A \to \cdot Abc$$
  $$A \to \cdot b$$

  Quindi gli item dello stato 2 sono:

  $$S \to a \cdot ABe$$
  $$A \to \cdot Abc$$
  $$A \to \cdot b$$

  Dallo stato 2 avremo quindi due possibili transizioni, una tramite A ed una tramite b.

[2]Come il lettore attento avrà già osservato, gli stati possono venire indicati con la notazione: $\tau(\text{stato di provenienza}, \text{transizione di provenienza})$.

- Partiamo ad analizzare $\tau(2, A) = 3$.
  Il kernel di questo stato è composto da due produzioni, dato che da 2 si può arrivare in 3 tramite due distinte produzioni:

  $$S \to aA \cdot Be$$
  $$A \to A \cdot bc$$

  Per verificare se questo stato è già stato raggiunto, vado a verificare che non siano presenti stati con gli stessi item; in questo caso non ce ne sono, per cui lo stato 3 non è ancora stato effettivamente aggiunto e quindi lo tengo. Calcoliamo la chiusura dello stato, che ci porta ad aggiungere la seguente produzione:

  $$B \to \cdot d$$

  Una volta calcolata la chiusura, mi segno i nuovi stati da visitare.

  - $\tau(3, B) = 5$
  - $\tau(3, b) = 6$
  - $\tau(3, d) = 7$

- Analizziamo ora lo stato $\tau(2, b) = 4$.
  Questo stato ha come kernel:

  $$A \to b\cdot$$

  e non presenta ulteriori possibili sviluppi, quindi passiamo oltre.

- Analizziamo lo stato $\tau(3, B) = 5$.
  Il kernel in questo caso è:

  $$S \to aAB \cdot e$$

  questo kernel è già chiuso (la sua chiusura, infatti, è un insieme vuoto) e ci offre come unica transizione possibile $\tau(5, e) = 8$.

- Analizziamo lo stato $\tau(3, B) = 6$.
  Il kernel in questo caso è:

  $$A \to Ab \cdot c$$

  Anche questo kernel è già chiuso; ci offre la transizione allo stato $\tau(6, c) = 9$.

- Analizziamo lo stato $\tau(3, d) = 7$.
  Questo stato ha come kernel:

  $$B \to d\cdot$$

  Tale kernel è chiuso e non presnta transizioni uscenti.

- Non ci rimane che analizzare gli stati 8 e 9 che presentano rispettivamente i seguenti kernel:

$$S \to aABe\cdot$$
$$A \to Abc\cdot$$

entrambi i kernel non presentano ulteriori transizioni.

Conclusione In conclusione, l'automa caratteristico che otteniamo da questo procedimento può essere visualizzato in Fig.8.1

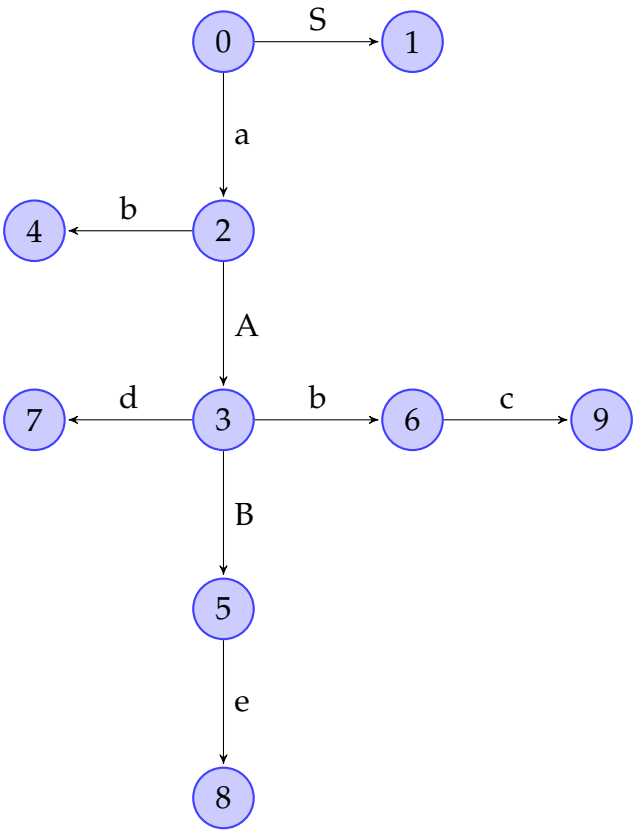


Figura 8.1: Automa caratteristico LR(0) per la grammatica 8.2.3

Una volta terminata questa arzigogolata esercitazione possiamo dare un'occhiata all'algoritmo per la costruzione di un automa LR(0) in Alg.14.

Algorithm 14: CharAutomataConstruction(GRAMMAR $\mathcal{G}$)

```
1  inizializza la collezione 𝒬 in modo che contenga P_0 =
     chiusura_0({S' → S})
2  etichetta P_0 come unmarked
3  while ∃P ∈ 𝒬 : P è unmarked do
4      mark(P)
5      foreach Y a destra del marker in qualche item di P do
6          calcola il kernel-set tmp per lo stato bersaglio della Y-transizione
             che parte da P
7          if 𝒬 contiene uno stato Q di kernel tmp then
8              sia Q lo stato bersaglio della Y-transizione che parte da P
9          else
10             aggiungi chiusura_0(tmp) a 𝒬 come stato unmarked
11             sia chiusura_0(tmp) lo stato bersaglio della Y-transizione che
                 parte da P
```

8.3 Costruzione di una tabella di parsing bottom-up

Introduzione Ci stiamo avvicinando, fra poco parleremo dell'algoritmo shift/reduce, vero e proprio direttore d'orchestra del parsing bottom-up. Questi compie delle mosse in base a cosa leggiamo su due pile ausiliarie ($stSt$, pila degli stati, e $symSt$, pila dei simboli), che sono valorizzate durante l'esecuzione a partire dalla parola considerata, e la decisione di quale mossa compiere è determinata dalla tabella di parsing della grammatica. Difatti, ora andremo a parlare di questa tabella.

Forma della tabella La tabella ha tante righe quanti gli stati dell'automa caratteristico, ed una colonna per ogni simbolo in $V \cup \{\$\}$. La tabella dipende dall'automa caratteristico: automi diversi portano a tabelle diverse che portano a tipi di parsing diversi.

Le riduzioni Le mosse di shift dipendono direttamente dalla funzione di transizione dell'automa, mentre le mosse di reduce sono più articolate: vanno effettuate solo quando raggiungiamo degli stati con etichette particolari, ed implicano che dobbiamo cancellare degli elementi dalla pila degli stati e dalla pila dei simboli, per poi inserire altri caratteri in quest'ultima pila. Inoltre, le mosse di reduce dipendono dal contenuto degli stati dell'automa: in un certo stato della tabella di parsing andiamo ad inserire una mossa di riduzione se la

produzione $A \to \beta$ è effettuata in uno stato contenente un reducing item per $A \to \beta$. A questo punto però ci chiediamo: cosa sono i reducing item?

- nel caso di LR(0)-items avranno forma $A \to \beta\cdot$, e indicano che siamo arrivati alla fine della derivazione di β, per cui il marker $\cdot$ si trova in fondo alla derivazione;
- nel caso LR(1)-items avranno forma $A \to \beta\cdot,\ \Delta$, e succede quando ho terminato l'analisi della produzione di β, ma con una condizione in più specificata da quel Δ.

Le riduzioni vengono applicate in base alla *lookahead function* $\mathcal{LA}$, la quale è definita per tutte le coppie $(P,\ A \to \beta)$ tali che P contiene un reducing item per $A \to \beta$.

Da menzionare il fatto che la scelta dell'automa e della lookahead function per la costruzione della parsing table sono le caratteristiche che distinguono le varie tecniche di bottom-up parsing: la procedura di compilazione della tabella di parsing e l'esecuzione dell'algoritmo shift/reduce sono uguali per ogni classe considerata.

8.3.1 Costruzione di una tabella di parsing generica

Vediamo ora nello specifico come costruire una tabella di parsing. Abbiamo una tabella con delle entries di forma $M[P, Y]$, dove P è uno stato e Y un simbolo del vocabolario V, e dobbiamo riempire ciascuna di queste secondo le seguenti regole:

- se Y è un terminale e $\tau(P, Y) = Q$ inserisci la mossa `shift` Q;
- se P contiene un reducing item per $A \to \beta$ e $Y \in \mathcal{LA}(P, A \to \beta)$, inserisci la mossa `reduce` $A \to \beta$ (notiamo che è in questa regola che emerge che le reduce dipendono dalla lookahead function);
- se P contiene l'accepting item e $Y = \$$ inserisci `accept`;
  - nel caso degli automi LR(0) l'accepting item è $\{S' \to S\cdot\}$;
  - nel caso degli automi LR(1) l'accepting item è $\{S' \to S\cdot,\ \Delta\}$;
- se Y è un terminale o \$ e nessuna delle condizioni precedenti è valida, inserisce `errore`;
- se Y è un non-terminale e $\tau(P, Y) = Q$ inserisci la mossa `goto` Q.

L'informazione che è contenuta nella tabella di parsing in posizione $M[P, Y]$ viene dunque utilizzata per stabilire che tipo di operazione effettuare durante il parsing, dipendentemente dallo stato P in cui ci troviamo e dal simbolo Y che leggiamo in cima all'input buffer, terminale o non-terminale che esso sia. Nonostante la grandezza della tabella dipenda dal tipo di algoritmo di parsing bottom-up che andiamo ad applicare e dalla funzione di lookahead ($\mathcal{LA}$), l'algoritmo che viene utilizzato per riempire la tabella rimane quello appena visto.

8.3.2 Conflitti

Visto che abbiamo già avuto modo di discutere di tabelle di parsing anche per il parsing di tipo top-down, risulta naturale metterle a confronto e chiedersi se, anche in questo caso, sia possibile ottenere dei **conflitti** (ovvero delle entries multiple defined). Questa situazione può verificarsi anche in questo caso in due differenti modalità: parleremo di

- **s/r conflict** (o shift/reduce conflict) nel caso in cui almeno un entry $M[P, Y]$ della tabella di parsing contenga sia un'operazione di `shift` Q (data dal fatto che esiste una Y-transizione che va dallo stato P, dove ci troviamo attualmente, allo stato Q), sia un'operazione di `reduce` $A \to \beta$ (poiché P contiene un reducing item per $A \to \beta$ e $Y \in \mathcal{LA}(P, A \to \beta)$);
- **r/r conflict** (o reduce/reduce conflict) nel caso in cui almeno un entry della tabella di parsing contenga due operazioni di `reduce` per produzioni distinte.

La prossima domanda sorge spontanea: cosa accade quando abbiamo un conflitto? La conclusione che possiamo trarre è che, se stiamo eseguendo un parsing di un certo tipo per una grammatica $\mathcal{G}$ e troviamo un conflitto mentre costruiamo la tabella, allora $\mathcal{G}$ *non* è una grammatica di quel particolare tipo; ad esempio, se troviamo un conflitto mentre costruiamo una tabella LR(1) per una grammatica $\mathcal{G}$, allora $\mathcal{G}$ non è LR(1). Questo significa che tale grammatica $\mathcal{G}$ non può essere parsata *in modo deterministico* utilizzando il parsing LR(1).

8.3.3 Costruzione di una tabella di parsing SLR(1)

Entriamo nel merito di come costruire una tabella di parsing SLR(1), ovvero parsing Simple Left (l'input viene letto da sinistra) Rightmost (utilizza derivazione rightmost); questo è un parsing poco raffinato e le tabelle contengono informazioni grossolane, e questo è dovuto a due motivi:

1. la loro costruzione ha come base di partenza un automa caratteristico con LR(0)-item;

2. la funzione di lookahead è pari a $\mathcal{LA}(P, A \to \beta) = follow(A)$ per ogni $A \to \beta\cdot \in P$.

Poiché repetita iuvant, ribadiamo che una grammatica $\mathcal{G}$ è SLR(1) se e solo se la parsing table ottenuta secondo quelle prescrizioni non ha conflitti.

Esempio di costruzione di tabella SLR(1)

Proviamo a costruire la tabella di parsing $SLR(1)$ per la seguente grammatica $\mathcal{G}$:

$$S \to aABe$$
$$A \to Abc \mid b$$
$$B \to d$$

Riconoscerete di certo questa grammatica, l'abbiamo già vista nella costruzione dell'automa caratteristico (si osservi Fig.8.2). Tuttavia, il nostro obiettivo qui è costruire la tabella di parsing, e per farlo utilizzeremo i risultati ottenuti in precedenza: sappiamo infatti che lo stato 1 (anche indicato dal colore verde) è quello che contiene l'**accepting item**[3], mentre invece gli stati contenenti i **reducing item** (segnalati come stati finali) sono invece 4, 7, 8, 9.

[3]Ovvero uno stato etichettato come $Accept$ e che, se raggiunto, implicherà una conclusione con successo dell'algoritmo.

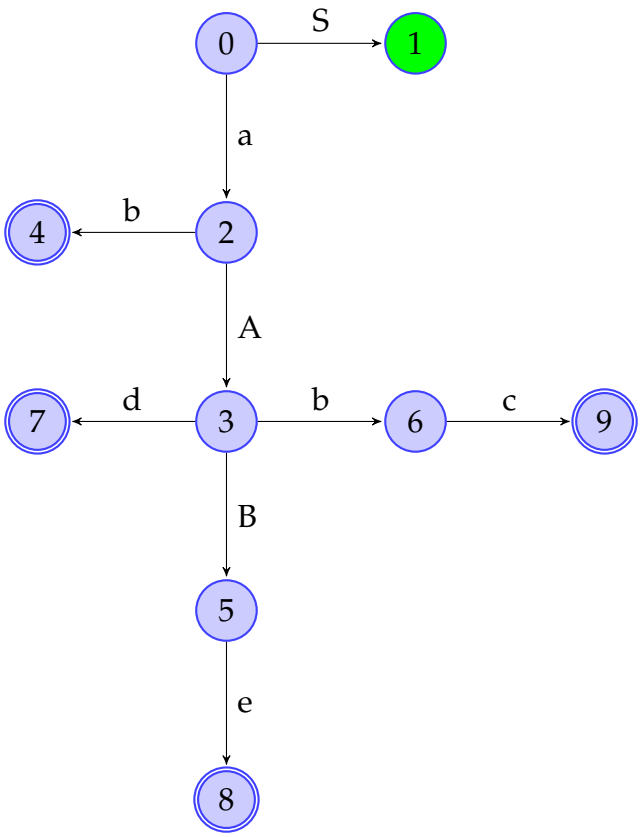


Figura 8.2: Automa caratteristico per la parsing table bottom-up

In particolare, possiamo dire che i reducing item sono:

1. $\{A \to b\cdot\}$ per lo stato 4;
2. $\{B \to d\cdot\}$ per lo stato 7;
3. $\{S \to aABe\cdot\}$ per lo stato 8;
4. $\{A \to Abc\cdot\}$ per lo stato 9;

Avendo a disposizione l'automa caratteristico e i dati sopra citati è dunque possibile costruire la tabella di parsing seguendo l'algoritmo per la costruzione di una tabella di parsing per il bottom-up parsing (ricordiamo all'utente distratto che siamo nell'ambito del parsing; parsing e chiudoing). Procediamo quindi in questo modo:

1. creiamo una tabella con tante righe quanti sono gli stati nel nostro automa caratteristico (in questo caso da 0 a 9) e con tante colonne quanti sono i

terminali (a cui va aggiunto il $) e i non-terminali presenti all'interno della nostra grammatica (in questo caso abbiamo di conseguenza *a b c d e $ A B S*);

2. facendo riferimento all'automa caratteristico, se Y è un terminale e c'è una transizione $\tau(P, Y) = Q$ (ovvero una transizione che va dallo stato P allo stato Q attraversando un arco con etichetta Y), allora inseriamo un'operazione di `shift` Q nella cella $M[P, Y]$; ad esempio, nel nostro caso inseriremo $M[0, a] = 2$;

3. a questo punto è necessario calcolare i follow (e di conseguenza i first, Tab.8.1), in quanto per poter inserire correttamente le operazioni di `reduce` abbiamo bisogno di avere queste informazioni per la funzione di lookahead ($\mathcal{LA}$);

	first	follow
S	a	\$
A	b	b, d
B	d	e

Tabella 8.1: SLR(1) parsing table - calcolo follow per lookahead

4. una volta calcolati i follow procediamo così: per tutti gli stati P che contengono un reducing item $A \to \beta$ (ad esempio, $M[4, b] = A \to b\cdot$), andiamo a inserire la mossa di reduce nella casella $M[P, Y]$, dove Y è calcolato con la lookahead function come $Y = \mathcal{LA}(P, A \to \beta)$, nel nostro caso $Y = follow(A)$;

5. inseriamo in posizione $M[1, \$]$ l'operazione di `Accept`, in quanto 1 contiene un accepting item;

6. a questo punto dovremmo valorizzare a `error` tutte quelle celle $M[P, Y]$ dove Y è o $ o un terminale che non ricade nei casi precedenti; tuttavia per una maggiore comprensione a livello visivo, nel nostro esempio queste celle verranno lasciate vuote;

7. infine, inseriamo un'operazione del tipo `goto` Q per tutti quelle celle $M[P, Y]$ dove Y è un non-terminale e per cui esiste una transizione $\tau(P, Y) = Q$ (ad esempio $M[0, S]$ = `goto` 1, come è anche osservabile dall'automa caratteristico).

In Tab.8.2 possiamo trovare il risultato finale dell'algoritmo. Ai fini di una corretta lettura della tabella, specifichiamo qui sotto alcune convenzioni che verranno impiegate per tenere il tutto conciso, come ad esempio:

- l'operazione di `shift` Q verrà indicata con sQ;
- l'operazione di `reduce` $A \to \beta$ con rK, dove K indica la K-esima riduzione; questa notazione è molto conveniente e abbassa il grado di entropia della tabella, ma in seguito forniremo anche una notazione estesa per queste riduzioni;
- *Acc* indica l'accepting item;
- l'operazione di `Goto` Q verrà indicata semplicemente da Q.

	a	b	c	d	e	$	S	A	B
0	s2						1		
1						Acc			
2		s4						3	
3		s6		s7					5
4		r3		r3					
5					s8				
6			s9						
7					r4				
8						r1			
9		r2		r2					

Tabella 8.2: Tabella di parsing SLR(1)

Notiamo che sulla porzione sinistra della tabella troviamo i vari terminali (compreso $) e, nelle varie entries, avremo mosse di shift, di reduce e delle celle valorizzate a `error`; a destra di questa, invece, troviamo i non terminali con le relative mosse di `goto`. Specifichiamo quindi il significato dei vari rK:

- $r1 = S \to aABe$
- $r2 = A \to Abc$
- $r3 = A \to b$

- $r4 = B \to d$

Dal momento che $\mathcal{G}$ dà origine a una tabella di parsing senza creare alcun conflitto, possiamo concludere che $\mathcal{G}$ è SLR(1).

Prima di passare alla prossima sezione è necessario precisare che *solamente* le celle vuote che hanno per colonna un terminale o \$ sarebbero valorizzate a `error`: infatti, le celle vuote che hanno per colonna un non-terminale rimangono semplicemente vuote, in quanto sono dei casi che non è possibile che si verifichino.

8.4 Algoritmo Shift/Reduce

Infine, eccoci arrivati: possiamo finalmente presentare l'algoritmo di shift/reduce, che ricordiamo essere quell'algoritmo che usiamo per ottenere il parsing di una parola w data la tabella di parsing bottom-up di una certa grammatica $\mathcal{G}$. Ricordiamo inoltre che l'algoritmo di `s/r` viene visto indipendentemente dalla classe di parsing considerata.

- **Input**: una stringa w e la tabella di parsing M per $\mathcal{G} = (V, T, S, \mathcal{P})$;
- **Output**: una derivazione di w in ordine inverso se $w \in \mathcal{L}(\mathcal{G})$ (per trovare i passi di derivazione che ci portano a w dovremo leggere tutte le reduce che abbiamo applicato in ordine inverso); otterremo invece `error` se $w \notin \mathcal{L}(\mathcal{G})$;
- **Inizializzazione**: Per l'algoritmo è necessario inserire all'interno dell'input buffer la stringa $w\$$, e inoltre abbiamo bisogno di due strutture dati:
  - $stSt$ (state stack - pila degli stati), che viene inizializzata ponendo in cima lo stato P_0 (ovvero prima chiusura dell'insieme di LR(0)-item);
  - $symSt$ (symbol stack - pila dei simboli).

Algorithm 15: bottomUpParsing(WORD w, TABLE[][] M)

```
1  b = the first symbol in the input buffer
2  initialize stSt by adding state 0
3  while true do
4      Let S be the top of stSt
5      if M[S, b] = shift T then
6          Push b onto symSt
7          Push T onto stSt
8          b = the next symbol in the input buffer
9      else if M[S, b] = reduce A → β then
10         Pop |β| symbols off symSt
11         Push A onto symSt
12         Pop |β| state off stSt
13         Let tmp be the top of stSt
14         Push T onto stSt, where T is such that M[tmp, A] = goto T
15         Output A → β
16     else if M[S, b] = ACCEPT then
17         return
18     else
19         error()
```

Andiamo subito a mettere in pratica questo algoritmo con un esempio.

Esercizio shift/reduce 1

Consideriamo questa parola:

$$w = abbcde$$

Andiamo a verificare se appartiene al linguaggio generato dalla grammatica:

$$\begin{aligned} \mathcal{G} : S &\to aABe \\ A &\to Abc \\ A &\to b \\ B &\to d \end{aligned} \tag{8.3}$$

Ne abbiamo già calcolato la tabella di parsing bottom-up in Tab.8.2.

Svolgimento La configurazione iniziale è la seguente:

$$\begin{aligned} w &= abbcde\$ \\ stSt &= 0 \\ symSt &= \end{aligned}$$

Il primo carattere che leggiamo nel buffer è a, per cui cerchiamo nella tabella del parsing il contenuto della casella [0,a]; scopriamo che contiene una mossa di shift verso lo stato 2, quindi andiamo ad inserire il nuovo stato di arrivo in $stSt$ e il carattere letto in $symSt$:

$$\begin{aligned} w &= \underline{a}bbcde\$ \\ stSt &= 0\ 2 \\ symSt &= a \end{aligned}$$

Ci troviamo ora nello stato 2 e leggiamo il secondo carattere della parola che è b, quindi controlliamo cosa contiene la casella [2,b] della tabella di parsing. Abbiamo un'altra mossa di shift che ci fa raggiungere lo stato 4 e leggere un altro carattere, e arriviamo quindi ad avere questa situazione:

$$\begin{aligned} w &= \underline{ab}bcde\$ \\ stSt &= 0\ 2\ 4 \\ symSt &= a\ b \end{aligned}$$

Ora siamo in 4 e leggiamo b. La cella [4,b] ci suggerisce di compiere una mossa di reduce, nello specifico $R : A \to b$; questo significa che compiremo due operazioni:

- elimineremo tanti stati da $stSt$ quanti gli elementi nel body della produzione (ovvero 1, dato che $|b| = 1$);
- eliminiremo il body stesso da $symSt$ e, al suo posto, inseriremo il driver della riduzione, in questo caso A.

Le nostre strutture saranno quindi valorizzate così:

$$\begin{aligned} w &= \underline{ab}bcde\$ \\ stSt &= 0\ 2\ \cancel{4} \\ symSt &= a\ A \end{aligned}$$

Ora però dobbiamo sostituire lo stato 4 con lo stato che è presente nella tabella in posizione [2,A], ovvero lo stato 3; la situazione si ristabilisce quindi così:

$$\begin{aligned} w &= \underline{ab}bcde\$ \\ stSt &= 0\ 2\ 3 \\ symSt &= a\ A \end{aligned}$$

Dobbiamo ancora consumare la b (le mosse di reduce non consumano i caratteri), quindi ora ci troviamo nella casella [3, b],e questa ci suggerisce la mossa shift 6; vediamo cosa succede:

$$\begin{aligned} w &= \underline{abb}cde\$ \\ stSt &= 0\ 2\ 3\ 6 \\ symSt &= a\ A\ b \end{aligned}$$

A questo punto la casella da guardare diventa [6,c], che ci dice shift 9, ob-la-di, ob-la-da:

$$\begin{aligned} w &= \underline{abbc}de\$ \\ stSt &= 0\ 2\ 3\ 6\ 9 \\ symSt &= a\ A\ b\ c \end{aligned}$$

Plot-twist! proprio mentre ci siamo abituati alla tranquillità delle mosse di shift la casella [9,d] ci lascia sbigottiti con una mossa di reduce $R : A \to Abc$, costringendoci ad eliminare ben 3 stati da $stSt$ e 3 caratteri da $symSt$ e sostituendoli con una sprezzante A:

$$\begin{aligned} w &= \underline{abbc}de\$ \\ stSt &= 0\ 2\ \not{3}\not{6}\not{9} \\ symSt &= a\ \not{A}\not{b}\not{c}A \end{aligned}$$

A questo punto siamo costretti a guardare in [2,A] per capire in quale stato siamo stati sballottati dal `goto`, e scopriamo che siamo malaugaratamente tornati nello stato 3:

$$\begin{aligned} w &= \underline{abbc}de\$ \\ stSt &= 0\ 2\ 3 \\ symSt &= a\ A \end{aligned}$$

Bene, riprendiamo con la lettura: eravamo rimasti a d, per cui ora controlliamo [3,d], che ci suggerisce shift 7:

$$\begin{aligned} w &= \underline{abbcd}e\$ \\ stSt &= 0\ 2\ 3\ 7 \\ symSt &= a\ A\ d \end{aligned}$$

Stato 7, il prossimo carattere nel buffer è e e la casella [7,e] presenta questa riduzione $R : B \to d$; ormai la prassi ci è nota (e prestate attenzione al fatto che

in [3, B] troviamo `goto` 5):

$$\begin{aligned} w &= \underline{abbcd}e\$ \\ stSt &= 0\ 2\ 3\ \not{7}\ 5 \\ symSt &= a\ A\ \not{d}\ B \end{aligned}$$

Dobbiamo ancora leggere la e e ci troviamo ora nello stato 5, e [5,e] ci dice shift 8:

$$\begin{aligned} w &= \underline{abbcde}\$ \\ stSt &= 0\ 2\ 3\ 5\ 8 \\ symSt &= a\ A\ \ B\ e \end{aligned}$$

Con 8 come stato e \$ come carattere in lettura ci assicuriamo una bella reduce in forma di $R : S \to aABe$, che ci fa arrivare a questa situazione (ricordiamoci del `goto`):

$$\begin{aligned} w &= \underline{abbcde}\$ \\ stSt &= 0\ 1 \\ symSt &= \end{aligned}$$

Il prossimo passo è descritto dalla casella [1,\$], che ci dice Accept; questo significa che abbiamo terminato la verifica e possiamo dire, con somma pace interiore, che $w \in \mathcal{L}(\mathcal{G})$.

Non abbiamo solo trovato una soluzione al quesito iniziale, ma abbiamo anche ricavato i passi di derivazione che ci portano ad ottenere la parola w stessa; questi. infatti, consistono nelle riduzioni che abbiamo incontrato nel nostro cammino elencate in senso contrario:

$$\begin{aligned} S &\to aABe & (aABe) \\ B &\to d & (aAde) \\ A &\to Abc & (aAbcde) \\ A &\to b & (abbcde) \end{aligned}$$

Il parsing dell'oca

Ai più attenti non sarà di certo sfuggito che questa stregoneria del parsing è stata fortemente ispirata dal gioco dell'oca. Pensiamoci: è come se, nel nostro parsing

dell'oca, l'obiettivo del gioco fosse quello di raggiungere una casella nascosta "dietro al via"[4]. Abbiamo tre tipi di caselle:

- le caselle di shift ci fanno avanzare di una casella;
- le caselle di reduce ci fanno indietrggiare di qualche casella;
- le celle error ci fanno perdere il gioco.

Purtroppo il tabellone di gioco non è un semplice percorso rettilineo, ma è una cosa più brutta che assomiglia un po' a un grafo. In compenso, nella nostra versione il percorso della pedina è *deterministico*: non c'è un lancio dei dadi, c'è una sorta di mappa da seguire, che altro non è se non la parola w che vogliamo analizzare. E infatti l'obiettivo del nostro gioco è quello di prendere una mappa (ovvero una parola), seguirla e verificare se riusciamo ad arrivare alla casella finale con delle opportune reduce (perché ripetiamolo, la casella finale si trova "dietro all via!"). E quindi, cosa succede quando arriviamo alla casella finale? Succede che abbiamo vinto, perché vuol dire che la mappa che abbiamo seguito è valida, ovvero che la parola w considerata appartiene alla grammatica $\mathcal{G}$ che ha generato il tabellone di gioco (ovvero il linguaggio); inoltre, possiamo anche capire esattamente come w deriva da $\mathcal{G}$, seguendo il nostro percorso al contrario e annotandoci le mosse di reduce effettuate. Questo era tutto, siamo finalmente riusciti a dispiegare il nostro parsing una volta per tutte! ~sam

Ottimo, adesso che abbiamo messo le cose in chiaro possiamo passare a un secondo esercizio, tanto più complesso quanto più completo e divertente del primo.

Esercizio shift/reduce 2

La consegna è di costruire la parsing table SLR(1) per la seguente, ormai ben nota, grammatica:

$$E \to E + E \mid E * E \mid id \tag{8.4}$$

Per costruire una parsing table abbiamo prima di tutto bisogno di ricavare l'automa a stati finiti determinato da tale grammatica.

[4]Abbiamo fiducia nel fatto che i nostri venticinque lettori stiano al gioco e riescano a intuire il senso di quest'espressione un po' criptica; la prima volta che l'ho letta pure a me suonava molto simile a quella storiella che i terrapiattisti tanto amano sui presunti racconti dei presunti diari di Richard Byrd, in cui l'ammiraglio affermerebbe di essersi spinto centinaia di miglia "oltre il Polo Sud". "Dietro al via", "Oltre il Polo", figata, no? ~pips

Costruzione dell'automa

1. Inizializziamo lo stato 0; il suo kernel è

   $$E' \to \cdot E$$

   Di questo kernel devo calcolare la chiusura:

   $$E \to \cdot E + E$$
   $$E \to \cdot E * E$$
   $$E \to \cdot id$$

   Essendo questi gli item per lo stato 0, posso identificare due possibili transizioni (e quindi due possibili nunovi stati): $\tau(0, E) = 1$ e $\tau(0, id) = 2$.

2. Analizziamo ora lo stato 2; il suo kernel è

   $$E \to id\cdot$$

   Non serve calcolare la chiusura di questo item; inoltre, tale item non comporta transizioni uscenti dallo stato 2, per cui terminiamo qui l'analisi.

3. Passiamo al vaglio lo stato 1, il cui kernel è popolato da questi item:

   $$E' \to E\cdot$$
   $$E \to E \cdot +E$$
   $$E \to E \cdot *E$$

   Tutti gli elementi del kernel sono già chiusi, e ci offrono queste transizioni: $\tau(1, +) = 3$ e $\tau(1, *) = 4$. Fatto ciò, abbiamo concluso l'analisi dello stato 1.

4. Passiamo allo stato 3, il cui kernel risulta

   $$E \to E + \cdot E$$

   Tale kernel presenta però la possibilità di chiusura, per cui aggiungiamo la chiusura agli item dello stato 3, che risultano essere:

   $$E \to E + \cdot E$$
   $$E \to \cdot E + E$$
   $$E \to \cdot E * E$$
   $$E \to \cdot id$$

   Quindi, da qui posso transizionare tramite E e tramite id. La transizione per E mi porta in $\tau(3, E) = 5$, la transizione per id invece è particolare: mi porterebbe in uno stato che ha come kernel $E \to id\cdot$, ma quello è lo stesso kernel dello stato 2! Quindi $\tau(3, id) = 2$.

5. Analizziamo lo stato 4, che ha per kernel

$$E \to E * \cdot E$$

Si può effettuare la chiusura, quindi gli item dello stato 4 sono:

$$E \to E * \cdot E$$
$$E \to \cdot E + E$$
$$E \to \cdot E * E$$
$$E \to \cdot id$$

Esistono due possibili transizioni uscenti dallo stato 4: una per E, che genera lo stato $\tau(4, E) = 6$, e una per id, che come per il caso precedente non genera un nuovo stato ma va a finire nello stato 2.

6. Quando analizzo lo stato 5, trovo che il kernel è

$$E \to E + E\cdot$$
$$E \to E \cdot + E$$
$$E \to E \cdot * E$$

Questo non necessita di chiusura, e ci fornisce due possibili transizioni:

- quella per $+$, che però ci porta in uno stato con kernel $E \to E + \cdot E$, ovvero nello stato 3, che abbiamo già analizzato;
- quella per $*$, che però ci porta in uno stato con kernel $E \to E * \cdot E$, ovvero nello stato 4, che abbiamo già analizzato.

7. Allo stesso modo, per lo stato 6 il kernel è

$$E \to E * E\cdot$$
$$E \to E \cdot + E$$
$$E \to E \cdot * E$$

e nemmeno questo necessita di chiusura, e perdipiù, attraverso le due transizioni per $+$ e per $*$, ci porta esattamente negli stati 3 e 4.

Una volta terminata l'analisi degli stati e dei relativi item ci preoccupiamo di trovare quali stati contengono l'accepting item e quali stati contengono i reducing item:

- l'accepting item nel metodo SLR(1) è l'item $S' \to S\cdot$ (ovvero $E' \to E\cdot$), che nel nostro caso è contenuto nello stato 1;

- i reducing item sono quelli con il marker · al termine di una produzione, nel nostro caso sono $E \to id\cdot$ (stato 2), $E \to E + E\cdot$ (stato 5) e $E \to E * E\cdot$ (stato 6).

Per concludere la parte di costruzione dell'automa caratteristico per la nostra grammatica in Eq.8.4, inseriamo in Fig.8.3 il disegno di tale sgorbio (scherzo, è carino, sembra un razzetto).

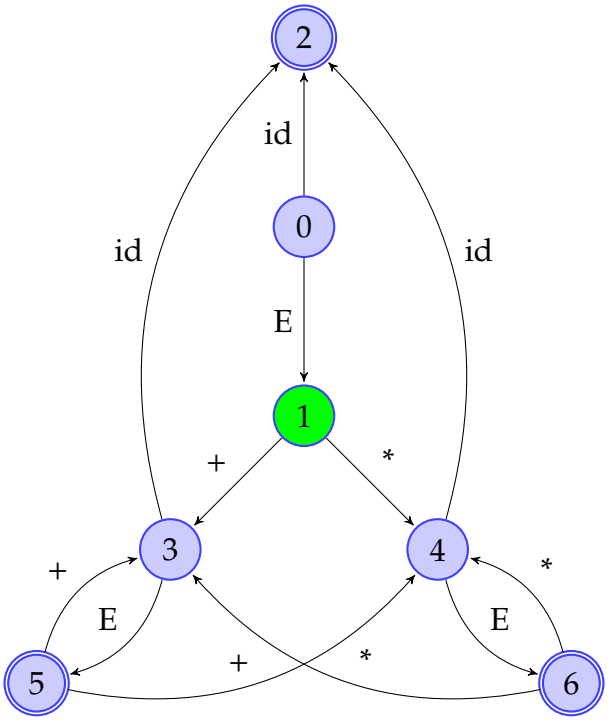


Figura 8.3: Automa caratteristico per la grammatica 8.4

8.4.1 Gestione dei conflitti

Quindi, partendo dalla grammatica Eq.8.4 siamo riusciti a ricavarne e l'automa caratteristico SLR(1) in Fig.8.3 e i reducing items, che ricordiamo essere:

$$2 : E \to id \qquad 5 : E \to E + E\cdot \qquad 6 : E \to E * E\cdot$$

Tutto questo è fantastico, ma purtroppo c'è un grosso elefante nella nostra stanza. L'intrinseca ambiguità di Eq.8.4 ha un effetto collaterale piuttosto sgradevole: la sua tabella di parsing SLR(1) avrà dei conflitti, come possiamo ben vedere in Tab.8.3.

	id	$+$	$*$	\$	E
0	s2				1
1		s3	s4	Acc	
2		r3	r3	r3	
3	s2				5
4	s2				6
5		s3, r1	s4, r1	r1	
6		s3, r2	s4, r2	r2	

Tabella 8.3: tabella di parsing SLR(1) per la grammatica 8.4

Quindi, se vogliamo perseguire la nostra ricerca di un parsing deterministico (e lo vogliamo), toccherà rimboccarsi le maniche e andare a capire come possiamo risolvere questi conflitti, non diversamente da quanto già avevamo fatto nel parsing top-down con le entries multiple defined.

I nostri conflitti sono localizzati solo in quattro situazioni, possiamo quando ci troviamo in uno tra gli stati 5 o 6 e, contemporaneamente, troviamo in lettura un simbolo tra $+$ o $*$.

	$+$	$*$
5	s3, r1	s4, r1
6	s3, r2	s4, r2

Tabella 8.4: Dettaglio di Tab.8.3

La situazione Proviamo a capire bene qual è il punto di tutta questa situazione. Se il nostro parser si trova nello stato 5, questo vuol dire che, in testa alla nostra pila dei simboli $symSt$, avremo necessariamente $E + E$; possiamo vederlo facilmente, è sufficiente buttare un occhio al nostro automa caratteristico (Fig.8.3) per renderci conto che lo stato 5 è raggiungibile solo dallo stato 3, il quale a sua volta può essere raggiunto o dalla sequenza di stati $(0, 1)$, $(4, 6)$ o $(3, 5)$, in tutti questi casi sulla testa di $symSt$ si troverà $E + E$. Il discorso è del tutto analogo se passiamo allo stato 6, dove avremo la testa di $symSt$ popolata di $E * E$, poiché 6 è raggiungibile tramite $\cdots \to 0 \to 1 \to 4 \to 6$, $\cdots \to 4 \to 6 \to 4 \to 6$ oppure $\cdots \to 3 \to 5 \to 4 \to 6$.

È chiaro che abbiamo dei conflitti quando siamo nella situazione in cui due simboli legati da un operatore sono stati già letti ($E + E$ o $E * E$) e, subito dopo, leggiamo un terzo simbolo. Questo è dovuto proprio al fatto che la grammatica,

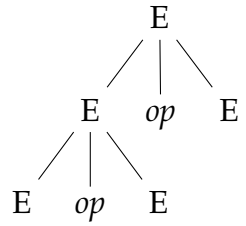
nella sua ambiguità, non riesce a esprimere opportunamente né associatività, né precedenza per i due operatori.

E insomma, qual è il segreto per risolvere questi dannati conflitti? Quello che faremo (almeno in questo caso) è, per ogni cella in cui abbiamo un conflitto, scegliere manualmente quale mossa mantenere[5]. Questa scelta pare essere in mano nostra, e nel nostro specifico esempio faremo sì che il parser finisca per conformarsi a quelle che sono le comuni regole di precedenza e associatività per $+$ e $*$.

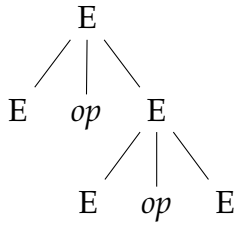
L'utilità dei parse trees Possiamo aiutarci se pensiamo ai parse tree della nostra grammatica: gli alberi, infatti, per loro stessa costruzione, veicolano informazioni sull'annidamento dei loro elementi in maniera molto efficace, e l'annidamento può tranquillamente avere una corrispondenza diretta con la parentesizzazione (quindi con le regole di precedenza tra espressioni). Di fatto assumendo, come facciamo noi, che i sottoalberi vengano risolti prima degli alberi padri, stabiliamo delle regole di precedenza del tutto simili a quelle che si possono ottenere con una parentesizzazione.

Nello specifico, noi siamo interessati a esprimere l'associatività sinistra degli operatori $+$ e $*$, nonché la precedenza di $*$ rispetto a $+$. Diciamolo in maniera ancora più pragmatica: se guardiamo gli esempi in Fig.8.4, vogliamo sempre avere l'albero Fig.8.4a al posto di Fig.8.4b quando stiamo valutando una stessa espressione ($E + E + E$, ad esempio), e vogliamo che la precedenza tra i due operatori sia tale da generare sempre parse tree come Fig.8.4c e Fig.8.4d: $E * E$ è più "in basso", ergo verrà eseguita prima, ergo avrà precedenza su $E + E$.

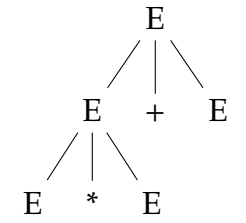
[5]Qualcuno potrebbe a ragione pensare che, in questo modo stiamo barando, e a dirla tutta avrebbe ragione; ma nel reame dei compilatori è meglio tenere la coscienza sporca e fare del parsing efficace rispetto a conservare l'integrità ma fare del parsing leeeeento.



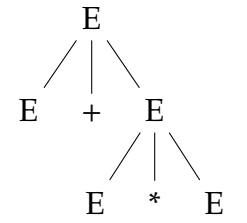
(a) Parse tree left-associativo



(b) Parse tree right-associativo



(c) Parse tree per $E * E + E$



(d) Parse tree per $E + E * E$

Figura 8.4

In questa interpretazione possiamo vedere le mosse di reduce come una parentesizzazione, perché stiamo assegnando un padre a un determinato sottoalbero del tipo $E + E$ o $E * E$[6].

Risoluzione dei conflitti A questo punto possiamo andare ad analizzare i quattro conflitti che abbiamo trovato e, per ciascuna cella, scegliere la mossa che rende il parsing conforme alle regole che abbiamo stabilito. Tenendo a mente Tab.8.4, rimbocchiamoci le maniche.

- $M[5, +]$: questa è la situazione in cui la testa di $symSt$ è popolata da $E + E$ e in lettura troviamo un altro $+$; gli operatori sono left-associativi, per cui vogliamo parentesizzare la testa del nostro $symSt$ e solo dopo leggere il nuovo simbolo; ergo, la mossa che scegliamo e inseriamo in $M[5, +]$ è $r1$.

- $M[6, *]$: vediamo subito questa cella perché presenta una situazione del tutto analoga a quella appena vista; la testa di $symSt$ è popolata di $E * E$ e questa volta l'operatore coinvolto è $*$, che al pari di $+$ è left-associativo, di conseguenza anche qui scegliamo la reduce $r2$ per dare precedenza alla parentesizzazione;

[6]Questa frase è importantissima e invitiamo il lettore a spenderci alcuni minuti, perché afferrarne appieno il significato implica di aver ottenuto una buona comprensione del parsing bottom-up.

- $M[5, *]$: torniamo adesso a parlare dello stato 5; la testa di $symSt$ è popolata come prima da $E + E$, ma questa volta il lettura troviamo un simbolo $*$. Questo cambia le carte in tavola, perché noi vogliamo che l'operatore $*$ abbia precedenza più alta di $+$; per ottenere questo, scegliamo la mossa di shift $s4$, il che significa che rimandiamo la parentesizzazione della testa di $symSt$ e proseguiamo nella lettura.
- $M[6, +]$: qui abbiamo trovato $E * E$ nella testa di $symSt$ e ci sta arrivando un $+$ in lettura, e ormai l'idea dovrebbe essere chiara: coerentemente con le regole di precedenza ($* \prec +$), vogliamo che la testa di $symSt$ sia parentesizzata prima di proseguire con la lettura di $+$, per cui qui teniamo la reduce $r2$.

	$+$	$*$
5	r1	s4
6	r2	r2

Figura 8.5: Soluzione dei conflitti di Tab.8.3

Fatto! Dopo tutto questo processo abbiamo ottenuto una tabella di parsing senza conflitti, ovvero senza entries multiple defined. Strategie di questo tipo sono abbastanza standard per le grammatiche degli operatori (non solo aritmetici, si pensi alle precedenze dell'algebra booleana); all'atto pratico, chi scrive le grammatiche (che ha studiato LFC e ha capito questi argomenti) si preoccuperà anche di predisporre delle direttive per il parser, sicché quest'ultimo abbia informazioni su quali scelte compiere in caso di conflitti.

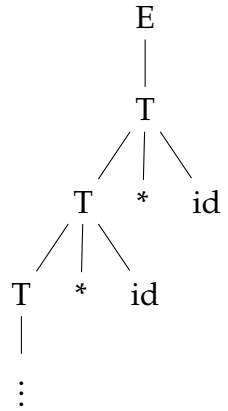
Un altro approccio: riscrivere la grammatica

Naturalmente, tutto questo è reso necessario dal fatto che la grammatica $\mathcal{G}$ in Eq.8.4 è ambigua; se avessimo semplicemente scritto un'altra grammatica $\mathcal{G}'$ non ambigua, tale che $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}')$, saremmo riusciti a ottenere una tabella di parsing senza conflitti da risolvere. Dovessimo percorrere questa strada, sia chiaro che è estremamente importante che la nuova grammatica esprima correttamente le regole di precedenza e associatività che vogliamo. Guardiamo ad esempio:

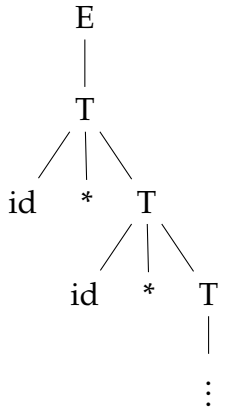
$$\begin{aligned} \mathcal{G}' : E &\to E + T \mid T \\ T &\to T * id \mid id \end{aligned} \tag{8.5}$$

L'ordine con cui abbiamo inserito le produzioni non è causale: abbiamo posizionato $*$ in un livello di produzione inferiore a $+$ perché, dal momento che la derivazione è rightmost e andiamo ad attraversare l'albero dal basso verso la radice (bottom-up, no?), ci assicuriamo in questo modo di parentesizzare prima tutte le occorrenze di $*$; non avremmo avuto questa garanzia se avessimo inserito produzioni come, ad esempio, $E \to T + E$.

Inoltre, se ad esempio avessimo scritto $T \to id * T$, avremmo perso la left-associatività per l'operatore $*$ (ma lo stesso sarebbe successo anche per $+$). Il modo migliore per convincersi di queste cose (nonché il nostro suggerimento per il lettore), è quello di provare da sé a cambiare la grammatica, provare a tracciare dei parse tree e vedere cosa succede, un po'come abbiamo fatto noi qui sotto in Fig.8.6.



(a) $T \to T * id$, left-associativa (versione corretta)



(b) Se invece scriviamo $T \to id * T$ perdiamo la left-associatività

Figura 8.6: Esempio di come possiamo alterare Eq.8.5 e vedere i cambiamenti direttamente dalla struttura dei parse trees

È bene far notare, inoltre, che questo procedimento ci ha portato ad aggiungere il non-terminale T alla grammatica. Certo, un non-terminale in più non ha tutto questo grande impatto sull'efficienza degli algoritmi, ma in altre situazioni la scrittura di una grammatica non ambigua che generi un linguaggio equivalente (a quello della grammatica ambigua di partenza, ndr) può rivelarsi davvero complicata e costosa; per questo motivo, spesso si preferisce semplicemente tenere le grammatiche ambigue e gestire i conflitti manualmente, tramite direttive per il parser: il prezzo da pagare per usare una grammatica non ambigua equivalente sarebbe troppo alto.

Esercizio sulla risoluzione dei conflitti

Sia data la seguente grammatica:

$$\begin{aligned} \mathcal{G} : S &\to aAd \mid bBd \mid aBe \mid bAe \\ A &\to c \\ B &\to c \end{aligned} \tag{8.6}$$

Il nostro obiettivo è costruire una tabella di parsing SLR(1) per la grammatica citata, e per farlo, come prima cosa, andiamo a tracciare l'automa caratteristico.

Costruzione dell'automa

1. Inizializziamo lo stato 0. Il suo kernel è:

   $$S' \to \cdot S$$

   Di questo kernel devo calcolare la chiusura:

   $$\begin{aligned} S &\to \cdot aAd \\ S &\to \cdot bBd \\ S &\to \cdot aBe \\ S &\to \cdot bAe \end{aligned}$$

   Essendo questi gli items per lo stato 0, posso identificare tre possibili transizioni (e quindi tre possibili nuovi stati): $\tau(0, S) = 1$, $\tau(0, a) = 2$ e $\tau(0, b) = 3$.

2. Analizziamo ora lo stato 1; il suo kernel è:

   $$S' \to S\cdot$$

   Non serve calcolare la chiusura di questo item, in quanto è formata solamente da sé stesso; dobbiamo però evidenziare il fatto che questo è lo stato contenente l'accepting item.

3. Analizziamo dunque lo stato 2, il cui kernel è popolato da questi item:

   $$\begin{aligned} S &\to a \cdot Ad \\ S &\to a \cdot Be \end{aligned}$$

   e di cui è necessario calcolare la chiusura:

   $$\begin{aligned} A &\to \cdot c \\ B &\to \cdot c \end{aligned}$$

Anche nel caso dello stato 2 possiamo osservare la presenza di tre transizioni e quindi di tre possibili nuovi stati che sono $\tau(2, A) = 4$, $\tau(2, B) = 5$ e $\tau(2, c) = 6$.

4. Passiamo ora allo stato 3, il cui kernel risulta:

$$S \to b \cdot Bd$$
$$S \to b \cdot Ae$$

Su questo kernel si presenta la possibilità di effettuare una chiusura, che risulta essere:

$$A \to \cdot c$$
$$B \to \cdot c$$

In questo caso è necessario prestare un po' più di attenzione, perché sebbene vi siano tre transizioni uscenti dallo stato 3, dovremo creare soltanto 2 nuovi stati: $\tau(3, B) = 7$, $\tau(3, A) = 8$ e $\tau(3, c) = 6$. Per convalidare questa affermazione è possibile osservare che il kernel dello stato 6 coinciderà esattamente con quello della transizione $\tau(2, c) = 6$.

5. Passiamo allo stato 4, il cui kernel è:

$$S \to aA \cdot d$$

La chiusura di tale stato non porta nuovi elementi, tuttavia è comunque possibile definire la transizione $\tau(4, d)$ allo stato 9.

6. Analizzo ora lo stato 5, e il suo kernel è:

$$S \to aB \cdot e$$

Tale stato non necessita di chiusura e ci fornisce la transizione $\tau(5, e)$ verso lo stato 10.

7. Procedendo come abbiamo fatto finora, il kernel per lo stato 6 è:

$$A \to c\cdot$$
$$B \to c\cdot$$

la cui chiusura non ci porta nulla di nuovo e possiamo osservare che nello stato 6 sono presenti due reducing: questo significa che avremo un conflitto r/r.

8.4.2 I limiti del parsing SLR(1)

Potremmo continuare senza fare ulteriori elucubrazioni, ma non possiamo non porre l'attenzione sullo stato appena analizzato: il fatto che vi possano essere due reducing items all'interno di un singolo stato non è necessariamente motivo di conflitti, in quanto la presenza di conflitti dipende dalla funzione di lookahead che stiamo utilizzando. Nel caso del parsing di tipo SLR(1) sappiamo che è possibile inserire un'operazione di `reduce` $A \to \beta$ in $M[P, Y]$ nel caso in cui P sia un stato contenente un reducing item e per tutti quei terminali $Y \in \mathcal{LA}(P, A \to \beta) = follow(A)$: in questo caso $follow(A) = follow(B) = \{d, e\}$[7], il che significa che avremo almeno due celle dove sono presenti dei conflitti nella nostra parsing table finale.

La situazione è ben rappresentata dalla figura 8.7 che riporta un pezzo dell'automa caratteristico che deriva da questa analisi.

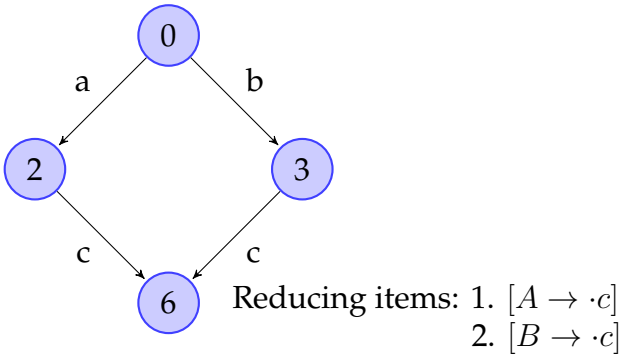


Figura 8.7: Dettaglio dell'automa di tipo LR(0) per la grammatica 8.6, si noti come con una c-transizione si possa arrivare in 6 sia da 3 che da 2

Tuttavia, in questo caso la colpa non è della grammatica che non è per nulla ambigua (produce solo 4 parole), ma va ricercata piuttosto nel tipo di parsing adoperato, fin troppo grossolano: invece di accorpare tutto all'interno di un singolo stato (riguardiamo lo stato 6), non sarebbe meglio avere due stati differenti?

Ovviamente, lo stato 2 e lo stato 3 "ricordano" delle informazioni differenti e per poter risolvere questa indecisione è necessario considerare in quale modo si è arrivati all'item di riduzione. Ad esempio, nel caso in cui avessimo letto fino ad ora la parola ac e ci trovassimo quindi in dubbio su che tipo di operazione di reduce effettuare, basterebbe semplicemente poter osservare quale fosse il prossimo simbolo in lettura:

[7]Suvvia, un po' di fiducia, questi calcoli li abbiamo fatti davvero.

- effettuiamo `reduce` $A \to c$ nel caso in cui il prossimo simbolo sia d in quanto è possibile derivare da S solamente una parola che inizi per a e termini con d, del tipo aAd;
- effettuiamo `reduce` $B \to c$ nel caso in cui il prossimo simbolo sia e in quanto è possibile derivare da S solamente una parola che inizi per a e termini con e, del tipo aBe.

Abbiamo modo di conservare queste informazioni nel nostro automa e risparmiarci questa fastidiosa situazione? Certo che l'abbiamo, e la risposta è quella che, oramai, abbiamo capito tutti: dobbiamo usare degli LR(1)-items e cambiare tipologia di parsing. Restate sintonizzati, ne vedremo delle belle.

Capitolo 9

Analisi sintattica: parsing bottom-up LR(1) e LALR(1)

9.1 L'automa caratteristico LR(1)

Riprendiamo il filo del discorso: nel precedente capitolo siamo andati a discutere tutti i passaggi che, partendo da una grammatica, ci conducono ad eseguire il parsing bottom-up. Consigliamo di tornare a dare un occhio allo schema presentato in sottosezione 8.1.1. Ci eravamo lasciati con una questione alquanto spinosa: avevamo constato che è possibile ritrovarsi con due reducing items in uno stesso stato anche se la grammatica di partenza non è ambigua, e questa situazione potrebbe essere risolta conservando, per ogni stato, delle informazioni rispetto a quali simboli potrebbero seguirgli in lettura. Ci eravamo resi conto di non aver modo di conservare e impiegare efficacemente queste informazioni con la tecnica di costruzione dell'automa SLR(1), il che lascia pensare solo a una cosa: è il momento di abbandonare questa tecnica e guardare a soluzioni più complesse, certo, ma senza dubbio più efficaci. È il momento di affrontare la costruzione di un automa caratteristico LR(1).

9.1.1 Stati

Gli LR(1)-items Per prima cosa dobbiamo ricordarci che la principale differenza sta proprio in cosa popola gli stati dei diversi automi: per gli automi SLR(1) utilizzavamo gli LR(0)-items, gli automi LR(1) invece utilizzano gli LR(1)-items.

Forma Gli LR(1)-items hanno questa forma:

$$[A \to \alpha \cdot B\beta, \Delta] \tag{9.1}$$

Possiamo vedere subito che un LR(1)-item è una tupla di due elementi:

1. il primo è un LR(0)-item $(A \to \alpha \cdot B\beta)$;
2. il secondo è un insieme di caratteri (Δ) detto **lookahead set**.

9.1.2 Chiusura di un insieme di LR(1)-item

Quando andiamo a calcolare la chiusura di un item di tipo LR(1) utilizziamo la $closure_1(item)$ e non la $closure_0(item)$, che funziona esattamente come la $closure_0$ ma in più ci permette di aggiornare l'insieme Δ, il lookahead set. Questo insieme ci torna poi utile quando dobbiamo inserire mosse di riduzione in qualche stato dell'automa caratteristico LR(1): il lookahead set specifica quali caratteri ci dobbiamo aspettare di vedere in lettura quando stiamo per utilizzare una mossa di reduce.

Funzionamento Ma specifichiamo ora meglio come è strutturata l'operazione di $closure_1(item)$:

- la prima parte della chiusura è in tutto e per tutto uguale alla $closure_0(item)$, quindi quando si calcola una $closure_1(item)$ la prima cosa che si fa è appunto calcolare la $closure_0(item)$, utilizzando la parte LR(0) dell'item LR(1) che stiamo chiudendo;
- la seconda parte prevede di aggiornare l'insieme di lookahead, che viene aggiunto alla $closure_0(item)$ appena calcolata.

Di fatto con questo metodo di calcolo l'informazione contenuta nell'insieme Δ di un LR(1)-item viene tramandata agli item che ne derivano grazie alla seconda parte della procedura $closure_1$, così se si segue una derivazione lungo tutto l'albero delle produzioni guardando a Δ si ha sempre sotto controllo quali simboli ci si aspetta di leggere in un certo stato quando si deve effettuare una mossa di riduzione.

Calcolo Come facciamo a calcolare la chiusura degli insiemi di LR(1)-items? Il metodo di calcolo che ci viene presentato è un classico calcolo risolvibile grazie al teorema del punto fisso:

Definizione 9.1.1. *Sia P un insieme di LR(1)-item, la $closure_1(P)$ identifica il più piccolo insieme di item, con il più piccolo lookahead-set, che soddisfa la seguente equazione:*

$$\begin{aligned} closure_1(P) = P \cup \{[B \to \cdot\gamma, \Gamma] : [A \to \alpha \cdot B\beta, \Delta] \in closure_1(P) \wedge \\ B \to \gamma \in \mathcal{P}' \wedge \\ first(\beta\Delta) \subseteq \Gamma\} \end{aligned}$$

dove $first(\beta\Delta) = \cup_{d\in\Delta} first(\beta d)$ *e ricordiamo che* $\mathcal{P}'$ *è ricavato dall'insieme delle produzioni* $\mathcal{P}$ *aggiungendo la produzione* $S' \to S$*.*

Algoritmo Dal momento che la Def.9.1.1 non risponde molto bene al richiamo degli attributi "semplice e intuitiva", potrebbe essere una buona idea dare un'occhiata alla sua formulazione algoritmica in Alg.16.

Algorithm 16: SET closure$_1$(SET P)

```
1  Tag each item in P as unmarked
2  while ∃item ∈ P : item is unmarked do
3      mark(item)
4      if item is in form [A → α · Bβ, Δ] then
5          Δ1 ← ⋃_{d∈Δ} first(βd)
6          foreach B → γ ∈ P' do
7              if B → ·γ ∉ prj(P) then
8                  add [B → ·γ, Δ1] to P as unmarked item
9              else
10                 if [B → ·γ, Γ] ∈ P ∧ Δ1 ⊈ Γ then
11                     Update [B → ·γ, Γ] to [B → ·γ, Γ ∪ Δ1] ∈ P
12                     tag [B → ·γ, Γ ∪ Δ1] as unmarked
13 return P
```

In questo algoritmo si deve fare conto che $B \to \cdot\gamma \notin prj(P)$ significa che non è ancora presente in $closure_1(P)$ una produzione $B \to \cdot\gamma$. Di fatto quello che succede all'interno del `foreach` centrale è che se l'elemento è nuovo vado ad aggiungerlo con il suo Δ, se l'elemento è già presente vado a vedere se è il caso di aggiungere al Δ di questa qualche elemento. Il procedimento potrebbe risultare un po' ostico, ma sarà tutto più chiaro dopo qualche esercizio esplicativo, come al solito.

Esercizio di calcolo di $closure_1(P)$

Facendo riferimento alla grammatica dell'esercizio precedente (riportata qui sotto), calcolare tutti gli stati dell'automa caratteristico LR(1).

$$\begin{aligned} \mathcal{G} : S &\to aAd \mid bBd \mid aBe \mid bAe \\ A &\to c \\ B &\to c \end{aligned} \tag{9.2}$$

Descrizione della procedura per lo stato 0 Partiamo con l'inizializzazione dello stato 0, che conterrà la $closure_1(\{[S' \to \cdot S, \{\$\}]\})$; questa inizializzazione sottolinea il fatto che, che una volta che avremo raggiunto il particolare stato in cui avremo l'accepting item $S' \to S\cdot$, vorremo vedere il \$ che viene utilizzato come terminatore per definire una parola appartenente al linguaggio. Quindi, che si fa ora?

Secondo la Def.9.1.1, la $closure_1(P)$ va inizializzata con P stesso, ovvero $[S' \to \cdot S, \{\$\}]$. In seguito dobbiamo andare a prendere, all'interno della $closure_1(P)$ (che è parziale, non ancora completa), tutti gli elementi in forma $[A \to \alpha \cdot B\beta, \Delta]$ e dovremo aggiungere alla stessa $closure_1(P)$ tutti quegli item che soddisfano la forma $[B \to \cdot\gamma, \Gamma]$, dove Γ è un insieme calcolato come descritto dall'algoritmo; niente paura, in seguito sarà tutto più chiaro.

Nel nostro caso l'unico item in $closure_1(P)$ è $[S' \to \cdot S, \{\$\}]$, che corrisponde alla forma desiderata ($[A \to a \cdot B\beta, \Delta]$), quindi possiamo assumere $\Delta = \{\$\}$ e $\beta = \varepsilon$; di conseguenza, $first(\varepsilon\$) = first(\$) = \{\$\} \subseteq \Gamma$. Gli item che otteniamo da $closure_1(\{[S' \to \cdot S, \{\$\}]\})$ saranno dunque:

$$[S' \to \cdot S, \{\$\}]$$
$$[S \to \cdot aAd, \{\$\}]$$
$$[S \to \cdot bBd, \{\$\}]$$
$$[S \to \cdot aBe, \{\$\}]$$
$$[S \to \cdot bAe, \{\$\}]$$

Quanto abbiamo appena ottenuto è la chiusura dell'insieme che costituisce lo stato 0 del nostro automa caratteristico di tipo LR(1). Come già detto prima, la prima parte di questi item è esattamente corrispondente all'item LR(0) che avremmo trovato utilizzando SLR parsing, la seconda parte (ovvero il Δ) è l'unica novità introdotta dalla procedura LR(1).

Iterazione della procedura sulle transizioni È da notare anche che la componente del lookahead-set è usata solo in caso di riduzioni, e non incide sulle transizioni, per cui il procedimento rimarrà sotto quel punto di vista inalterato rispetto al metodo SLR. Di conseguenza, possiamo affermare la presenza, di tre transizioni che portano a tre stati differenti: $\tau(0, S) = 1$, $\tau(0, a) = 2$ e $\tau(0, b) = 3$. A questo punto è possibile procedere come abbiamo sempre fatto, prestando attenzione al calcolo della $closure_1(S)$:

1. Proseguiamo osservando lo stato $\tau(0, S) =$1. Il suo kernel è:

$$[S' \to S\cdot, \{\$\}]$$

dobbiamo dunque calcolare $closure_1(\{[S' \to S\cdot, \{\$\}]\})$, per cui sappiamo che $B = \varepsilon$ e dunque non possiamo fare altro se non appuntarci che lo stato 1 contiene l'Accepting Item.

2. Analizziamo dunque lo stato $\tau(0, a)$ =2, il cui kernel è popolato da questi item:

$$[S \to a \cdot Ad, \{\$\}]$$
$$[S \to a \cdot Be, \{\$\}]$$

entrambi soddisfano la forma $[A \to \alpha \cdot B\beta, \Delta]$. Nel primo caso abbiamo le seguenti corrispondenze: $A = S$, $\alpha = a$, $B = A$, $\beta = d$ e $\Delta = \$$; andiamo a cercare nella nostra grammatica se sono presenti derivazioni in forma $B \to \gamma$. Troviamo $A \to c$, quindi grazie alla formula del calcolo di $closure_1(P)$ sappiamo che dobbiamo aggiungere agli item dello stato 2 un item formato così: $[A \to \cdot c, \Gamma]$, dove $\Gamma = first(\beta\Delta) = first(d\$) = \{d\}$; in definitiva, aggiungiamo l'item $A \to \cdot c, \{d\}$.

Abbiamo detto che anche la seconda produzione ($[S \to a \cdot Be, \{\$\}]$) soddisfa la forma $[A \to \alpha \cdot B\beta, \Delta]$, quindi con lo stesso procedimento appena adottato possiamo trovare la chiusura di $[S \to a \cdot Be, \{\$\}]$: in questo caso le corrispondenze sono: $A = S$, $\alpha = a$, $B = B$, $\beta = e$ e $\Delta = \$$, la produzione da analizzare è $B \to c$ e l'item che ne ricaviamo è $[B \to \cdot c, \Gamma]$, dove $\Gamma = first(\beta\Delta) = first(e\$) = \{e\}$.

In sostanza, lo stato 2 contiene questi LR(1) items:

$$[S \to a \cdot Ad, \{\$\}]$$
$$[S \to a \cdot Be, \{\$\}]$$
$$[A \to \cdot c, \{d\}]$$
$$[B \to \cdot c, \{e\}]$$

Come succedeva nell'esempio con il parsing LR(0), nello stato 2 possiamo osservare la presenza di tre transizioni e quindi di tre possibili nuovi stati che sono $\tau(2, A) = 4$, $\tau(2, B) = 5$ e $\tau(2, c) = 6$.

...

6. Ovviamente la parte interessante dell'esercizio è verificare se siamo riusciti a risolvere il conflitto che occorreva nel parsing SLR(0). In modo pressoché analogo a quanto accadeva, abbiamo che il kernel per lo stato 6 è :

$$[A \to c\cdot, \{d\}]$$
$$[B \to c\cdot, \{e\}]$$

Anche questa volta nello stato sono presenti due item di riduzione, ma non troviamo più il conflitto che avevamo con il parsing LR(0), perché in questo caso la riduzione $A \to c$ si applica solo se nell'input buffer si sta leggendo d, mentre la riduzione $B \to c$ si applica solo nel caso in cui nell'input *buffer* si sta leggendo e. Questo implica che nello stato 6, grazie al parsing LR(1), è stata eliminata l'ambiguità del conflitto r/r.

Facciamone un altro, dai

Sia data la seguente grammatica:

$$\begin{aligned} \mathcal{G} : S &\to L = R \mid R \\ L &\to *R \mid id \\ R &\to L \end{aligned} \tag{9.3}$$

Inizializziamo lo stato 0 ponendo come suo kernel:

$$[S' \to \cdot S, \{\$\}]$$

Calcoliamo $closure_1(0)$. Questo item soddisfa la forma $[A \to \alpha \cdot B\beta, \Delta]$, con le seguenti corrispondenze: $B = S$, $\beta = \varepsilon$, $\Delta = \{\$\}$ (e quindi $\Gamma = first(\beta\Delta) = first(\varepsilon\$) = \{\$\}$). Le due produzioni di B (ovvero di S) in questo caso sono $S \to L = R \mid R$, quindi aggiungo i due item LR(1) seguenti:

$$\begin{aligned} &[S \to \cdot L = R, \{\$\}] \\ &[S \to \cdot R, \{\$\}] \end{aligned} \tag{9.4}$$

A questo punto la chiusura non è completa in quando abbiamo aggiunto due elementi che possono essere presi in considerazione nella definizione ricorsiva della chiusura, ovvero soddisfano anch'essi la famosa forma $[A \to \alpha \cdot B\beta, \Delta]$. Analizziamo dunque $[S \to \cdot L = R, \{\$\}]$, le sue corrispondenze sono: $B = L$, $\beta = \ = R$, $\Delta = \{\$\}$ (e quindi $\Gamma = first(\beta\Delta) = first(= R\$) = \{=\}$); le produzioni di L sono $L \to *R \mid id$, quindi arriviamo a generare i seguenti LR(1)-items:

$$\begin{aligned} &[L \to \cdot * R, \{=\}] \\ &[L \to \cdot id, \{=\}] \end{aligned}$$

che non presentano altre possibili espansioni (grazie al cielo).

Ci rimane ora da analizzare $[S \to \cdot R, \{\$\}]$, che soddisfa la nostra formosa forma e presenta le seguenti corrispondenze: $B = R$, $\beta = \{\varepsilon\}$, $\Delta = \{\$\}$ (e

quindi $\Gamma = first(\beta\Delta) = first(\varepsilon\$) = \{\$\}$), inoltre le produzioni di R sono $R \to L$, quindi otteniamo il seguente LR(1)-items:

$$[R \to \cdot L, \{\$\}]$$

questo item presenta il marker davanti a un non-terminale; può generare nuovi item per la chiusura che stiamo calcolando? Nonostante la produzione con driver L sia già stata analizzata (vedi Eq.9.4), è necessario eseguire nuovamente la sua chiusura in quanto ha un lookahed-set differente (nel caso precedente $\beta = = R$, mentre ora $\beta = \varepsilon$), il che ci porta a ripetere la chiusura per l'item $[R \to \cdot L, \{\$\}]$, ricavando i seguenti LR(1)-items:

$$[L \to \cdot * R, \{\$\}]$$
$$[L \to \cdot id, \{\$\}]$$

Visto che però avevamo già inserito due LR(1)-item con lo stesso LR(0)-item (la parte $L \to \cdot qualcosa$) possiamo semplicemente accrescere i lookahead-set corrispondenti. Il risultato finale per la chiusura degli item dello stato iniziale è:

$$[S' \to \cdot S, \{\$\}]$$
$$[S \to \cdot L = R, \{\$\}]$$
$$[S \to \cdot R, \{\$\}]$$
$$[L \to \cdot * R, \{\$, =\}]$$
$$[L \to \cdot id, \{\$, =\}]$$
$$[R \to \cdot L, \{\$\}]$$

Bene, ora abbiamo fatto un po' di allenamento con il calcolo degli stati, ma dobbiamo ricordare che non è questo il nostro obiettivo finale, noi vogliamo ricavare l'automa caratteristico!

9.1.3 Costruzione di un automa caratteristico LR(1)

Finalmente possiamo spiegare come si costruisce un automa caratteristico LR(1).

Idea L'idea di base è semplice:

1. costruiamo lo stato di partenza con l'item $[S' \to \cdot S, \$]$;
2. ricorsivamente aggiungiamo gli stati che si possono raggiungere dagli stati presenti nell'automa;

3. aggiungamo le transizioni che ci permettono di passare da uno stato all'altro.

E come facciamo a capire quali stati sono raggiungibili a partire da un certo stato P? Se uno stato P contiene un item nella forma $[A \to \alpha \cdot Y\beta, \Delta]$ allora esiste una transizione da P ad uno stato Q che contiene l'item $[A \to \alpha Y \cdot \beta, \Delta]$; inoltre, siccome Q contiene $[A \to \alpha Y \cdot \beta, \Delta]$, esso contiene anche tutti gli item in $closure_1([A \to \alpha Y \cdot \beta, \Delta])$.

Algoritmo Proviamo ora a chiarire tale procedura tramite la sua scrittura in forma algoritmica, si veda 17.

Algorithm 17: LR(1)-automatonConstruction(GRAMMAR $\mathcal{G}$)

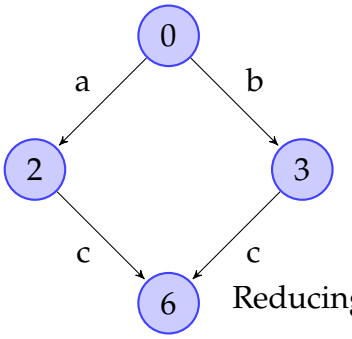
```
1  Initialize che collection 𝒬 to contain P_0 = chiusura_1({[S' → ·S, {$}]})
2  tag P_0 as unmarked
3  while ∃ a state P ∈ 𝒬 : P is unmarked do
4      mark(P)
5      foreach Y on the right side of the marker in some item of P do
6          Compute in tmp the kernel-set of the Y-target of P
7          if 𝒬 already contains a state Q whose kernel is tmp then
8              Let Q be the Y-target of P
9          else
10             Add chiusura_1(tmp) to 𝒬 as unmarked state
11             Let chiusura_1(tmp) be the Y-target of P
```

Esempio di costruzione

Indovinate un po' con quale grammatica stiamo per andare a osservare cosa si ottiene con questa procedura? Oh sì! proprio lei:

$$\begin{aligned} \mathcal{G} : S &\to aAd \mid bBd \mid aBe \mid bAe \\ A &\to c \\ B &\to c \end{aligned} \tag{9.5}$$

Ora chiederemo al lettore di fare uno sforzo mnemonico poderoso e di ricordare che in passato, quando abbiamo provato a costruire l'automa caratteristico SLR(1) per questa grammatica ci siamo trovati ad avere la situazione rappresentata in figura 8.7, che riprtiamo qui sotto per comodità.



Reducing items: 1. $[A \to \cdot c]$
2. $[B \to \cdot c]$

Figura 9.1: Automa di tipo LR(0), si noti il conflitto sullo stato 6

Abbiamo già calcolato quali sono gli stati LR(1) per la grammatica in questione in un esercizio precedente (vedi 13), ed applicando l'algoritmo per la costruzione degli automi LR(1) arriviamo ad ottenere il seguente automa (parziale):

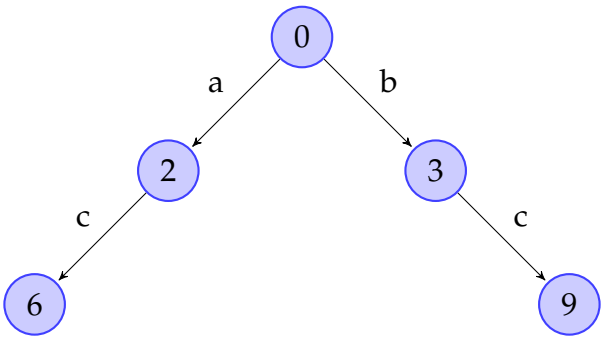


Figura 9.2: Automa di tipo LR(1), si noti che il conflitto sullo stato 6 è stato eliminato

Ma siamo sicuri di aver effettivamente risolto il problema? per verificarlo dobbiamo costruire la parsing table!

Nonostante qui la prenderemo come magia nera per non spostare il focus dalla costruzione dell'automa in sé, possiamo anticipare che la costruzione della tabella di parsing LR(1) è semplice da ottenere, perché si crea esattamente come la sua corrispondente SLR(1), con queste uniche due differenze:

- l'automa caratteristico è di tipo LR(1);
- la lookahead function utilizzata è la seguente: per ogni $[A \to \beta\cdot, \Delta] \in P$, $\mathcal{LA}(P, A \to \beta) = \Delta$.

Naturalmente una grammatica $\mathcal{G}$ è di tipo LR(1) se la sua tabella di parsing LR(1) non presenta conflitti. Sbirciamo quindi cosa sarebbe successo costruendo la parsing table LR(1) per la nostra grammatica preferita.

Innanzitutto, l'automa caratteristico LR(1) si presenta come in Fig.9.3

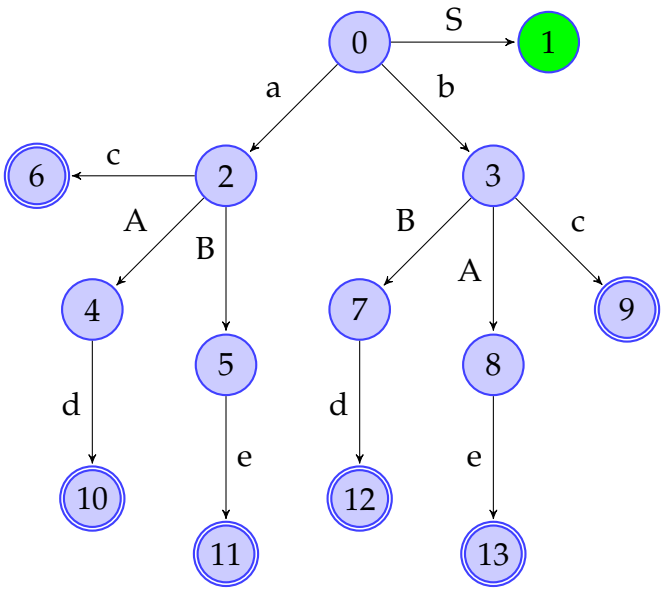


Figura 9.3: Automa di tipo LR(1) completo

Negli stati 6 e 9, che ci avrebbero causato problemi con la costruzione di tipo LR(0), ora troviamo i seguenti item:

- Stato 6

$$A \to c\cdot, d$$
$$B \to c\cdot, e$$

- Stato 9

$$A \to c\cdot, e$$
$$B \to c\cdot, d$$

Quindi, seguendo le regole di compilazione della tabella di parsing LR(1) avremmo:

- nella casella M[6,d] la riduzione $A \to c\cdot, d$;

- nella casella M[6,e] la riduzione $B \to c\cdot, e$;
- nella casella M[9,e] la riduzione $A \to c\cdot, e$;
- nella casella M[9,d] la riduzione $B \to c\cdot, d$;

Festa! Festa! non ci sono più conflitti! È però notevole il fatto che il numero di stati sia aumentato parecchio: questo è quello che succede quando si va ad utilizzare tecniche di parsing più precise e raffinate.

9.2 Costruzione di una tabella di parsing LR(1)

Chi avesse seguito con sufficiente costanza e attenzione a questo punto dovrebbe sapere che non ci sarà alcuna nuova informazione in questa sezione, perché la procedura di costruzione della tabella di parsing è la medesima per tutte le tipologie di parsing bottom-up. Di conseguenza, se qualcuno avesse bisogno di una rinfrescata, suggeriamo di tornare al capitolo precedente prima di proseguire; per tutti gli altri, rimbocchiamoci le maniche e andiamo a mettere le mani in pasta su queste tabelle di parsing.

9.2.1 La grammatica dei puntatori

La grammatica che scegliamo come esempio è questa:

$$\begin{aligned} &S \to L = R \mid R \\ &L \to *R \mid id \\ &R \to L \end{aligned} \tag{9.6}$$

Questa è una grammatica che ci parla di puntatori, ma capiremo in seguito cosa significa ciò, per ora concentriamoci sulla sua risoluzione.

Definizione stati e costruzione dell'automa

Il primo passo, naturalmente, è la costruzione dell'automa caratteristico. Per fare questo è necessario prima definire come sono fatti gli stati del nostro automa, ossia quali LR(1)-items comprenderanno; per fare questo, sappiamo che abbiamo la nostra fida $closure_1()$. Aggiungiamo la produzione canonica $S' \to S$ e cominciamo.

Stato 0 Il kernel dello stato è $S' \to \cdot S$, a cui andiamo subito ad aggiungere il risultato del calcolo di $closure_1(S)$. Ricordiamo che il lookahead set Γ si calcola come first($\beta\Delta$); nel caso di $closure_1(S)$ abbiamo $\beta = \varepsilon$ e $\Delta = \{\$\}$.

$$\begin{aligned} Kernel: \quad & S' \to \cdot S, \{\$\} \\ closure_1(S): \quad & S \to \cdot L = R, \{\$\} \\ & S \to \cdot R, \{\$\} \end{aligned}$$

Abbiamo calcolato la chiusura per S, ma adesso dobbiamo calcolarla anche per L e R, dal momento figurano in un qualche item con il marker $\cdot$ davanti. Per quanto riguarda il calcolo di Γ:

- in $closure_1(L)$ abbiamo che $\beta = \{= R\}$ e $\Delta = \{\$\}$;
- in $closure_1(R)$ abbiamo che $\beta = \varepsilon$ e $\Delta = \{\$\}$;

$$\begin{aligned} closure_1(L): \quad & L \to \cdot * R, \{=\} \\ & L \to \cdot id, \{=\} \\ closure_1(R): \quad & R \to \cdot L, \{\$\} \end{aligned}$$

Non è ancora finita purtroppo, perché trovato un altro item con il marker $\cdot$ davanti alla L; che facciamo, dobbiamo ricalcolarla? Sì, dobbiamo farlo assolutamente! Nel caso precedente la produzione $S \to \cdot L = R$ aveva $\beta = \{= R\}$, mentre in questo caso la produzione è $R \to L$ e ha $\beta = \varepsilon$, per cui il lookahead set delle produzioni di L potrebbe essere diverso in questo secondo caso, e difatti è proprio questo che succede, perché il calcolo della chiusura ci dice questo:

$$\begin{aligned} & L \to \cdot * R, \{\$\} \\ & L \to \cdot id, \{\$\} \end{aligned}$$

Quindi lo stato 0 contiene i seguenti items, e presenta queste transizioni:

items:	transizioni:
$S' \to \cdot S, \{\$\}$	$\tau(0, S) = 1$
$S \to \cdot L = R, \{\$\}$	$\tau(0, L) = 2$
$S \to \cdot R, \{\$\}$	$\tau(0, R) = 3$
$L \to \cdot * R, \{=, \$\}$	$\tau(0, *) = 4$
$L \to \cdot id, \{=, \$\}$	$\tau(0, id) = 5$
$R \to \cdot L, \{\$\}$	

Stato 1 Andiamo adesso ad analizzare lo stato 1, che ha kernel

$$S' \to S\cdot, \{\$\}$$

È l'accepting item, non c'è nulla da dire oltre a questo (e soprattutto nulla da calcolare).

Stato 2 Lo stato 2 ha questo kernel:

$$S \to L\cdot = R, \{\$\}$$
$$R \to L\cdot, \{\$\}$$

Questo stato è già chiuso, perché nel primo item il marker $\cdot$ si trova davanti al terminale $=$, mentre invece il secondo item ci propone la riduzione $[R \to L, \{\$\}]$; inoltre, il primo item ci propone la transizione $\tau(2, =)$. Notiamo anche che in questo stato abbiamo sia uno shift che una riduzione, teniamolo a mente per dopo.

Stato 3 Passiamo allo stato 3, e il kernel ci porta delle belle notizie:

$$S \to R\cdot, \{\$\}$$

Questo stato è già chiuso e l'unica cosa che ci regala è la mossa di reduce $[S \to R\cdot, \{\$\}]$.

Stato 4 Il kernel di questo stato, purtroppo, non è altrettanto lungimirante:

$$L \to *\cdot R, \{=, \$\}$$

Siamo infatti costretti a calcolare la chiusura $closure_1(R)$, che ci porta a inserire l'item $R \to \cdot L, \{=, \$\}$ allo stato; quest'ultimo, dal canto suo, ci impone di inserire allo stato la chiusura $closure_1(L)$, dal momento che abbiamo il marker $\cdot$davanti al non-terminale L. A fine del processo lo stato appare così:

$$L \to *\cdot R, \{=, \$\}$$
$$R \to \cdot R, \{=, \$\}$$
$$L \to \cdot * R, \{=, \$\}$$
$$L \to \cdot id, \{=, \$\}$$

Le mosse di shift che troviamo sono invece:

$$\tau(4, R)$$
$$\tau(4, L)$$
$$\tau(4, *)$$
$$\tau(4, id)$$

Stato 5 Abbiamo nuovamente uno stato tranquillo. Il kernel è:

$$L \to id\cdot, \{=, \$\}$$

Questo ci offre la rispettiva riduzione ($[L \to id\cdot, \{=, \$\}]$), senza alcuna mossa di shift.

Stato 6 Questo stato viene calcolato seguendo la transizione $\tau(2, =)$. Dal momentoche abbiamo il marker $\cdot$ davanti al non-terminale R, dobbiamo calcolare $closure_1(R)$, che in omaggio ci porta anche il calcolo di $closure_1(L)$.

$$\begin{aligned} Kernel: &\quad S \to L = \cdot R, \{\$\} \\ closure_1(R): &\quad R \to \cdot L, \{\$\} \\ closure_1(L): &\quad L \to \cdot * R, \{\$\} \\ &\quad L \to \cdot id, \{\$\} \end{aligned}$$

Troviamo anche le seguenti mosse di shift:

$$\tau(6, R)$$
$$\tau(6, L)$$
$$\tau(6, *)$$
$$\tau(6, id)$$

Stato 7 Calcoliamo questo stato grazie allo shift $\tau(4, R)$. Il kernel:

$$L \to *R\cdot, \{=, \$\}$$

Un altro stato tranquillo insomma, che ci offre solamente una riduzione ($[L \to *R\cdot, \{=, \$\}]$) e nulla di più.

Stato 8 Il copione è esattamente lo stesso di quanto visto sopra: calcoliamo questo stato grazie allo shift $\tau(4, L)$. Il kernel:

$$R \to L\cdot, \{=, \$\}$$

Andiamo avanti lisci, abbiamo la nostra riduzione $[L \to *R\cdot, \{=, \$\}]$ e nulla di più.

Stato 9 Anche in questo stato abbiammo del lavoro da fare. Ci arriviamo tramite $\tau(4, *)$, e il kernel si presenta così:

$$L \to * \cdot R, \{=, \$\}$$

Ma fermi... è esattamente il kernel dello stato 4! Questo vuol dire che troviamo subito una transizione $\tau(4, *) = 4$, il che significa che lo stato 4 ha un self loop per $*$.

Proviamo allora a ricavare uno stato 9 dalla transizione $\tau(4, id)$. Il kernel è questo:

$$L \to id\cdot, \{=, \$\}$$

E niente, è di nuovo un altro kernel che abbiamo già visto nello stato 5, per cui concludiamo che $\tau(4, id) = 5$.

Facciamo un altro tentativo, questa volta con $\tau(6, R)$. Il kernel è:

$$S \to L = R\cdot, \{\$\}$$

Fatto, finalmente siamo riusciti a creare un nuovo stato con questa transizione $\tau(6, R) = 9$. Non abbiamo altre chiusure da calcolare, e torniamo a casa con la riduzione $[S \to L = R\cdot, \{\$\}]$.

Stato 10 Adesso seguiamo la transizione $\tau(6, L)$. Il kernel è:

$$R \to L\cdot, \{\$\}$$

È vero, abbiamo già uno stato con questo kernel (lo stato 8), ma in quel caso il lookahead set è diverso, per cui creiamo ugualmente un nuovo stato. Abbiamo una mossa di shift $\tau(6, L) = 10$ e la riduzione data dal kernel $[R \to L\cdot, \{\$\}]$.

Stato 11 La situazione è molto simile a quella descritta sopra. Seguendo la transizione $\tau(6, *)$ troviamo:

$$L \to *\cdot R, \{\$\}$$

Questo è lo stesso kernel dello stato 9, ma per via del differente lookahead set creiamo comunque un nuovo stato ($\tau(6, *) = 11$). Dobbiamo anche calcolare $closure_1(R)$ che, anche questa volta, forza il calcolo di $closure_1(L)$; lo stato sarà infine composto di questi items:

$$\begin{aligned} Kernel: &\quad L \to *\cdot R, \{\$\} \\ closure_1(R): &\quad R \to \cdot L, \{\$\} \\ closure_1(L): &\quad L \to \cdot * R, \{\$\} \\ &\quad L \to \cdot id, \{\$\} \end{aligned}$$

Notiamo che, con l'eccezione dell'ultimo, tutti gli item sono stati già visitati, e già sappiamo dove portano ($R \to \cdot L, \{\$\}$ a $\tau(11, 2) = 10$ e $L \to \cdot * R, \{\$\}$ a $\tau(11, *) = 11$, self loop), per cui ci risparmiamo di analizzare in futuro quelle mosse di shift. Ci sentiamo anche di anticipare che pure l'ultima transizione, $\tau(11, id)$, non porterà a un nuovo stato, poiché risulterà essere un collegamento allo stato 12 ($\tau(11, id) = 12$).

Stato 12 Uh, per fortuna questo stato è tranquillo. Stiamo infatti analizzando la transizione $\tau 11, R$ e il kernel è:

$$L \to id\cdot, \{\$\}$$

E, come ormai sappiamo, ci propone solamente la rispettiva mossa di riduzione.

Stato 13 Dulcis in fundo, analizziamo la transizione $\tau(11, R)$. Il kernel è:

$$L \to *R\cdot, \{\$\}$$

Lo stato 7 ha lo stesso kernel LR(0), ma ancora, il lookahead set è diverso, per cui creiamo comunque un nuovo stato $\tau(11, R) = 13$. Nessuna nuova transizione, solamente la riduzione proposta dal kernel.

Rimarrebbe da analizzare $\tau(11, id)$ ma, come già anticipato, conduce allo stato 12, per cui non c'è bisogno di aggiungere un quattordicesimo stato al nostro automa.

Costruzione dell'automa Ora che abbiamo gli stati non c'è più nulla di diverso rispetto a quello che facevamo prima, per cui possiamo terminare con la costruzione dell'automa caratteristico. Quest'ultimo (Fig.9.4) è alquanto complicato e pare una strana mappa del tesoro, ma ce l'aspettavamo: abbiamo già sottolineato come e perché gli automi LR(1) sono più complessi di un corrispondente SLR(1).

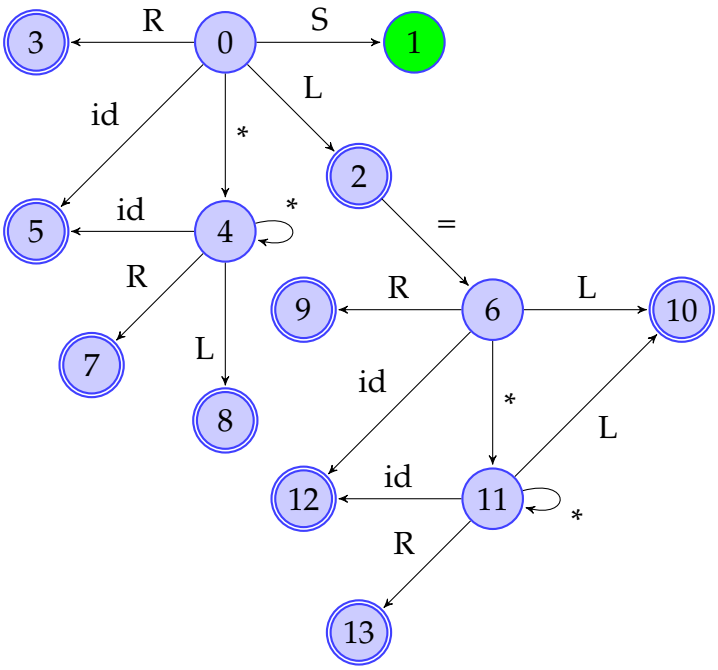


Figura 9.4: Automa caratteristico LR(1) per Eq.9.6

A questo punto potremmo chiederci: la grammatica Eq.9.6 è effettivamente LR(1)? Non è che abbiamo fatto fatica per niente? Per rispondere, la prima cosa che ci viene in mente è costruire la tabella di parsing a partire dal nostro automa, ma possiamo essere più scaltri di così.

La grammatica è LR(1)? Infatti, possiamo fin da subito escludere la presenza di conflitti r/r (ossia reduce/reduce), perché nessuno stato del nostro automa presenza due reducing items.

Anche per verificare la presenza di conflitti s/r (shift/reduce) abbiamo una furberia simile, anche se un po' meno veloce: dobbiamo accertarci che nessuno degli stati che contengono reducing items abbia contemporaneamente delle mosse di shift; detto in termini semplici, i cerchietti col doppio cerchiettino non devono avere una freccettina uscente. Una volta che abbiamo attivato la nostra vista di falco possiamo vedere che gli stati 3, 5, 7, 8, 9, 10, 12, 13 hanno tutti il doppio cerchiettino (reducing items) ma non hanno nessuna freccia uscente, il che ci farebbe festeggiare, se non fosse per quel dannato stato 2, colpevole di avere e il doppio cerchiettino, e la freccia uscente. È finita, quindi? Abbiamo davvero un conflitto?

No, fortunatamente: la riduzione, infatti, si applica solo quando il prossimo simbolo in lettura è $, mentre invece lo shift viene applicato quando in lettura c'è =; non essendoci ulteriori possibilità di conflitto, possiamo felicemente asserire che la nostra grammatica è LR(1).

La grammatica è SLR(1)? Per rispondere a questa domanda è comunque conveniente costruirne il relativo automa caratteristico; per gli smemorati, ricordiamo che l'automa caratteristico SLR(1) si costruisce con stati di LR(0)-items e utilizzando come lookahead function l'insieme follow(P). Il risultato finale dovrebbe essere più o meno come Fig.9.5.

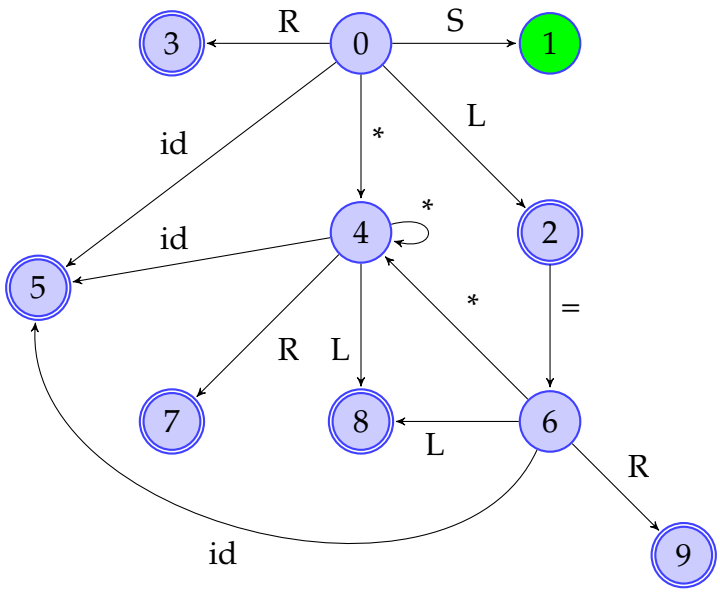


Figura 9.5: Automa caratteristico SLR(1) per Eq.9.6

E purtroppo vediamo subito il problema: dal momento che la nostra lookahead function è follow(B), lo stato 2 presenta sia una mossa di shift verso =, sia una mostra di reduce. Pertanto, nella tabella avremo un conflitto nella cella [2, =], e tanto basta per permetterci di dire che la grammatica Eq.9.6 non è SLR(1).

La grammatica dei puntatori Questo paragrafo al momento è vuoto perché a lezione il discorso è stato lasciato cadere magistralmente, ma forse poi ci torniamo sopra perché pare interessante.

$S \to L = R \mid R$
$L \to *R \mid id$
$R \to L$

in 2: reduce $R \to L$ per i follow di R

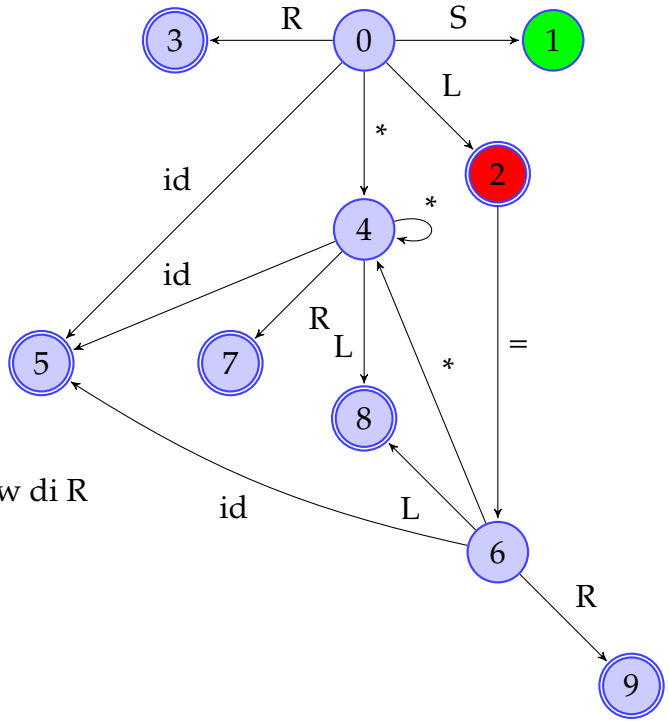


Figura 9.6: Automa caratteristico SLR(1) per Eq.9.6, con evidenziati conflitti e grammatica di partenza

Cerchiamo di capire come mai arrivare nello stato 2 ci porta a un conflitto, se utilizziamo un automa SLR(1). La prima cosa che osserviamo è che 2 è raggiungibile solo dallo stato 0 e solo quando in lettura troviamo L; poiché L è un non-terminale, non arriveremo mai a 2 con una mossa di shift; l'unico modo per raggiungere 2 è una mossa di goto da 0, e perché questo succeda vuol dire che devo aver prima operato una riduzione tale che sono ritornato in 0 e mi trovo con L nella pila dei simboli. Una riduzione che potrebbe asservire lo scopo è la riduzione dello stato 5; capiamolo meglio eseguendo il parsing della parola $w = "id = id"$.

Parsing di $w = id = id$ Quindi, penna alla mano, lanciamo l'algoritmo shift/reduce. Al lancio della procedura abbiamo queste strutture:

$$w = id{=}id\$$$
$$stSt = 0$$
$$symSt =$$

Da 0 leggo id ed eseguo la relativa transizione verso lo stato 5. Situazione:

$$\begin{aligned} w &= \underline{id}{=}id\$ \\ stSt &= 0\ 5 \\ symSt &= id \end{aligned}$$

In 5 troviamo la riduzione $L \to id$, per cui aggiorniamo le nostre strutture secondo le regole che ben conosciamo; ci dovremmo quindi ritrovare di nuovo nello stato 0 e, leggendo $=$ e compiendo la transizione a 2, ci troviamo finalmente nello stato del conflitto:

$$\begin{aligned} w &= \underline{id}{=}\ id\$ \\ stSt &= 0\ \not{5}\ 2 \\ symSt &= \not{id}\ L \end{aligned}$$

Come risolviamo quindi questo conflitto s/r? Cosa applico, la reduce $R \to L$, oppure uno shift verso lo stato 6? Distogliamo un attimo gli occhi dal ragionamento e guardiamo la grammatica da cui siamo partiti (convenientemente riportata in Fig.9.6): non ha nessun senso applicare la riduzione, perché durante la procedura non abbiamo mai eseguito produzioni $R \to L$, per cui in questo caso la mossa giusta è lo shift.

Parsing di $w = ***id{=}id$ Proviamo adesso con un'altra parola, questa volta con tre terminali $*$. L'inizio è il medesimo della parola precedente:

$$\begin{aligned} w &= ***id{=}id\$ \\ stSt &= 0 \\ symSt &= \end{aligned}$$

Le prime tre mosse sono piuttosto lisce: leggiamo tre $*$, per cui inizialmente percorreremo la $*$-transizione da 0 a 4, e successivamente percorreremo due volte il self-loop dello stato 4; infine, leggiamo id e operiamo la transizione verso lo stato 5. A questo punto le nostre strutture sono così:

$$\begin{aligned} w &= \underline{***id}{=}\ id\$ \\ stSt &= 0\ 4\ 4\ 4\ 5 \\ symSt &= ***id \end{aligned}$$

E niente, c'è la riduzione $L \to id$ da fare: rimuovo 5 da $stSt$ e sostituisco il body id con il driver L in $symSt$. Notiamo che, a differenza di prima, la

riduzione in questo caso ci fa tornare allo stato 4; da questo, quindi, leggiamo L e transizioniamo verso lo stato 8.

$$\begin{aligned} w &= \underline{***id} = id\$ \\ stSt &= 0\ 4\ 4\ 4\ 8 \\ symSt &= ***L \end{aligned}$$

Non è ancora finita, perché in 8 troviamo un'altra riduzione, in particolare $R \to L$. Aggiorniamo le strutture e torniamo indietro allo stato 4; leggendo quindi R ci ritroviamo nello stato 7, che ci porta a operare una nuova riduzione.

$$\begin{aligned} w &= \underline{***id} = id\$ \\ stSt &= 0\ 4\ 4\ 4\ 7 \\ symSt &= ***L \end{aligned}$$

La riduzione da operare è $L \to^* R$, per cui cancelliamo gli ultimi due stati da $stSt$ e modifichiamo contestualmente la testa di $symSt$, per ritrovarci infine in questa situazione:

$$\begin{aligned} w &= \underline{***id} = id\$ \\ stSt &= 0\ 4\ 4\ 8 \\ symSt &= **L \end{aligned}$$

Siamo di nuovo nello stato 4 con L in lettura: la medesima situazione in cui ci eravamo ritrovati poco sopra! Questo vuol dire che andremo a ripetere di nuovo le stesse operazioni: transizione verso 8, riduzione $R \to L$, transizione verso 7, riduzione $L \to^* R$; questo ciclo andrà iterato finché tutti gli $*$ non saranno spariti, e descrive molto bene quello che succede a livello macchina quando andiamo a risolvere un puntatore. Infatti, quando avremo risolto tutti gli $*$, saremo di nuovo nello stato 0 e avremo un goto L, punto di boa a cui eravamo arrivati anche nella precedente parola analizzata. Adesso completiamo il ragionamento e andiamo fino in fondo.

Itaque seguiamo il goto L e dirigiamoci verso le acque burrascose del tormentato stato 2.

$$\begin{aligned} w &= \underline{id} = id\$ \\ stSt &= 0\ 2 \\ symSt &= L \end{aligned}$$

Che fare? Non possiamo più affidarci a ragionamenti pigri come quello che abbiamo fatto prima, ma magari ci va bene uguale; scegliamo di fare uno shift

e andiamo avanti a testa alta e petto in fuori, diretti verso la gloria o verso il baratro. Arriviamo quindi a 6.

$$\begin{aligned} w &= \underline{id{=}}id\$ \\ stSt &= 0\ 2\ 6 \\ symSt &= L{=} \end{aligned}$$

In lettura abbiamo id, per cui facciamo unp shift verso 5; bruciamo la tappa ed eseguiamo subito la riduzione $L \to id$ che ci fa rimbalzare indietro dallo stato 6 e infine allo stato 8.

$$\begin{aligned} w &= \underline{id{=}\ id}\$ \\ stSt &= 0\ 2\ 6\ \cancel{5}\ 8 \\ symSt &= L{=}\cancel{id}L \end{aligned}$$

Di nuovo una riduzione: questa $R \to L$ ci fa saltapicchiare nuovamente allo stato 6 e questa volta finiamo nello stato 9. Situazione pile:

$$\begin{aligned} w &= \underline{id{=}\ id}\$ \\ stSt &= 0\ 2\ 6\ \cancel{8}\ 9 \\ symSt &= L{=}\cancel{L}R \end{aligned}$$

Ci siamo: in 9 abbiamo la riduzione $S \to L = R$, per cui torniamo indietro allo stato 0, leggiamo S e corriamo veloci ad abbracciare l'accepting item nello stato 1.

Nonostante la scelta arbitraria, ci è decisamente andata bene; ma se invece dello shift avessimo scelto la riduzione? Torniamo a quel bivio e proviamo a percorrere anche quest'altra strada. Operando la riduzione $R \to L$ saremmo passati allo stato 3:

$$\begin{aligned} w &= \underline{id}{=}\ id\$ \\ stSt &= 0\ 3 \\ symSt &= R \end{aligned}$$

Da qui leggiamo $=$, ma non abbiamo assolutamente nessuna mossa da fare: cadiamo nel baratro delle caselle `error`, e il nostro parsing si ferma qui. La scelta giusta, di nuovo, era lo shift.

9.2.2 La differenza pratica e concettuale tra SLR(1) e LR(1)

E così abbiamo concluso che Eq.9.6 non è una SLR(1), ma LR(1) invece sì. Vi va di indagare un attimino di più su quali siano le cause di ciò?

Lo stato che genera un conflitto all'interno del automa caratteristico SLR(1) è quello identificato dal numero 2: poiché in quest'ultimo gli item di riduzione vengono trasformati in riduzioni effettive in presenza dei follow del driver delle produzioni non riusciamo ad avere un'espressività tale da discriminare le operazioni di shift e riduzione, per cui si crea un conflitto s/r. Utilizzando un parsing LR(1) nel medesimo stato avremo invece un'operazione di `shift 6` solo nel caso in cui leggessimo {=} dall'input buffer mentre invece `reduce` $R \to L$ nel caso in cui il simbolo letto in input sia {$}.

Supponiamo a questo proposito di fare un'analisi di una stringa del tipo $w_1 = w_2$ utilizzando l'automa caratteristico LR(1): visto che = è un simbolo generato solo se si sceglie una produzione del tipo $S \to L = R$, una volta che abbiamo letto tutto ciò che fa a capo w_1 dobbiamo accertarci di eseguire una riduzione ad L. A questo punto, però, osservando la grammatica dovrebbe essere abbastanza chiaro che le uniche parole derivabili da L appartengono a $\{*^n id \mid n \geq 0\}$ e che le uniche derivazioni di parole in $\{*^n id \mid n > 0\}$ coinvolgono la R (i.e. se vogliamo una stringa in cui compaia almeno una volta il simbolo $*$ è necessario utilizzare la produzione $L \to *R$).

Se la stringa che si sta analizzando è del tipo $id = w_2$, allora ci spostiamo dallo stato 5 per poi effettuare riduzione $L \to id$ e spostarci allo stato 2. Se invece ci stiamo concentrando su una stringa del tipo $*w_1' = w_2$, allora ci sposteremo invece allo stato 4, dove continueremo a eseguire un ciclo fino a che non esauriremo gli $*$; nel momento in cui leggeremo invece id ci sposteremo allo stato 5, eseguiremo riduzione $L \to id$ e finiremo con lo spostarci allo stato 8 e **non** al 2.

Il parsing di tipo SLR(1) non riconosce dunque la differenza fra lo stato 2 e lo stato 8 a differenza del parsing LR(1), che ha un costo maggiore. Quindi, il problema del parsing SLR(1) (in questo caso) è che andiamo a piazzare un'operazione di riduzione per tutti i follow dei driver mentre in quello di tipo LR(1) lo si fa per sottoinsiemi dei follow dei driver.

9.3 Il parsing LALR(1)

Nonostante il parsing di tipo LR(1) sia il più completo è possibile che l'automa caratteristico risultante contenga delle ridondanze: nell'esempio precedente vi sono infatti delle sottostrutture che sono isomorfe e ciò deriva dal fatto che, visto che le transizioni dipendono sempre dalla prima componente (i.e. item LR(0)), è

possibile notare che le ridondanze si verificano nel caso in cui gli stati abbiano la stessa proiezione LR(0).

Per mantenere la precisione el'affidabilità del parsing LR(1) ma senza rinunciare all'efficienza di quello SLR(1) possiamo utilizzare una terza via, il parsing LALR(1): ques'ultimo utilizza gli stati in maniera simile al parsing SLR(1), ma utilizza una funzione di lookahead un filo più raffinata. Andiamo a conoscerla più da vicino

9.3.1 L'Automa caratteristico LRm(1)

L'automa LRm(1) $\mathcal{AM}$ è costruito a partire dall'automa LR(1) $\mathcal{A}$ (la lettera m sta per *merged*).

Gli **Stati** del nuovo automa caratteristico sono ottenuti unendo all'interno di un singolo stato di $\mathcal{AM}$ tutti gli item negli stati $< P_1, ..., P_n >$ di $\mathcal{A}$ che hanno le stesse LR(0)-proiezioni (i.e. con lo stesso item LR(0)).

Transizioni: Se lo stato P di $\mathcal{A}$ ha la stessa Y-transizione a Q e se P è stato unito in $< P_1, ..., P_n >$ e Q in $< Q_1, ..., Q_m >$, allora c'è una Y-transizione in $\mathcal{AM}$ da $< P_1, ..., P_n >$ a $< Q_1, ..., Q_m >$.

Se prendiamo in considerazione l'automa LR(1) definito precedente, possiamo osservare che gli stati 4 e 11 hanno la medesima prima proiezione e quindi possiamo unirli all'interno di un nuovo stato.

Cosa facciamo per le transizioni? Nell'automa LR(1) abbiamo un'unica transizione etichettata id che va dallo stato 4 allo stato 5 (e allo stesso modo un'altra etichettata id dallo stato 11 al 12). Visto che gli stati 4 e 11 sono stati uniti in un unico stato (4&11) e lo stesso vale per 5 e 12 (5&12), per la definizione precedente possiamo inserire in $\mathcal{AM}$ una transizione da 4&11 a 5&12 etichettata id.

Eseguendo l'operazione descritta la dimensione (i.e. il numero di stati) dell'automa LRm(1) così generato è sicuramente uguale a quella dell'automa LR(0): andando a combinare stati con con gli stessi item LR(0) ci ritroveremo con lo stesso numero di stati dell'automa LR(0) e con lo stesse transizioni.

9.3.2 Costruzione di una tabella di parsing LALR(1)

Le parsing table LALR(1) sono ottenute prendendo:

- l'automa caratteristico LRm(1)
- la lookahead function $\mathcal{LA}(P, A \to \beta) = \cup_{[A \to \beta \cdot, \Delta_j]} \Delta_j$

Nel caso in cui non vi sia più di uno stato con la stessa proiezione LR(0) allora tale stato non subisce l'operazione di unione e viene inserito direttamente nell'automa LRm(1) con il medesimo lookahead-set.

La grammatica $\mathcal{G}$ è LALR(1) se e solo se la sua tabella di parsing LALR(1) non ha conflitti.

Il vantaggio della grammatica LALR(1) è di essere più potente (i.e. espressiva) della grammatica SLR(1) pur rimanendo con delle dimensioni contenute rispetto alla LR(1).

Esercizio parsing LALR(1) - c'è sì ma in realtà no perchè è LR(1)

$$
\begin{aligned}
&S \to AaB \mid b \\
&A \to BcBaA \mid \varepsilon \\
&B \to \varepsilon
\end{aligned}
$$

Il nostro obbiettivo è quello di costruire la parsing table LALR(1) per la grammatica citata: per poter procedere dobbiamo però prima costruire l'automa caratteristico.

Ricaviamo l'automa

1. Inizializziamo lo stato 0; il suo kernel è

   $$S' \to \cdot S, \{\$\}$$

   Di questo kernel devo calcolare la chiusura:

   $$
   \begin{aligned}
   &S \to \cdot AaB, \{\$\} \\
   &S \to \cdot b, \{\$\} \\
   &A \to \cdot BcBaA, \{a\} \\
   &A \to \cdot, \{a\} \\
   &B \to \cdot, \{c\}
   \end{aligned}
   $$

   Essendo questi gli item per lo stato 0, possiamo identificare quattro possibili transizioni (e quindi quattro possibili nuovi stati): $\tau(0, S) = 1$, $\tau(0, A) = 2$, $\tau(0, b) = 3$ e $\tau(0, B) = 4$.

   Interessante notare che le produzioni del tipo $A \to \varepsilon$ vengono convertite in reducing item $A \to \cdot$.

2. Analizziamo ora lo stato 1; il suo kernel è

$$S' \to S\cdot, \{\$\}$$

Non serve calcolare la chiusura di questo item in quanto è formata solamente da sé stesso, possiamo però evidenziare il fatto che questo è lo stato contenente l'**Accepting Item**.

3. Analizziamo dunque lo stato 2, il cui kernel è composto solamente da:

$$S \to A \cdot aB, \{\$\}$$

Nemmeno in questo caso è necessario calcolare la sua chiusura in quanto il marker si trova davanti ad un non terminale: aggiungiamo dunque la transizione $\tau(2, a) = 5$ e proseguiamo.

4. Il kernel dello stato 3 è dato da

$$S \to b\cdot, \{\$\}$$

Il che vuol dire che non è possibile calcolare la chiusura e che lo stato 3 contiene un reducing item.

5. Passiamo allo stato 4, il cui kernel è:

$$A \to B \cdot cBaA, \{a\}$$

Per gli stessi motivi dello stato 2 la chiusura di tale stato non porta nuovi elementi; è comunque possibile definire la transizione $\tau(4, c)$ allo stato 6

6. Il kernel dello stato 5 è:

$$S \to Aa \cdot B, \{\$\}$$

La cui chiusura risulta essere pari a

$$B \to \cdot, \{\$\}$$

Per questo motivo sappiamo che lo stato contiene un reducing item e possiede una transizione $\tau(5, B)$ allo stato 7

7. Procedendo come abbiamo fatto fino ad ora il kernel per lo stato 6 è

$$A \to Bc \cdot BaA, \{a\}$$

la cui chiusura corrisponde a

$$B \to \cdot, \{a\}$$

Come è intuibile lo stato 6 contiene un reducing item e la sua transizione è $\tau(6, B) = 8$

8. Il kernel dello stato 7 è
$$S \to AaB\cdot, \{\$\}$$
Essendo che il marker è in fondo alla produzione non è possibile effettuare la chiusura ma solo considerare che lo stato 7 contiene un reducing item

9. Lo stato 8 ha kernel
$$A \to BcB \cdot aA, \{a\}$$
e possiede una transizione $\tau(8, a)$ allo stato 9

10. Lo stato 9 ha kernel
$$A \to BcBa \cdot A, \{a\}$$
di cui possiamo calcolare la chiusura ottenendo
$$A \to \cdot BcBaA, \{a\}$$
$$A \to \cdot, \{a\}$$
$$B \to \cdot, \{c\}$$
Che contiene due reducing item e possiede due transizioni: la prima è $\tau(9, A) = 10$ e ha come target un nuovo stato mentre la seconda è $\tau(9, B) = 4$ che ha come target uno stato che fa già parte del nostro automa caratteristico.

11. Lo stato 10 infine ha kernel
$$A \to BcBaA\cdot, \{a\}$$
Visto che il marker è posto alla fine non è possibile calcolare la chiusura di questo stato e possiamo concludere aggiungendo che lo stato 10 ha un reducing item.

L'automa caratteristico LR(1) risulta dunque così costruito

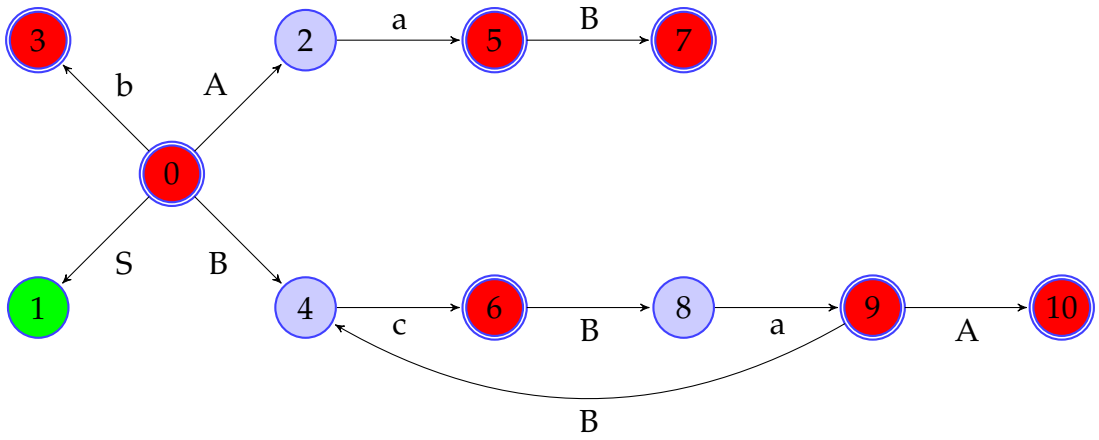


Figura 9.7: Automa LR(1)/LALR(1)

Visto che non è possibile unire nessuno degli stati dell'automa LR(1), l'automa disegnato poc'anzi corrisponderà anche a quello LRm(1). Di seguito dunque possiamo costruire la tabella di parsing

1. $S \to AaB$
2. $S \to b$
3. $A \to BcBaA$
4. $A \to \varepsilon$
5. $B \to \varepsilon$

	a	b	c	$	S	A	B
0	r4	s3	r5		1	2	4
1				Acc			
2	s5						
3				r2			
4			s6				
5				r5			7
6	r5						8
7				r1			
8	s9						
9	r4		r5			10	4
10	r3						

Tabella 9.1: LR(1) & LALR(1) Parsing Table

9.3.3 L'automa simbolico

La classe di grammatiche LALR contiene le grammatiche più interessanti per i linguaggi di programmazione. Abbiamo visto come la modalità più semplice per ottenere le tabelle di parsing LALR è quella di creare in primis un'automa LR(1) e poi tradurlo in LRm(1).

Oggi vediamo un altro metodo che prevede di eliminare il passaggio dall'automa LR(1) e di creare fin da subito un automa con la stessa struttura dell'automa

LRm(1). Quello che andremo a creare è detto automa simbolico, preché utilizza dei simboli (delle variabili) al posto dei lookahead set; una votla terminata la costruzione dell'automa simbolico risolvendo i simboli si va a calcolare quelli che poi saranno i lookahead set dell'automa LRm(1).

Quindi gli item di questo automa simbolico possono essere immaginati cme suddivisi in due componenti:

- una componente di tipo LR(0);
- una parte composta da un insieme di lookahead simbolico.

Sostanzialmente questi item sono del tutto assimilabili agli item LR(1), con l'unica differenza che il Δ set è simbolico e presto al lettore sarà ben chiaro questo concetto.

Mentre creaimo l'automa simbolico ci memorizziamo in una tabella delle equazioni che ci serviranno alla fine per tradurre i lookahead set simbolici in lookahead effettivi. Come è ormai consuetudine il modo più semplice per spiegare questo procedimento è applicarlo direttamente ad un esempio.

Esempio di costruzione dell'automa simbolico

Andiamo a costruire l'automa simbolico per la nostra cara grammatica

$$
\begin{aligned}
&S \to L = R \mid R \\
&L \to *R \mid id \\
&R \to L
\end{aligned}
$$

La costruzione dell'automa simbolico verrà affiancata dalla costruzione dell'automa LR(1) per rendere più chiare le differenze tra queste due costruzioni.

Il primo passo è installare lo stato 0 dell'automa (Figura 9.8):

0:

$S' \to \cdot S, \{x_0\}$
$S \to \cdot L = R, \{x_0\}$
$S \to \cdot R, \{x_0\}$
$L \to \cdot * R, \{= x_0\}$
$L \to \cdot id, \{= x_0\}$
$R \to \cdot L, \{x_0\}$

invece di

0:

$S' \to \cdot S, \{\$\}$
$S \to \cdot L = R, \{\$\}$
$S \to \cdot R, \{\$\}$
$L \to \cdot * R, \{= \$\}$
$L \to \cdot id, \{= \$\}$
$R \to \cdot L, \{\$\}$

Figura 9.8: Stato 0: sulla sinistra automa simbolico, sulla destra automa LR(1)

È subito chiaro cosa intendevamo dicendo "invece di inserire il lookahead set inseriamo una *variabile*", di fatto come lookahead inseriamo un simbolo, che risolveremo solo alla fine della costruzione dell'automa.

In questo caso al posto di inserire \$ inseriamo una variabile chiamata x_0. La corrispondenza tra x_0 e \$ la salviamo (ce la scriviamo) nel sitema di equazioni che ci porteremo dietro per tutta la costruzione.

$$Sistema:$$
$$x_0 = \{\$\}$$

Quando andiamo a fare la chiusura, essendo gli item simbolici in tutto e per tutto simili agli item LR(1), utilizziamo la $closure_1$. Questa volta però abbiamo come lookahead set le variabili, in questo caso il nostro lookahead set è x_0.

Procedendo un passettino alla volta la chiusura dello stato 0 si fa così:

- in primis dobbiamo includere la chiusura per le produzioni di S:
  - dalle due prduzioni di S della nostra grammatica otteniamo i due item:

    $$S \to \cdot L = R,\ \{x_0\}$$
    $$S \to \cdot R,\ \{x_0\}$$

    in questo caso il lookahead set è facile da calcolare perché $\beta = \varepsilon$ mentre $\Delta = x_0$;
- ora abbiamo introdotto anche un marker davanti alla L, quindi dobbiamo chiudere anche per L:
  - dalle produzioni di L otteniamo i due item:

    $$L \to \cdot * R,\ \{=\}$$
    $$L \to \cdot id,\ \{=\}$$

    in questo caso, dato che $\beta \neq \varepsilon$, vediamo come il lookahead set corrisponda ad un carattere e non una variabile, questo è possibile mentre stiamo calcolando la chiusura di item simbolici;
- infine, avendo introdotto anche R, dobbiamo calcolare la chiusura indotta dalle produzioni di R:
  - abbiamo solo una produzione per R, che ci porta ad inserire l'item

    $$R \to \cdot L,\ \{x_0\}$$

    è andato tutto liscio come l'olio no?

- invece no! perché abbiamo introdotto una nuova chiusura per L, che è diversa dalla precedente perché ora il lookahead set della chiusura è x_0 invece che $=$, quindi quello che dobbiamo fare è ricalcolare le chiusure di L con questo nuovo lookahead; dato che siamo skillati e sappiamo già come andrà a finire aggiungiamo semplicemente $\{x_0\}$ al lookahead set delle produzioni di L che abbiamo già elencato poco fa, quindi:

$$L \to \cdot * R,\ \{=, x_0\}$$
$$L \to \cdot id,\ \{=, x_0\}$$

Il consiglio per chi non ha capito questi passaggi è quello di fare qualche esercizio di costruzione di automi LR(1), fidatevi ne vale la pena!

Orbene, passiamo ad analizzare le transazioni uscenti dallo stato 0. La prima transizione che troviamo è $\tau(0, S)$ che ci porta ad un nuovo stato, lo stato 1.
Kernel dello stato 1 (Figura 9.9):

1:
$S' \to S\cdot, \{x_1\}$

invece di

1:
$S' \to S\cdot, \{\$\}$

Figura 9.9: Kernel stato 1: sulla sinistra automa simbolico, sulla destra automa LR(1)

notiamo come invece che scrivere il lookahead set come insieme creiamo una nuova variabile per indicarlo. Dobbiamo aggiornare anche il nostro sitema in cui salviamo il valore effettivo della variabile. In questo caso il lookahed set del kernel, essendo $\beta = \varepsilon$ è proprio uguale al lookahead set della produzione da cui arriviamo, quindi x_0, aggiorniamo dunque alacremente il nostro sistema di equazioni.

$$Sistema:$$
$$x_0 = \{\$\}$$
$$x_1 = x_0$$

Lo stato 1 non ha chiusure da calcolare e nemmeno transizioni, quindi passiamo oltre.

La prossima transizione che andiamo ad analizzare è $\tau(0, L)$ che ci porta in un nuovo stato, diciamo 2.
Kernel dello stato 2 (Figura 9.10)

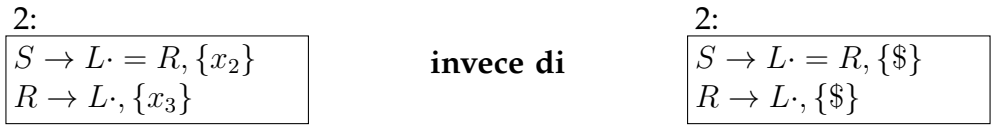


Figura 9.10: Kernel stato 2: sulla sinistra automa simbolico, sulla destra automa LR(1)

in questo caso il kernel contiene due item perché nello stato 0 abbiamo due item che presentano la possibilità di una L-transizione; ognuno di questi item quindi avrà il suo lookahead set simbolico: andiamo ad aggiornare subito il nostro sistema, sapendo che le produzioni che ci portano allo stato due hanno come lookahead set x_0.

$$
\begin{aligned}
&Sistema: \\
&\qquad x_0 = \{\$\} \\
&\qquad x_1 = x_0 \\
&\qquad x_2 = x_0 \\
&\qquad x_3 = x_0
\end{aligned}
$$

grazie al cielo non si presentano nè chiusure, le transizioni invece le analizzeremo in seguito.

Passiamo quindi all'osservazione della transizione $\tau(0, R)$, che ci porta nello stato 3.
Kernel dello stato 3 (Figura 9.11):

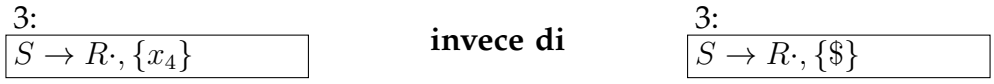


Figura 9.11: Kernel stato 3: sulla sinistra automa simbolico, sulla destra automa LR(1)

anche in questo caso abbiamo che $\beta = \varepsilon$ quindi il lookahead set del kernel è

uguale a Δ, riportiamo subito questa informazione all'interno del sistema.

$$
\begin{aligned}
&Sistema: \\
&\qquad x_0 = \{\$\} \\
&\qquad x_1 = x_0 \\
&\qquad x_2 = x_0 \\
&\qquad x_3 = x_0 \\
&\qquad x_4 = x_0
\end{aligned}
$$

il kernel dello stato 3 non presenta nè chiusure nè transizioni, quindi procediamo oltre senza esitazioni.

La prossima transizione che dobbiamo analizzare è $\tau(0, *)$, questa deriva dalla produzione $[L \to \cdot * R,\ \{=, x_0\}]$.
Kernel dello stato 4 (Figura 9.12):

4:

$L \to * \cdot R, \{x_5\}$
$R \to \cdot L, \{x_5\}$
$L \to \cdot * R, \{x_5\}$
$L \to \cdot id, \{x_5\}$

invece di

4:

$L \to * \cdot R, \{=, \$\}$
$R \to \cdot L, \{=, \$\}$
$L \to \cdot * R, \{=, \$\}$
$L \to \cdot id, \{=, \$\}$

Figura 9.12: Kernel stato 4: sulla sinistra automa simbolico, sulla destra automa LR(1)

anche in questo caso $\beta = \varepsilon$ quindi il lookahead set del kernel è uguale a Δ, aggiorno quindi il sistema di equazioni.

$$
\begin{aligned}
&Sistema: \\
&\qquad x_0 = \{\$\} \\
&\qquad x_1 = x_0 \\
&\qquad x_2 = x_0 \\
&\qquad x_3 = x_0 \\
&\qquad x_4 = x_0 \\
&\qquad x_5 = \{=, \$\}
\end{aligned}
$$

finalmente uno stato che ci dia la *soddisfazione* di calcolare delle chiusure, grazie alle produzioni di R in primis e di L poi otteniamo la seguente chiusura per gli

item dello stato 4:

$$R \to \cdot L,\ \{x_5\}$$
$$L \to \cdot * R,\ \{x_5\}$$
$$L \to \cdot id,\ \{x_5\}$$

che spettacolo, eh? passiamo oltre

Prossima transizione da analizzare: $\tau(0, id)$ che deriva dall'item $[L \to \cdot id,\ \{=, x_0\}]$ e ci porta nello stato 5.
Kernel dello stato 5 (Figura 9.13):

5:		5:
$L \to id\cdot, \{x_6\}$	**invece di**	$L \to id\cdot, \{=, \$\}$

Figura 9.13: Kernel stato 5: sulla sinistra automa simbolico, sulla destra automa LR(1)

anche in questo caso $\beta = \varepsilon$ quindi il lookahead set del kernel è uguale a Δ, aggiorno quindi il sistema di equazioni.

$$Sistema:$$
$$x_0 = \{\$\}$$
$$x_1 = x_0$$
$$x_2 = x_0$$
$$x_3 = x_0$$
$$x_4 = x_0$$
$$x_5 = \{=, \$\}$$
$$x_6 = \{=, \$\}$$

Anche qui, pace all'anima nostra, inseriamo il lookahead set come variabile e la valorizziamo nel nostro sistema di equazioni.

Abbiamo finito le transizioni dallo stato 0! prossima trnsizione da analizzare $\tau(2, =)$, che deriva dall'item $[S \to L\cdot = R,\ \{x_2\}]$, ci porta direttamente allo stato 6.
Kernel dello stato 6 (Figura 9.14):

6:

$S \to L = \cdot R, \{x_7\}$
$R \to \cdot L, \{x_7\}$
$L \to \cdot * R, \{x_7\}$
$L \to \cdot id, \{x_7\}$

invece di

6:

$S \to L = \cdot R, \{\$\}$
$R \to \cdot L, \{\$\}$
$L \to \cdot * R, \{\$\}$
$L \to \cdot id, \{\$\}$

Figura 9.14: Kernel stato 6: sulla sinistra automa simbolico, sulla destra automa LR(1)

in questo caso $\beta = \varepsilon$ quindi il lookahead set del kernel è uguale a Δ, aggiorno quindi il sistema di equazioni.

$$
\begin{aligned}
&Sistema: \\
&\quad x_0 = \{\$\} \\
&\quad x_1 = x_0 \\
&\quad x_2 = x_0 \\
&\quad x_3 = x_0 \\
&\quad x_4 = x_0 \\
&\quad x_5 = \{=, \$\} \\
&\quad x_6 = \{=, \$\} \\
&\quad x_7 = x_2
\end{aligned}
$$

Ora passiamo ad analizzare le chiusure degli item di questo stato. Grazie alla presenza del marker davanti ad una R aggiungiamo questi item allo stato:

$$
\begin{aligned}
&R \to \cdot L, \ \{x_7\} \\
&L \to \cdot * R, \ \{x_7\} \\
&L \to \cdot id, \ \{x_7\}
\end{aligned}
$$

Passiamo ora ad analizzare la transizione $\tau(4, R)$, che deriva dall'item $[L \to * \cdot R, \ \{x_5\}]$ e ci spara nello stato 7.
Kernel dello stato 7 (Figura 9.15):

7:

$L \to *R\cdot, \{x_8\}$

invece di

7:

$L \to *R\cdot, \{=, \$\}$

Figura 9.15: Kernel stato 7: sulla sinistra automa simbolico, sulla destra automa LR(1)

in questo caso $\beta = \varepsilon$ quindi il lookahead set del kernel è uguale a Δ, aggiorno quindi il sistema di equazioni.

$$
\begin{aligned}
&Sistema: \\
&\qquad x_0 = \{\$\} \\
&\qquad x_1 = x_0 \\
&\qquad x_2 = x_0 \\
&\qquad x_3 = x_0 \\
&\qquad x_4 = x_0 \\
&\qquad x_5 = \{=, \$\} \\
&\qquad x_6 = \{=, \$\} \\
&\qquad x_7 = x_2 \\
&\qquad x_8 = x_5
\end{aligned}
$$

a questo punto dovremmo fare la chiusura no? ma è già tutto chiuso, come alle 23:30 di sera a Trento...

Passiamo quindi ad analizzare la transizione $\tau(4, L)$ che prende le mosse dall'item $[R \to \cdot L,\ \{x_5\}]$ e ci porta nello stato 8.
Kernel dello stato 8 (Figura 9.16):

8:
$R \to L\cdot, \{x_9\}$

invece di

8:
$R \to L\cdot, \{=, \$\}$

Figura 9.16: Kernel stato 8: sulla sinistra automa simbolico, sulla destra automa LR(1)

in questo caso il lookahead set del kernel è uguale a Δ, aggiorno quindi il

sistema di equazioni.

$$
\begin{aligned}
&Sistema: \\
&\qquad x_0 = \{\$\} \\
&\qquad x_1 = x_0 \\
&\qquad x_2 = x_0 \\
&\qquad x_3 = x_0 \\
&\qquad x_4 = x_0 \\
&\qquad x_5 = \{=, \$\} \\
&\qquad x_6 = \{=, \$\} \\
&\qquad x_7 = x_2 \\
&\qquad x_8 = x_5 \\
&\qquad x_9 = x_5
\end{aligned}
$$

anche qui niente chiusure.

Passiamo ad analizzare la transizione $\tau(4, *)$, che deriva dall'item $[L \rightarrow \cdot * R, \{x_5\}]$; ora, se osserviamo bene il kernel dello stato in cui arriveremmo tramite questa transizione ci rediamo conto che è un kernel che abbiamo già incontrato, nello specifico proprio nello stato 4 da cui stiamo partendo, cosa significa tutto ciò?

Significa semplicemente che c'è un arco uscente da 4 che ci riporta in 4 (self loop in gergo) tramite una $*$-transizione. È qui che vediamo finalmente la grande differenza di costruzione tra automi LR(1) ed automi simbolici: in un automa LR(1) avremmo costruito un nuovo stato (perché il lookahead set in questo caso è diverso dal lookahead set dello stato 4), invece ora andiamo semplicemente ad aggiornare il lookahead set dello stato 4 con il lookahead set della transizione

$\tau(4, *)$ (che fatalità corrisponde propiro a x_5).

$$
\begin{aligned}
&Sistema: \\
&\qquad x_0 = \{\$\} \\
&\qquad x_1 = x_0 \\
&\qquad x_2 = x_0 \\
&\qquad x_3 = x_0 \\
&\qquad x_4 = x_0 \\
&\qquad x_5 = \{=, \$\} \cup \{x_5\} \\
&\qquad x_6 = \{=, \$\} \\
&\qquad x_7 = x_2 \\
&\qquad x_8 = x_5 \\
&\qquad x_9 = x_5
\end{aligned}
$$

Passiamo ora ad analizzare la transizione $\tau(4, id)$, che deriva dall'item $[L \to \cdot id,\ \{x_5\}]$, colpo di scena! anche questo kernel ci è noto, corrisponde a quello dello stato 5, quindi andiamo ad aggiornare il lookahead set dello stato 5. Il lookahead set della transizione $\tau(4, id)$ è ancora una volta x_5.

$$
\begin{aligned}
&Sistema: \\
&\qquad x_0 = \{\$\} \\
&\qquad x_1 = x_0 \\
&\qquad x_2 = x_0 \\
&\qquad x_3 = x_0 \\
&\qquad x_4 = x_0 \\
&\qquad x_5 = \{=, \$\} \cup \{x_5\} \\
&\qquad x_6 = \{=, \$\} \cup \{x_5\} \\
&\qquad x_7 = x_2 \\
&\qquad x_8 = x_5 \\
&\qquad x_9 = x_5
\end{aligned}
$$

Passiamo oltre, la prossima transizione su cui ci concentriamo è $\tau(6, R)$, derivante da $[S \to L = \cdot R,\ \{x_7\}]$, che ci porta a scoprire lo stato 9.
Kernel dello stato 9 (Figura 9.17):

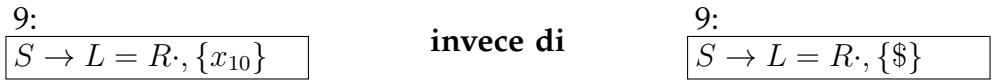


Figura 9.17: Kernel stato 9: sulla sinistra automa simbolico, sulla destra automa LR(1)

in questo caso $\beta = \varepsilon$ quindi il lookahead set del kernel è uguale a Δ, ovvero $x_{10} = \{x_7\}$, aggiorno quindi il sistema di equazioni.

$$
\begin{aligned}
&Sistema: \\
&\qquad x_0 = \{\$\} \\
&\qquad x_1 = x_0 \\
&\qquad x_2 = x_0 \\
&\qquad x_3 = x_0 \\
&\qquad x_4 = x_0 \\
&\qquad x_5 = \{=, \$\} \cup \{x_5\} \\
&\qquad x_6 = \{=, \$\} \cup \{x_5\} \\
&\qquad x_7 = x_2 \\
&\qquad x_8 = x_5 \\
&\qquad x_9 = x_5 \\
&\qquad x_{10} = x_7
\end{aligned}
$$

anche questo kernel ci regala poche emozioni (e nessuna chiusura) quindi concludiamo l'analisi dello stato 9.

La prossima transizione è $\tau(6, L)$ che deriva da $[R \to \cdot L, \{x_7\}]$; tale transizione ci porterebbe in uno stato con kernel LR(0) = $[R \to L\cdot]$, che quindi corrisponde al nostro stato 8: invece che creare un nuovo stato andiamo ad aggiornare il lookahead set dello stato 8 con il lookahead set che deriva dalla transizione

$\tau(6, L)$.

$$
\begin{aligned}
&Sistema: \\
&\qquad x_0 = \{\$\} \\
&\qquad x_1 = x_0 \\
&\qquad x_2 = x_0 \\
&\qquad x_3 = x_0 \\
&\qquad x_4 = x_0 \\
&\qquad x_5 = \{=, \$\} \cup \{x_5\} \\
&\qquad x_6 = \{=, \$\} \cup \{x_5\} \\
&\qquad x_7 = x_2 \\
&\qquad x_8 = x_5 \\
&\qquad x_9 = x_5 \cup \{x_7\} \\
&\qquad x_{10} = x_7
\end{aligned}
$$

niente chiusure, passiamo oltre.

Prossima transizione $\tau(6, *)$, derivante dall'item $[L \to \cdot * R, \{x_7\}]$, il target di questa transizione è uno stato con kernel LR(0) $[L \to * \cdot R]$, ovvero lo stato 4. Quindi anche questa volta non creiamo un nuovo stato ma andiamo ad aggionrare il kernel dello stato 4.

$$
\begin{aligned}
&Sistema: \\
&\qquad x_0 = \{\$\} \\
&\qquad x_1 = x_0 \\
&\qquad x_2 = x_0 \\
&\qquad x_3 = x_0 \\
&\qquad x_4 = x_0 \\
&\qquad x_5 = \{=, \$\} \cup \{x_5\} \cup \{x_7\} \\
&\qquad x_6 = \{=, \$\} \cup \{x_5\} \\
&\qquad x_7 = x_2 \\
&\qquad x_8 = x_5 \\
&\qquad x_9 = x_5 \cup \{x_7\} \\
&\qquad x_{10} = x_7
\end{aligned}
$$

Passiamo alla prossima transizione che è $\tau(6, id)$, derivante da $[L \to \cdot id, \{x_7\}]$, che ci porta in uno stato con kernel LR(0) = $[L \to id\cdot]$, che corrisponde al già

visitato stato 5, quindi aggiorniamo semplicemente il lookahead set di tale stato.

$$
\begin{aligned}
&Sistema: \\
&\qquad x_0 = \{\$\} \\
&\qquad x_1 = x_0 \\
&\qquad x_2 = x_0 \\
&\qquad x_3 = x_0 \\
&\qquad x_4 = x_0 \\
&\qquad x_5 = \{=, \$\} \cup \{x_5\} \cup \{x_7\} \\
&\qquad x_6 = \{=, \$\} \cup \{x_5\} \cup \{x_7\} \\
&\qquad x_7 = x_2 \\
&\qquad x_8 = x_5 \\
&\qquad x_9 = x_5 \cup \{x_7\} \\
&\qquad x_{10} = x_7
\end{aligned}
$$

Oh butei, abbiamo finito di analizzare l'automa, quindi andiamo a risolvere lo strascico di equazioni che ci siamo tirati dietro per tutto l'esercizio:

$$
\begin{aligned}
x_0, x_1, x_2, x_3, x_4, x_7, x_{10} &= \{\$\} \\
x_5, x_6, x_8, x_9 &= \{=, \$\}
\end{aligned}
$$

Questa risoluzione è fatta a occhio, ma esiste un metodo algoritmico per risolvere il sistema in modo molto efficiente. L'algoritmo in questione utilizzale classi di equivalenza, inaspettatamente.

Complessivamente cosa abbiamo ottenuto? Un automa con le stesse dimensioni di tipo LR(0), poichè abbiamo creato esattamente gli stessi stati che avremmo cerato costruendo un automa LR(0), ma abbiamo utilizzato delle chiusure di tipo LR(1), ricavando quindi dei lookahead set raffinati.

Per concludere questo esempio ricordiamo brevemente quali sarebbero i passi da seguire a questo punto per ottenere l'automa merged:

1. inseriamo tutti gli stati;
2. le mosse di shift (le transizioni) sono sesattamente quelle identificate e corrispondono alle transizioni dell'automa caratteristico LR(0) per questa grammatica;
3. gli stati con riduzioni sono tutti quelli che presentatno una produzione con il marker in fondo al body;

4. le lookahead function (le etichette per le riduizioni) corrispondono ai lookahead set calcolati nella risoluzione del sistema.

Carino questo esempio, eh? per festeggiare il suo completamento ci concediamo la risoluzione di un altro esercizio dello stesso tipo.

9.3.4 Esercizio di costruzione dell'automa simbolico

La grammatica di cui vogliamo ricavare l'automa simbolico questa volta è la nostra cara

$$S \to aSb \mid ab \tag{9.7}$$

Empezamos con lo stato 0.
Kernel dello stato 0:

$$S' \to \cdot S, x_0$$

Ci salviamo il valore di x_0.

$$\begin{aligned} Sistema: \\ x_0 = \{\$\} \end{aligned}$$

Passiamo alla chiusura del kernel:

$$\begin{aligned} S \to \cdot aSb, x_0 \\ S \to \cdot ab, x_0 \end{aligned}$$

Lo stato 0 non ci regala altre chiusure, ma ci regala due belle transizioni, una per a ed una per S.

Analizziamo quindi la transizione $\tau(0, S)$, che ci porta nel nuovo stato 1.
Kernel dello stato 1:

$$S' \to S\cdot, x_1$$

In questo caso $\beta = \varepsilon$, quindi il lookahead set del kernel sarà ugualeal lookahead set della produaizone che ci ha portati in 1, ci salviamo questo risultato.

$$\begin{aligned} Sistema: \\ x_0 = \{\$\} \\ x_1 = x_0 \end{aligned}$$

Questo stato non ci offre altri divertimenti, passiamo oltre.

Analizziamo la transizione $\tau(0, a)$, che ci porta nel nuovo stato 2.
Kernel dello stato 2:

$$S \to a \cdot Sb, x_2$$
$$S \to a \cdot b, x_3$$

In questo caso il lookahead set di entrambe le produzioni del kernel sarà uguale al lookahead della produzione che ci ha portato in 2, poiché $\beta = \varepsilon$ in entrambi i casi.

$$Sistema:$$
$$x_0 = \{\$\}$$
$$x_1 = x_0$$
$$x_2 = x_0$$
$$x_3 = x_0$$

In questo caso il primo item del kernel richiede che venga calcolata una chiusura:

$$S \to \cdot aSb, \{b\}$$
$$S \to \cdot ab, \{b\}$$

Ma questa volta dobbiamo prestare attenzione perché la chiusura per S ha come β il carattere b, quindi il lookahead set dei due item appena ottenuti è $\{b\}$. Ora che abbiamo calcolato la chiusura del kernel possiamo passare alla prossima transizione.

Analizziamo quindi $\tau(2, S)$, che ci porta allo stato 3 tramite l'item $[S \to a \cdot Sb, x_2]$.
Kernel dello stato 3:

$$S \to aS \cdot b, x_4$$

Orbene, aggiorniamo il nostro sistema equatoriale sapendo che $\beta = b$ mentre $\Delta = x_2$.

$$Sistema:$$
$$x_0 = \{\$\}$$
$$x_1 = x_0$$
$$x_2 = x_0$$
$$x_3 = x_0$$
$$x_4 = \{b\}$$

Lo stato 3 non ci offre ulteriori motivi di svago.

La prossima produzione che analizziamo è $\tau(2, b)$, che ci capitombola nel nuovo stato 4 tramite l'item $[S \to a \cdot b, \{x_3\}]$.
Kernel dello stato 4:

$$S \to ab\cdot, x_5$$

A questo punto devo aggiornare il sistema, tenendo conto che in questo caso non si applica la famosa forma $[A \to \alpha \cdot B\beta]$, quindi il lookahead set del kenrel di 4 sarà esattamente il lookahead set della produzione che ci ha portati in 4.

$$\begin{aligned} Sistema: \\ x_0 &= \{\$\} \\ x_1 &= x_0 \\ x_2 &= x_0 \\ x_3 &= x_0 \\ x_4 &= \{b\} \\ x_5 &= x_3 \end{aligned}$$

Lo stato è chiuso.

Passiamo ora ad analizzare la transizione $\tau(2, a)$ che, tramite gli item $[S \to \cdot aSb, \{b\}]$ e $[S \to \cdot ab, \{b\}]$, ci porta in un nuovo stato, lo stato 5.
Kernel dello stato 5:

$$\begin{aligned} S \to a \cdot Sb, x_6 \\ S \to a \cdot b, x_7 \end{aligned}$$

Ma c'è un colpo di scena, perché noi abbiamo già visto questo kernel una volta: è esattamente il kernel dello stato 2! Quindi la procedura ci richiede di aggiornare solo il lookahead set dello stato 2, senza generare un nuovo stato 5.

$$\begin{aligned} Sistema: \\ x_0 &= \{\$\} \\ x_1 &= x_0 \\ x_2 &= x_0 \cup \{b\} \\ x_3 &= x_0 \cup \{b\} \\ x_4 &= \{b\} \\ x_5 &= x_3 \end{aligned}$$

Ci resta da analizzare la transizione $\tau(3, b)$ che tramite l'item $[S \to aS \cdot b, \{x_4\}]$ ci porta al nuovo (per davvero questa volta) stato 5.
Kernel dello stato 5:

$$S \to aSb\cdot, x_6$$

In questo caso il lookahead set del kernel rimane uguale a quello della transizione che ci ha portato in 5.

$$\begin{aligned} Sistema: \\ x_0 &= \{\$\} \\ x_1 &= x_0 \\ x_2 &= x_0 \cup \{b\} \\ x_3 &= x_0 \cup \{b\} \\ x_4 &= \{b\} \\ x_5 &= x_3 \\ x_6 &= x_4 \end{aligned}$$

Questo stato è chiuso, quindi terminiamo la sua analisi.

Avendo analizzato le transizioni non ci rimane altro da fare se non risolvere il sistema di equazioni per valorizzare le nostre x.

$$\begin{aligned} x_0 = x_1 &= \{\$\} \\ x_2 = x_3 = x_5 &= \{\$, b\} \\ x_4 = x_6 &= \{b\} \end{aligned}$$

Andiamo quindi, per curiosità, ad analizzare gli stati in cui sono presenti le riduzioni ed a piazzarvi i lookahead set:

- in 1 abbiamo la riduzione $S' \to S\cdot,\ \{\$\}$;
- in 4 abbiamo $S \to ab\cdot,\ \{\$, b\}$;
- in 5 abbiamo $S \to aSb\cdot,\ \{b\}$.

Notimao quindi che non si presentano conflitti, quindi la grammatica è di tipo LALR.

Capitolo 10

Analisi semantica

10.1 Introduzione all'analisi semantica

10.1.1 Grammatiche attribuite

Nei capitoli precedenti abbiamo trattato esaustivamente la fase dell'analisi sintattica, in questo capitolo è finalmente arrivato il momento di passare all'analisi semantica.

Per approcciarsi a questa fase del processo della compilazione è necessario introdurre il concetto di grammatica attribuita (Syntax-Directed Definitions, da ora in poi SDD).

Questo tipo di grammatica è tale e quale ad una grammatica di quelle che siamo abituati ad utilizzare ormai quotidianamente, ma ci sorprende con due elementi aggiuntivi:

- **Attributi** che sono associati ai simboli della grammatica e possono essere numeri, tipi, riferimenti alla tabella dei simboli ecc;
- **Regole** che sono associate alle produzioni della grammatica e di norma sono funzioni degli attributi dei simboli della produzione.

Sia simboli che regole sono utilizzati per compiere l'analisi semantica, ovvero dare un valore a quello che è espresso da una parola di un a certa grammatica.

10.1.2 Tipi di attributi

Gli attributi sono divisi in due categorie:

- **attributi sintetizzati**, sono gli attributi del driver di una produzione, questi sono definiti in funzione degli attributi dei simboli del body della produzione;

- **attributi ereditati**, sono gli attributi dei non-terminali nel body della produzione e sono definiti in funzione degli altri simboli del body della produzione.

Il lettore accorto si sarà reso cont senz'altro che non abbiamo definito degli attributi per i caratteri terminali, questo perché gli attributi dei terminali sono sempre noti: sono dati dall'analisi lessicale e di conseguenza non serve alcuna regola per calcolarli. Questo concetto risulta più chiaro se si tiene bene a mente che gli attributi servono per valorizzare i caratteri di una parola: i non-terminali hanno un valore che dipende da che funzione rappresentano e da quali terminali "utilizzano", i terminali invece rappresentano le variabili, ovvero dei valori ben definiti che sono già stati riconosciuti e salvati nel momento dell'analisi lessicale.

Di fatto l'analizzatore lessicale prende il codice, sostituisce i terminali con i loro identificatori e ne memorizza il valore. L'analizzatore lessicale passa poi all'analizzatore sintattico la tabella dei simboli in cui ogni identificatore ha associato il suo valore lessicale (che in seguito indicheremo con la keyword `lexval`).

È arrivato, come da prassi, il momento di applicare la nostra filosofia del *learning by doing*, osservando un esempio di utilizzo di una grammatica attribuita.

10.1.3 Esempio

Prendiamo come esempio la nostra grammatica per la valutazione delle espressioni aritmetiche per valutare l'espressione rappresentata dall'albero di derivazione in Fig.10.1.

$$
\begin{aligned}
&V \to E \\
&E \to E + T \mid T \\
&T \to T * F \mid F \\
&F \to (E) \mid digit
\end{aligned}
$$

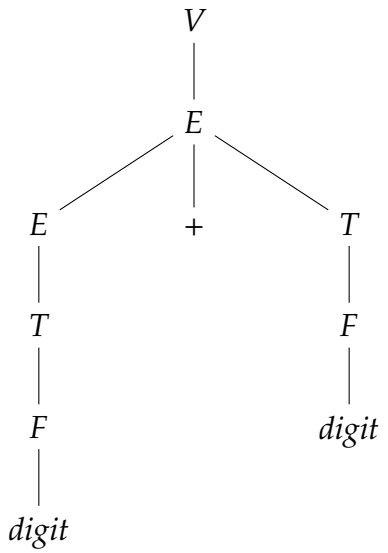


Figura 10.1: Albero di derivazione per una somma

Prendiamo il caso in cui il primo digit abbia assegnato come valore 3, mentre il secondo abbia valore 4. Questi valori, come spiegato prima, sono memorizzati nella tabella dei simboli.

Naturalmente essendo questa una somma tra 3 e 4 ci aspettiamo di trovare, una volta terminata la fase di valutazione, che il valore riportato in V sia 7.

Vediamo subito quale forma deve avere l'SDD per la grammatica che stiamo utilizzando:

$$V \to E \qquad \{V.val = E.val\} \tag{10.1}$$
$$E \to E_1 + T \qquad \{E.val = E_1.val + T.val\} \tag{10.2}$$
$$E \to T \qquad \{E.val = T.val\} \tag{10.3}$$
$$T \to T_1 * F \qquad \{T.val = T_1.val * F.val\} \tag{10.4}$$
$$T \to F \qquad \{T.val = F.val\} \tag{10.5}$$
$$F \to (E) \qquad \{F.val = E.val\} \tag{10.6}$$
$$F \to digit \qquad \{F.val = digit.lexval\} \tag{10.7}$$

Nota che i pedici numerici usati in alcune produzioni sono usati solo per differenziare il body dal driver, non hanno nessuna valenza semantica, di fatto E_1 ed E sono lo stesso non-terminale.

Significato delle regole quelle che abbiamo elencato sulla destra delle produzioni tra parentesi giraffe sono le regole che compongono una grammatica attribuita.

Tutto ciò che si trova all'interno delle regole sono gli attributi (nota però che i simboli + e $*$ in questo caso non sono attributi, ma sono proprio le operazioni matematiche).
Proviamo ora a dare una spiegazione del significato delle varie regole così da aiutare il lettore a capire meglio qual è lo scopo delle grammatiche attribuite.

Reg.10.1 la prima produzione è una produzione aggiuntiva che abbiamo inserito noi (un po' come $S' \to S$) e ci dice che quando incontriamo produzioni $V \to E$ il valore di V è $E.val$;

Reg.10.2 per produzioni come $E \to E_1 + T$ avremo che il valore di E corrisponde alla somma del sottoalbero E_1 più il valore di T (nota che E_1 è usato solo per differenziare da E);

Reg.10.3 se ho una produzione $E \to T$ il valore di E è uguale al valore di T;

Reg.10.4 se ho una produzione $T \to T_1 * F$ il valore di T corrisponde al valore del sottoalbero T_1 moltiplicato per il valore di F (nota che T_1 è usato solo per differenziare da T);

Reg.10.5 se ho una produzione del tipo $T \to F$ il valore di T è uguale a $F.val$;

Reg.10.6 se ho una produzione del tipo $F \to (E)$ il valore che assegno a F corrisponde a $E.val$;

Reg.10.7 se ho una produzione del tipo $F \to digit$ mi aspetto che il valore di F sia il valore riportato nella tabella dei simboli per tale $digit$.

Utilizzo dell'SDD Ora che abbiamo la grammatica attribuita possiamo utilizzarla appunto per dare un valore al parse tree della nostra somma.

Per valutare un albero di derivazione si utilizza un parse tree annotato. Questo è un parse tree in cui ogni nodo contiene le annotazioni che riguardano gli attributi del simbolo che si trova in quel nodo. Si può capire al volo cosa intendiamo guardando il parse tree annotato corrispondente al parse tree precedente (Fig.10.2) che è raffigurato qui sotto.

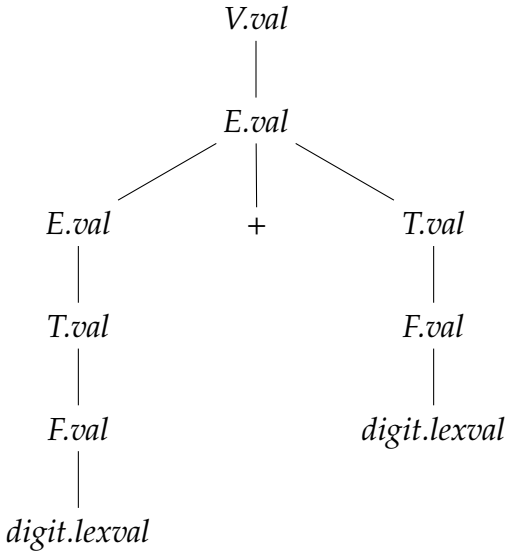


Figura 10.2: Albero di derivazione per una somma in 10.1

Ciò a cui si deve fare attenzione in questo caso è che ad ogni nodo è aggiunta l'annotazione riguardante il valore del nodo stesso.

Ma a cosa serve il parse tree annotato? È presto detto!
Il parse tree annotato viene letto dal basso verso l'alto seguendo le regole di attribuzione che sono segnate sui vari nodi per ottenere la valutazione dell'albero stesso. Vediamo ora questo procedimento passo per passo.

1. Partendo dal basso applichiamo innanzitutto la regola

$$F \rightarrow digit \qquad \{F.val = digit.lexval\}$$

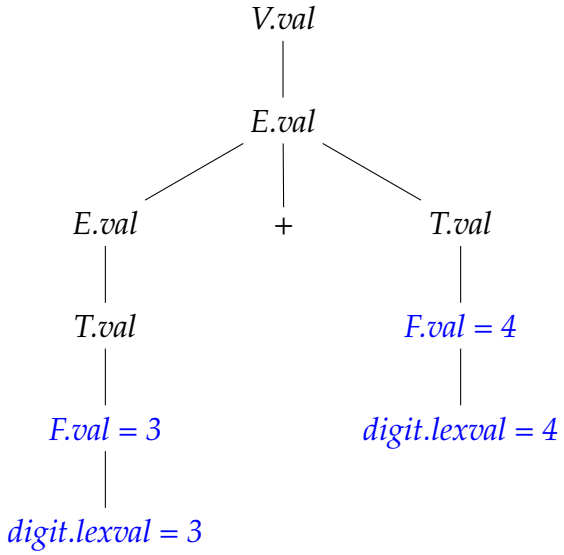


Figura 10.3: Passo 1 della valutazione tramite SDD

2. Il secondo passo prevede di applicare la regola

$$T \to F \qquad \{T.val = F.val\}$$

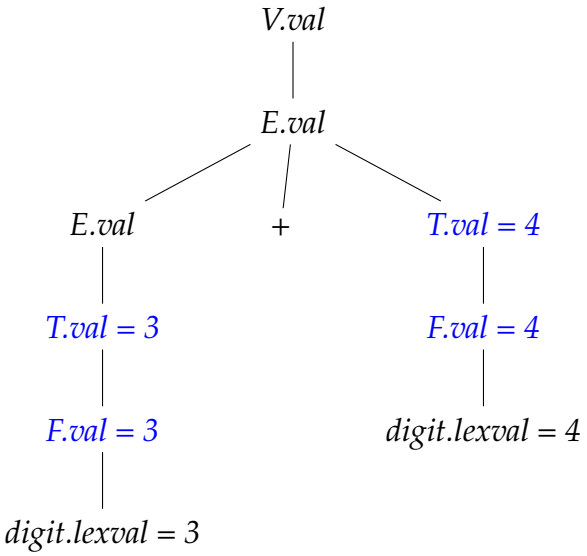


Figura 10.4: Passo 2 della valutazione tramite SDD

3. Successivamente applichiamo la regola

$$E \to T \qquad \{E.val = T.val\}$$

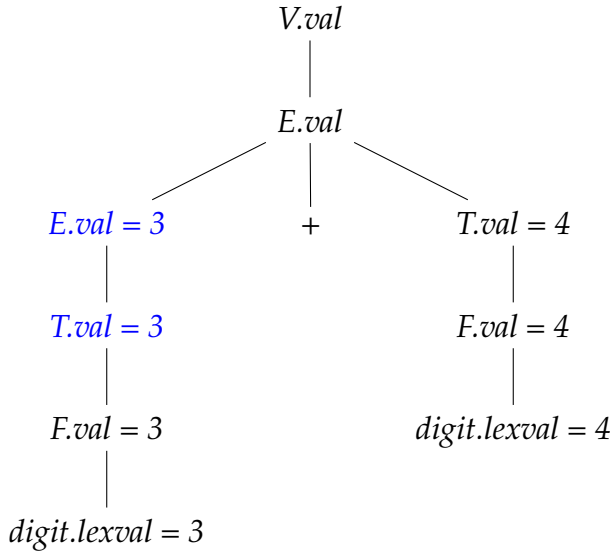


Figura 10.5: Passo 3 della valutazione tramite SDD

4. Il quarto passo prevede di applicare la regola

$$E \to E_1 + T \qquad \{E.val = E_1.val + T.val\}$$

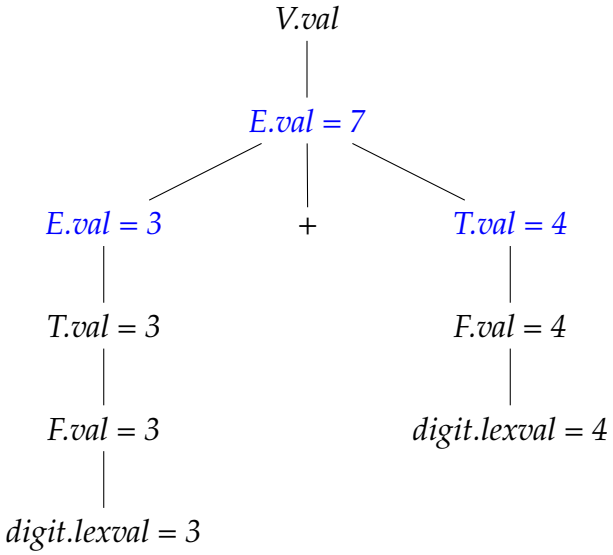


Figura 10.6: Passo 4 della valutazione tramite SDD

5. Infine, andiamo ad applicare la regola

$$V \to E \qquad \{V.val = E.val\}$$

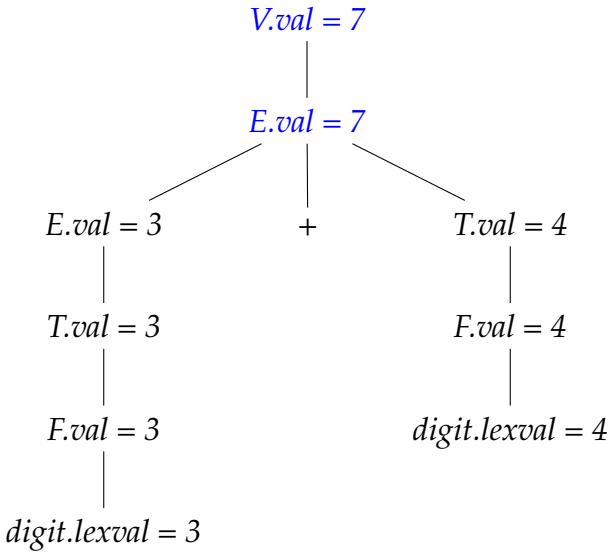


Figura 10.7: Passo finale della valutazione tramite SDD

Dovrebbe essere chiaro a questo punto come si possa utilizzare una grammatica attribuita per calcolare il valore di una certa parola di una data grammatica:

1. si crea il parse tree annotato per tale parola;
2. si utilizza il parse tree annotato in combinazione con le regole dettate dalla grammatica attribuita per valorizzare un nodo alla volta tutti i nodi del parse tree annotato;
3. una volta valorizzata la radice del parse tree annotato si è ricavato il valore della parola.

Una nota interessante riguardo all'albero di parsing annotato visto in questo esempio è che gli attributi che compaiono in tutti i nodi sono attributi sintetizzati.

10.1.4 Verificare se un parse tree annotato può essere valutato

Non è sempre detto che un parse tree annotato possa essere valutato, per verificare che ciò sia possibile dobbiamo utilizziare un grafo delle dipendenze e verificare che non vi siano conflitti tra le dipendenze.

Costruire il grafo delle dipendenze per definire un grafo delle dipendenze per un determinato parse tree annotato dobbiamo svolgere i seguenti passi:

- impostiamo un nodo nel grafo delle dipendenze per ogni attributo associato a qualche nodo del parse tree;
- per ogni attributo $X.x$ usato per definire l'attributo $Y.y$, creiamo un arco dal nodo di $X.x$ fino al nodo di $Y.y$ (rappresentando quindi la dipendenza di $Y.y$ da $X.x$).

Utilizzare il grafo delle dipendenze una volta creato il grafo delle dipendenze, per verificare che non vi siano conflitti dobbiamo trovare un ordinamento topologico per tale grafo; se un ordinamento topologico non esiste, allora l'albero di parsing annotato non è valutabile.
Se invece troviamo un ordinamento topologico per questo dependency graph allora l'SDD è valutabile ed abbiamo anche un ordine da seguire per la valutazione.

Ad esempio per il parse tree annotato dell'esercizio precedente otteniamo questo grafo delle dipendenze in cui le dipendenze sono indicate con le frecce azzurre (mentre la linea tratteggiata rappresenta gli archi del parse tree).

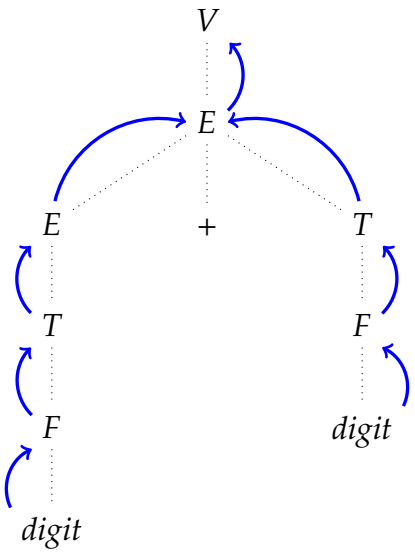


Figura 10.8: Albero di parsing annotato e relativo grafo delle dipendenze

In generale vale questa regola: quando tutti gli attributi sono di tipo sintetizzato allora una visita in postordine può sostituire sempre l'ordinamento topologico, capiremo in seguito le motivazioni dietro questa affermazione.

Riportiamo lo pseudocodice della visita in postordine, per chi non avesse ascoltato il Montre a suo tempo:

Algorithm 18: postorder(N)

```
1 foreach child C in N dalla sinistra do
2     postorder(C)
3 calcola attributi di N
```

Figura 10.9: Algoritmo della visita in postordine, versione PQ

Quando un SDD contiene sia elementi sintetizzati che ereditati allora non c'è la garanzia che un ordinamento topologico esista per tale SDD. Per esempio, se si ha una regola come

$$A \to B \qquad \{A.s = B.i; B.i = A.s + 7\}$$

si potrebbe trovare un ciclo all'interno del grafo delle dipendenze con una forma simile:

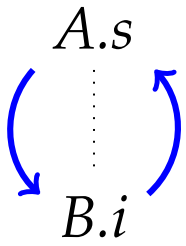


Figura 10.10: Conflitto di dipendenza

Quindi nel caso generale si calcola sempre l'ordinamento topologico dell'SDD, solo nel caso specifico di SDD costituiti solo da attributi sintetizzati ci basta la visita in postordine (che in questo caso corrisponde proprio ad un ordinamento topologico).

Esistono due classi di SDD per cui è sempre garantita l'esistenza di un ordinamento topologico:

- **S-attributed SDD** (grammatiche o SDD s-attribuiti): tutti gli attributi dell'SDD sono sintetizzati, in questo caso come già detto ci basta fare una visita in postordine;
- **L-attributed SDD**: ci possono essere anche attributi ereditati che rispettano però il seguente vincolo sulle eredità: si può ereditare solo dal padre o dai fratelli sinistri; scrivendo in linguaggio matematico tale vincolo:
  - per ogni produzione $A \to X_1 ... X_n$ la definizione di ogni $X_j.i$ utilizza solamente
    - attributi ereditati di A oppure
    - attributi sintetizzati o ereditati dei fratelli sinistri di X_j, ovvero $X_1 ... X_{j-1}$

Gli SDD S-attribuiti sono ideali per il parsing bottom-up, perché si può creare e valutare il parse tree attribuito in contemporanea al parsing della stringa.
Viceversa gli SDD L-attribuiti sono invece ideali per il parsing top-down, perché si può creare e valutare il parsing tree attribuito in contemporanea al parsing della stringa.

10.1.5 Un altro esempio di utilizzo dell'SDD

Grammatiche differenti pongono sfide differenti nella definizione degli SDD. Nel primo esercizio di applicazione dell'SDD abbiamo studiato una grammatica con

ricorsione sinistra, osserviamo ora un caso di grammatica LL(1), consideriamo il caso della grammatica per le espressioni aritmetiche $+$ e $*$.

$$
\begin{aligned}
V &\to E \\
E &\to TE' \\
E' &\to +TE \mid \varepsilon \\
T &\to FT' \\
T' &\to *FT' \mid \varepsilon \\
F &\to (E) \mid digit
\end{aligned}
$$

Volgiamo definire l'SDD che ci permette di valutare tale grammatica.

Vediamo di costruire il parse tree di $3 * 5$ Non riportiamo l'intero albero perché la Quaglia non l'ha disegnato. Questo è un sottoalbero per l'albero di parsing della formula analizzata:

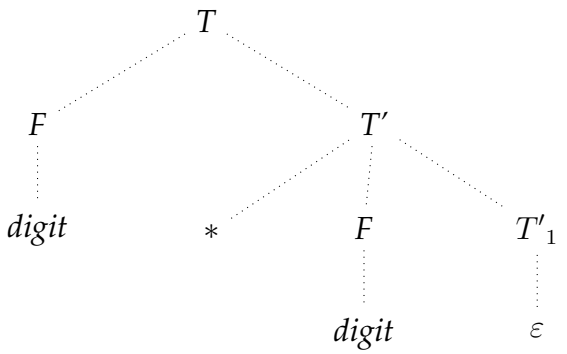


Figura 10.11: Pezzo del parse tree di $3 * 5$

Osserviamo come in questo caso abbiamo una digit che si trova nel sottoalbero sinistro della radice, mentre l'operatore $*$ e l'altra digit si trovano nel sottoalbero destro della radice. Come facciamo a fare in modo che $T.val$ abbia valore 15?
Chiediamo l'intervento degli attributi ereditati
Questo è il caso in cui possiamo aiutarci con l'utilizzo di attributi ereditati, presentiamo in Fig.10.12 come si possono utilizzare gli attributi ereditati; in blu abbiamo ancora una volta gli archi del grafo delle dipendenze e le etichette $.i$ e $.s$ rappresentano rispetivamente attributi ereditati e sintetizzati.

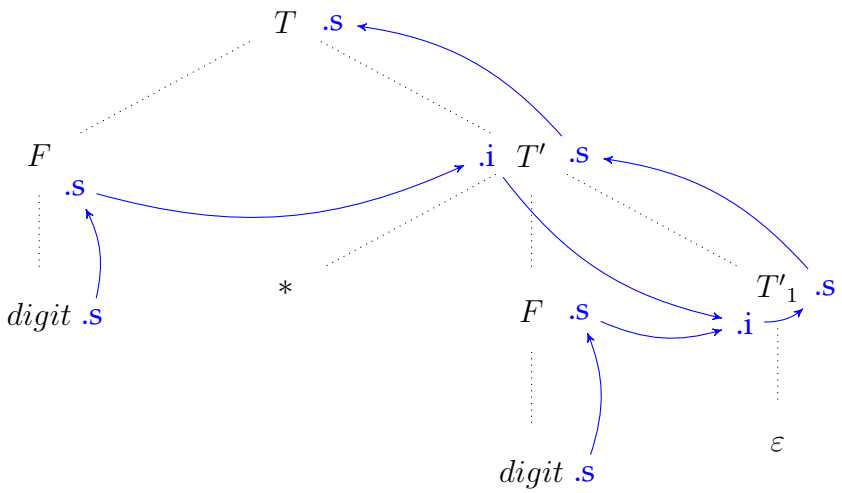


Figura 10.12: Pezzo del grafo delle dipendenze di $3 * 5$

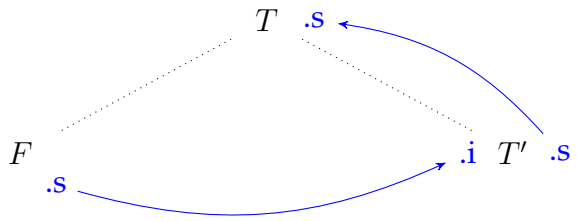
Vediamo quindi come in questo caso sia necessario utilizzare gli attributi ereditati, che però aggiungono un livello di complessità.

Quando studiamo la regola per una produzione dobbiamo usare solo le informazioni che abbiamo a quel livello, in quella produzione, così viene affrontata la costruzione dell'SDD.

Andiamo ora a vedere nello specifico come viene affrontata la questione delle dipendenze quando si generano le regole per questo determinato SDD. La prima cosa da notare è che quando il valore di T' viene copiato in T contiene già il valore della moltiplicazione. In questo caso la regola che andiamo ad associare alla produzione rappresentata è

$$T \to FT' \qquad \{T'.i = F.s;\ T.s = T'.s\}$$

ovvero passiamo a T' come valore ereditato $F.s$ ed assegnamo a T come valore sintetizzato $T'.s$, il che viene rappresentata graficamente così:



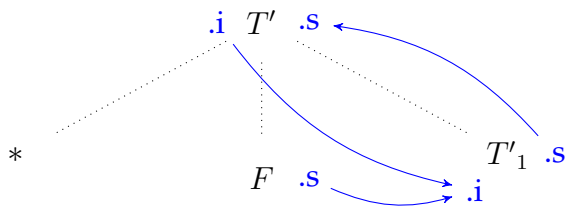
Se andiamo invece ad osservare come viene generato il valore di T' vediamo dal pezzo di parse tree annotato in Fig.10.12 che in pirmis ereditiamo un elemento

dal fratello di sinistra, poi sappiamo che lo moltiplichiamo per il valore che ci è dato da $F.s$ per ottenere il valore di T'_1, che poi passeremo a T'.

La regola per questa produzione è quindi

$$T' \to *FT'_1 \qquad \{T'_1.i = T'.i * F.s;\ T'.s = T'_1.s\}$$

che si può rappresentare graficamente così:



Ora che abbiamo studiato i due punti critici dell'SDD per questa grammatica riportiamo l'SDD nella sua completezza.

$V \to E$	$\{V.s = E.s\}$
$E \to TE'$	$\{E.s = E'.s;\ E'.i = T.s\}$
$E' \to +TE'_1$	$\{E'.s = E'_1.s;\ E'_1.i = E'.i + T.s\}$
$E' \to \varepsilon$	$\{E'.s = E'.i\}$
$T \to FT'$	$\{T.s = T'.s;\ T'.i = F.s\}$
$T' \to *FT'_1$	$\{T'.s = T'_1.s;\ T'_1.i = T'.i * F.s\}$
$T' \to \varepsilon$	$\{T'.s = T'.i\}$
$F \to (E)$	$\{F.s = E.s\}$
$F \to digit$	$\{F.s = digit.lexval\}$

10.1.6 Traduzione durante il parsing

Il nostro obiettivo è considerare la traduzione direttamente durante la fase di parsing per poter ottimizzare le attività della compilazione. Questo ci permette di evitare una prima elaborazione sintattica per ottenere il parse tree e la seconda, semantica, per valutare gli attributi.

Il caso più semplice in cui queste due elaborazioni possono essere eseguite in modo congiunto lo si ha considerando una grammatica adatta al parsing bottom-up a cui possiamo applicare l'algoritmo di shift/reduce (i.e. SLR(1), LR(1), LALR(1)) e l'SDD è S-attribuito (i.e. gli unici attributi presenti sono sintetizzati). Come mai?

L'algoritmo di shift/reduce si può eseguire con complessità lineare e necessità di due pile AA: una per mantenere gli stati (stSt) mentre l'altra per la catena dei simboli per poter conservare la derivazione parziale (sySt). L'idea è quella che insieme a queste due pile possiamo mantenerne una terza, che chiameremo **semSt**, dove conserveremo gli attributi (e.g. nel caso di una grammatica di espressioni aritmetiche degli interi).

Esempio Algoritmo shift/reduce con semSt

Per poter fornire un'idea di come utilizzare questa pila vediamo un esempio per cui è possibile valorizzare identificatori ed espressioni nel momento in cui si hanno a disposizione tutte le informazioni sugli attributi dei simboli del body.

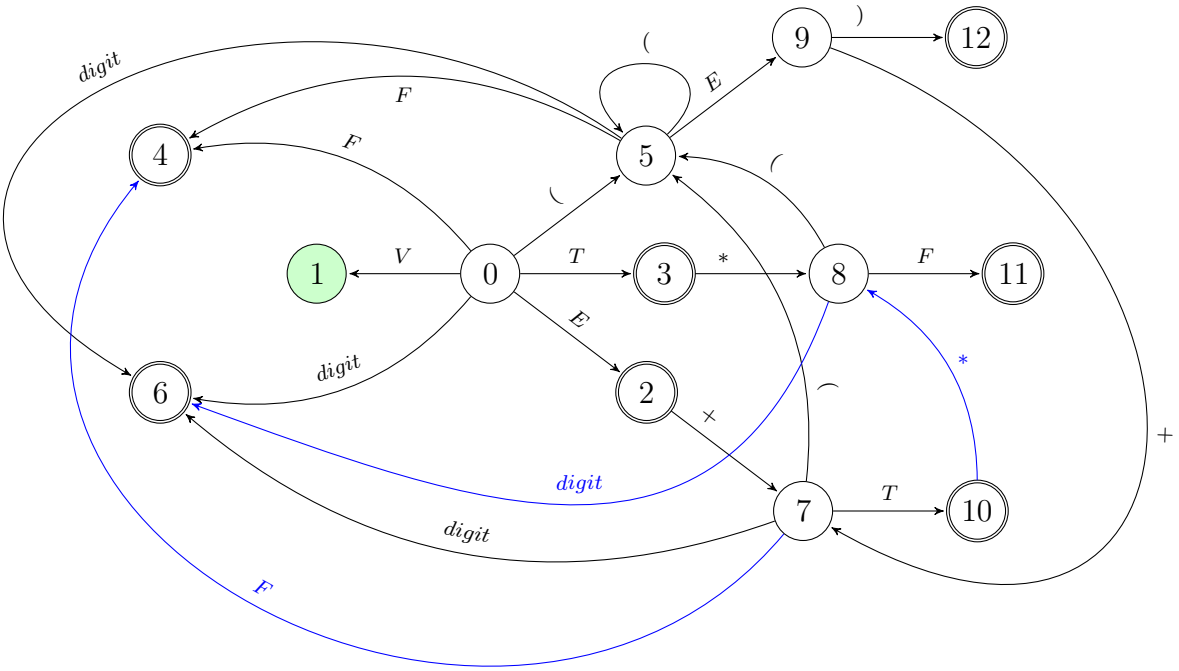
Consideriamo la seguente grammatica per definire le espressioni aritmetiche:

$$
\begin{aligned}
&V \to E \\
&E \to E + T \mid T \\
&T \to T * F \mid F \\
&F \to (E) \mid digit
\end{aligned}
$$

Il nostro obiettivo è quello di svolgere la traduzione durante il parsing ma per poter fare ciò abbiamo anche bisogno delle regole associate ad ogni produzione della grammatica

$$
\begin{aligned}
V &\to E && \{print(E.val)\} \\
E &\to E_1 + T && \{E.val = E_1.val + T.val\} \\
E &\to T && \{E.val = T.val\} \\
T &\to T_1 * F && \{T.val = T_1.val * F.val\} \\
T &\to F && \{T.val = F.val\} \\
F &\to (E) && \{F.val = E.val\} \\
F &\to digit && \{F.val = digit.lexval\}
\end{aligned}
$$

Come abbiamo potuto osservare nei capitoli precedenti, l'algoritmo shift/reduce necessita della tabella di parsing per operare. La cortesia di poter osservare che tipo di operazioni venissero eseguite purtroppo non è stata concessa a quei poveri disgraziati che erano a lezione ma, per poter comprendere appieno l'esercizio, ci sentiamo in dovere di fornire, a chiunque stia leggendo, l'automa caratteristico e la relativa tabella di parsing LALR(1).



	+	*	(	)	*digit*	$	*V*	*E*	*T*	*F*
0			s5		s6		1	2	3	4
1						Acc				
2	s7					r1				
3	r3	s8		r3		r3				
4	r5	r5		r5		r5				
5			s5		s6			9	3	4
6	r7	r7		r7		r7				
7			s5						10	
8			s5		s6					11
9	s7			s12						
10	r2	s8		r2		r2				
11	r4	r4		r4		r4				
12	r6	r6		r6		r6				

Tabella 10.1: Tabella di parsing LALR(1) Espressioni Aritmetiche

dove

r1: $V \to E$

r2: $E \to E + T$

r3: $E \to T$

r4: $T \to T * F$

r5: $T \to F$

r6: $F \to (E)$

r7: $F \to digit$

Eseguiamo l'algoritmo shift/reduce dato in input $w = digit + digit\$$ per la grammatica LALR(1) descritta poco sopra. Visto che vogliamo eseguire la traduzione degli attributi durante il parsing supponiamo che il primo abbia valore lessicale pari a 3 (i.e. $digit.lexval = 3$) mentre il secondo pari a 4; per questo motivo per poter mantenere traccia degli attributi faremo uso della pila **semSt** (semantic stack).

Come sempre iniziamo partendo dallo stato 0 con la seguente situazione iniziale:

$$\begin{aligned} w &= digit + digit\$ \\ stSt &= 0 \\ symSt &= \\ semSt &= \end{aligned}$$

Il primo simbolo che leggiamo dall'input buffer è $digit$ quindi, come è possibile osservare sia dall'automa caratteristico che dalla parsing table, dobbiamo eseguire un'operazione di `shift` 6. Oltre che valorizzare le pile come durante un normale esecuzione dell'algoritmo eseguiamo `push` $digit.lexval$ per poter inserire 3 sulla cima di $semSt$ a seguito dell'operazione di shift.

$$\begin{aligned} w &= \underline{digit} + digit\$ \\ stSt &= 0\ 6 \\ symSt &= digit \\ semSt &= 3 \qquad\qquad \texttt{push}\ digit.lexval \end{aligned}$$

A questo punto il simbolo letto dall'input buffer è $+$ e, visto che ci troviamo nello stato 6, abbiamo la possibilità di eseguire `reduce` $F \to digit$: a seguito di un'operazione di riduzione dobbiamo anche eseguire il codice associato (i.e. la regola) con la produzione $F \to digit$ (i.e. $\{F.val = digit.lexval\}$). In questo caso ciò si traduce in

$$\begin{aligned} w &= \underline{digit} + digit\$ \\ stSt &= 0\ 4 \\ symSt &= F \qquad\qquad \texttt{reduce}\ F \to digit \\ semSt &= 3 \qquad\qquad \texttt{pop}\ digit.lexval;\ \texttt{push}\ F.val = digit.lexval \end{aligned}$$

Nello stato 4 abbiamo nuovamente la possibilità di eseguire `reduce` $T \to F$ leggendo dall'input buffer un $+$: in questo caso la regola associata con la produzione è $\{T.val = F.val\}$ per cui il procedimento sarà simile a quello precedente

$$\begin{aligned} w &= \underline{digit} + digit\$ \\ stSt &= 0\ 3 \\ symSt &= T \qquad\qquad \texttt{reduce}\ T \to F \\ semSt &= 3 \qquad\qquad \texttt{pop}\ F.val;\ \texttt{push}\ T.val = F.val \end{aligned}$$

Il processo è analogo per lo stato 3 in cui eseguiremo una `reduce` $E \to T$ e applicheremo la regola associata $\{E.val = T.val\}$

$$\begin{aligned} w &= \underline{digit} + digit\$ \\ stSt &= 0\ 2 \\ symSt &= E \qquad\qquad \texttt{reduce}\ E \to T \\ semSt &= 3 \qquad\qquad \texttt{pop}\ T.val;\ \texttt{push}\ E.val = T.val \end{aligned}$$

Giunti finalmente nello stato 2 possiamo eseguire `shift` 7 che ci porta a consumare il simbolo $+$ dall'input buffer. Visto che abbiamo eseguito uno shift dovremmo caricare il simbolo $+$ sulla pila $semSt$ ma ciò creerebbe un'inconsistenza in quanto, tecnicamente, dovrebbe contenere solamente solo dei numeri

interi: per poter superare questa problematica carichiamo lo 0 come dummy in modo da mantenere l'allineamento con la pila dei simboli.

$$
\begin{aligned}
w &= \underline{digit + digit}\$ \\
stSt &= 0\ 2\ 7 \\
symSt &= E + \\
semSt &= 3\ 0 \qquad\qquad \texttt{push}\ dummy
\end{aligned}
$$

Dallo stato 7 possiamo eseguire `shift` 6 leggendo dall'input buffer $digit$: a questo punto possiamo caricare anche il valore del secondo digit sulla pila semantica.

$$
\begin{aligned}
w &= \underline{digit + digit}\$ \\
stSt &= 0\ 2\ 7\ 6 \\
symSt &= E + digit \\
semSt &= 3\ 0\ 4 \qquad\qquad \texttt{push}\ digit.lexval
\end{aligned}
$$

Il processo a questo risulta essere analogo a quello fatto precedentemente per il primo $digit$ quindi ci riserviamo la possibilità di non spiegare così nel dettaglio i prossimi passaggi

$$
\begin{aligned}
w &= \underline{digit + digit}\$ \\
stSt &= 0\ 2\ 7\ 4 \\
symSt &= E + F \qquad\qquad \texttt{reduce}\ F \to digit \\
semSt &= 3\ 0\ 4 \qquad\qquad \texttt{pop}\ digit.lexval;\ \texttt{push}\ F.val = digit.lexval
\end{aligned}
$$

$$
\begin{aligned}
w &= \underline{digit + digit}\$ \\
stSt &= 0\ 2\ 7\ 10 \\
symSt &= E + T \qquad\qquad \texttt{reduce}\ T \to F \\
semSt &= 3\ 0\ 4 \qquad\qquad \texttt{pop}\ F.val;\ \texttt{push}\ T.val = F.val
\end{aligned}
$$

Nello stato 10 eseguiamo la `reduce` $E \to E + T$ leggendo dall'input buffer \$. tale produzione è associata alla regola $\{E.val = E_1.val + T.val\}$ che ci permette tra le varie cose di eseguire la somma e di rimuovere il dummy dal $semSt$.

$$
\begin{aligned}
w &= \underline{digit + digit}\$ \\
stSt &= 0\ 2 \\
symSt &= E \qquad\qquad \texttt{reduce}\ E \to E + T \\
semSt &= 7 \qquad\qquad \texttt{pop}\ T.val;\ \texttt{pop}\ dummy;\ \texttt{pop}\ E_1.val; \\
&\qquad\qquad\qquad\qquad \texttt{push}\ E = E_1.val + T.val
\end{aligned}
$$

Dallo stato 2 possiamo eseguire questa volta `reduce` $V \to E$ in quanto il simbolo che leggiamo dall'input buffer è \$: come sempre associamo a tale produzione la relativa regola (i.e. $\{print(E.val)\}$).

$$
\begin{aligned}
w &= \underline{digit + digit}\$ \\
stSt &= 0\ 1 \\
symSt &= V \qquad\qquad \texttt{reduce}\ V \to E;\ Acc \\
semSt &= 7 \qquad\qquad \texttt{pop}\ E.val;\ print(E.val)
\end{aligned}
$$

Questo tipo di approccio è sempre possibile nel caso in cui l'SDD sia S-attribuito: essendo che siamo in grado di calcolare il valore degli attributi del padre di un sottoalbero solamente sulla base degli attributi dei suoi figli, il modo più facile di implementare l'algoritmo è eseguire le azioni nel momento in cui viene effettuata la riduzione (i.e. il momento in cui conosciamo tutti i valori dei figli).

Avendo la possibilità di fare un po' più di pre-processing si riesce comunque in bottom-up ad eseguire tale procedura per SDD che sono L-attribuiti ma naturalmente aumenta la complessità.

10.1.7 Tradurre Stringhe in Numeri

Analizziamo ora delle grammatiche che permettono di tradurre stringhe in numeri: l'idea è che data una certa stringa vogliamo ottenere il loro valore decimale.

Possiamo pensare di scrivere una grammatica che genera il linguaggio delle stringhe di digit e il relativo schema di traduzione.

La prima grammatica su cui ragioneremo è

$$\begin{aligned} S &\to Digits && \{print(Digits.v)\} \\ Digits &\to Digits_1 d && \{Digits.v = Digits_1.v * 10 + d.lexval\} \\ Digits &\to d && \{Digits.v = d.lexval\} \end{aligned}$$

La grammatica che abbiamo di fronte è LALR(1) e l'SDD è S-attribuito: l'ultima affermazione è verificabile dal fatto che il valore del padre dei sottolaberi possibili è sempre e solamente dipendente da quello dei figli. In questo caso dunque è possibile eseguire le due elaborazioni in modo congiunto e quindi possiamo calcolare il tutto durante il parsing.

Aggiungiamo una variazione dell'esercizio: ora vogliamo tradurre stringhe di digit al loro valore interpretato in decimale oppure in ottale.

Una possibile grammatica che genera il linguaggio descritto è questa:

$$\begin{aligned} S &\to Num \\ Num &\to o\ Digits \mid Digits \\ Digits &\to Digits\ d \mid d \end{aligned}$$

Quando ci troviamo nel caso di una stringa con il simbolo o davanti intendiamo che vogliamo ottenere il valore ottale della sequenza di digit.

La grammatica anche in questo caso è LALR(1) però abbiamo un problema: non è possibile sintetizzare le informazioni nello stesso modo di prima in quanto queste non sono disponibili nel momento in cui ci servono. Ragionando infatti sulla derivazione $S \Rightarrow Num \Rightarrow oDigits$ ci si rende conto del fatto che il simbolo o si trova nel sottoalbero sinistro dell'albero mentre i $Digits$ in quello destro: questo vuol dire che di fatto non siamo in grado di calcolare il valore fino a quando non raggiungiamo la cima dell'albero e quindi ciò significa che non ci troviamo di fronte a un SDD S-attribuito (il valore del sottoalbero di destra dipende infatti dal valore dei fratelli e non solamente da quello dei figli).

Per poter raggiungere i nostri obbiettivi a livello di efficienza dobbiamo riprovare modificando la grammatica in modo che ci consenta di svolgere entrambe le operazioni contemporaneamente.

Per poter ottenere un risultato migliore potremmo inserire due nuovi non terminali in modo da poterci aiutare con le regole in un secondo momento: consideriamo infatti di modificare la grammatica nel modo seguente

$$
\begin{aligned}
S &\to Num \\
Num &\to O\ Digits \mid D\ Digits \\
O &\to o \\
D &\to \varepsilon \\
Digits &\to Digits\ d \mid d
\end{aligned}
$$

Di per sé non sembra essere cambiato molto ma in realtà in questo caso abbiamo la possibilità di inserire delle regole dove possiamo valorizzare delle variabili globali in modo da stabilire la base:

$$
\begin{aligned}
S &\to Num && \{print(Num.v)\} \\
Num &\to O\ Digits && \{Num.v = Digits.v\} \\
Num &\to D\ Digits && \{Num.v = Digits.v\} \\
O &\to o && \{base = 8\} \\
D &\to \varepsilon && \{base = 10\} \\
Digits &\to Digits_1\ d && \{Digits.v = Digits_1.v * base + d.lexval\} \\
Digits &\to d && \{Digits.v = d.lexval\}
\end{aligned}
$$

Stabilendo separatamente la base mediante le regole abbiamo dunque la possibilità di poter svolgere tutti i calcoli necessari senza rinunciare all'efficienza.

10.2 Alberi di Sintassi Astratta

10.2.1 Introduzione

Arriviamo finalmente a parlare degli alberi di sintassi astratta (Abstract Syntax Trees) che altro non sono che una rappresentazione compatta degli alberi di derivazione.

Un albero di sintassi astratta si può ottenere dal rispettivo albero di derivazione di una parola tramite dei passi di semplificazione, per avere chiaro a cosa ci riferiamo facciamo subito un esempio. Sia data la grammatica

$$
\begin{aligned}
E &\to E + T \mid T \\
T &\to T * F \mid F \\
F &\to (E) \mid id
\end{aligned}
$$

Se vogliamo fare il parsing della parola $digit + digit$ otteniamo i seguenti alberi di parsing e di sintassi astratta.

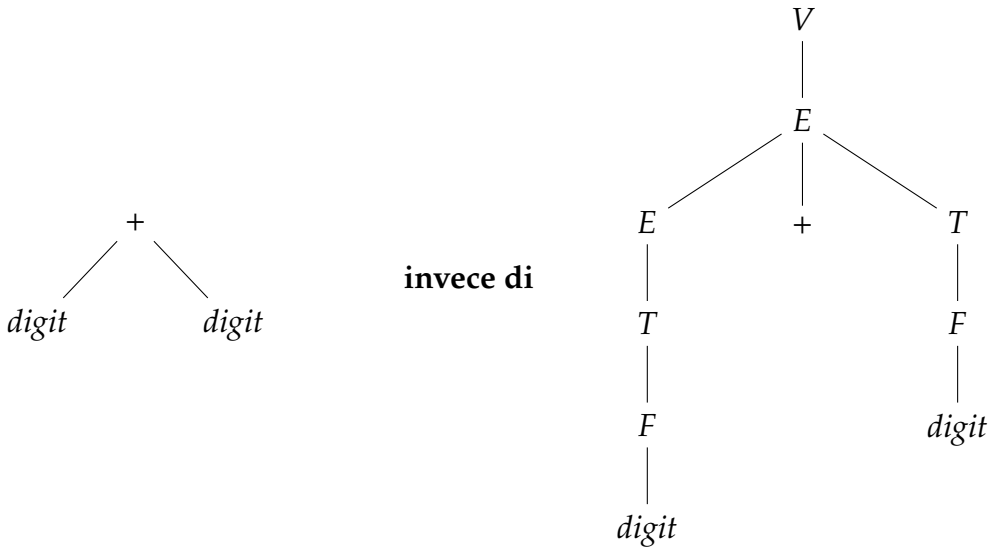


Figura 10.13: Due alberi per la parola $digit + digit$: sulla sinistra albero di sintassi astratta, sulla destra albero di derivazione

Si noti come l'albero della sintassi astratta a sinistra sia molto più compatto.

Memorizzazione degli AST Di fatto questa struttura va a creare in memoiria un albero in cui ogni nodo rappresenta un passaggio intermedio del processo descritto dalla parola che stiamo analizzando. L'AST ottenuto dalla semplificazione dell'albero di derivazione viene quindi salvato nodo per nodo nel seguente modo:

- le foglie vengono salvate come link a record nella tabella dei simboli: viene creata una foglia per ogni identificatore ed ogni valore diretto (es.: valore numerico) presente nella parola che stiamo parsando;
- i nodi intermedi dell'albero costituiscono la struttura topologica dell'albero e rappresentano tutte le operazioni compiute tra variabili o valori diretti (quindi tutte le operazioni tra altri nodi).

10.2.2 Creazione di un AST

L'AST si può creare semplicemente dopo aver ottenuto l'albero di derivazione tramite delle operazoni di compattazione, tuttavia il nostro obiettivo è quello

di calcolare l'AST direttamente durante la fase di parsing, ovvero contemporaneamente all'analisi sintattica. Per fare questo dobbiamo immaginare di avere a disposizione due funzioni:

- `newLeaf(label, val)` che va a salvare un nodo foglia con due campi: l'etichetta ed il valore;
- `newNode(label, c1, ..., ck)` che crea un nodo con k figli, l'etichetta (che indica l'operazione rappresentata dal nodo) è `label` mentre `c1, ..., ck` sono i riferimenti ai nodi figli;

Applicando queste due funzioni all'esempio precedente (Fig.10.13) avremo l'operazione $+$ rappresentata da un nodo che avrà come etichetta $+$ e come figli i due nodi $digit$. I nodi $digit$ saranno nodi foglia con identificatore e valore relativo (il valore sarà quello che ci viene ritornato dall'analisi lessicale, quindi lo possiamo ricavare con $digit.lexval$).

Funzioni di attribuzione Per creare l'albero di sintssi astratta durante la fase di parsing dobbiamo definire delle regole semantiche (dette anche funzioni di attribuzione o azioni semantiche) che, ogni volta che incontriamo una riduzione, ci indichino quando e come istanziare un nodo per l'AST durante il parsing.
Vediamo un esempio: data la grammatica LALR per operazioni algebriche di somma e sottrazione

$$\begin{aligned} &E \to E + T \mid E - T \mid T \\ &T \to (E) \mid id \mid num \end{aligned} \tag{10.8}$$

abbiamo che le regole (le funzioni di attribuzione) per la creazione dell'AST durante il parsing sono le seguenti

$$\begin{aligned} &E \to E_1 + T && \{E.node = newNode('+', E_1.node, T.node)\} \\ &E \to E_1 - T && \{E.node = newNode('-', E_1.node, T.node)\} \\ &E \to T && \{E.node = T.node\} \\ &T \to (E) && \{T.node = E.node\} \\ &T \to id && \{T.node = newLeaf(id, id.entry)\} \\ &T \to num && \{T.node = newLeaf(num, num.lexval)\} \end{aligned} \tag{10.9}$$

Si ricorda che i pedici numerici usati in alcune produzioni sono usati solo per differenziare il body dal driver, non hanno nessuna valenza semantica, di fatto E_1 ed E sono lo stesso non-terminale.

Analizziamo velocemente le regole più significative:

- $newLeaf(num, num.lexval)$ va a prendere il valore di num che ci è dato dall'analisi lessicale;
- in modo simile $newLeaf(id, id.entry)$ va a creare un nodo foglia in cui abbiamo come etichetta l'identificatore mentre il valore del nodo si troverà all'interno della tabella dei simboli compilata in fase di analisi lessicale.

Per le altre regole il significato, abbastanza intuitivo, è quello della creazione di un nodo intermedio che mantiene come informazioni l'etichetta dell'operazione da compiere e i riferimenti ai nodi degli operandi necessari.

Esempio di creazione di un AST

Ora che abbiamo le regole da applicare per costruire un albero di sintassi astratta, andiamo a costruirne uno per la parola $id - num + id$ derivata dalla grammatica appena vista (10.8), immaginando che la parola $id - num + id$ sia arrivata dall'analisi lessicale di $a - 4 + c$.
Innanzitutto dobbiamo arrivare alla fase del parsing, quindi ci dobbiamo procurare la tabella di parsing.
O almeno questo ci saremmo aspettati da un esercizio normale, ma invece no! La nostra guida spirituale a questo punto ci fa notare che abbiamo imparato a fare le tabelle di parsing per imparare a non farle più: dato che ci serve solo l'ordine delle produzioni utilizzate per ottenere la parola che stiamo analizzando possiamo barare e fare la derivazione a mente.

Quindi, tornando a noi e saltando qualche passaggio (che forse verrà inserito in seguito, dipende da quanto verranno pagati gli scrittori della dispensa), la sequenza di riduzioni che si ottiene dal parsing della parola $id - num + id$ è la seguente:

$$
\begin{aligned}
T &\to id \\
E &\to T \\
T &\to num \\
E &\to E - T \\
T &\to id \\
E &\to E + T
\end{aligned}
$$

Si invita il lettore a fermarsi un attimo e rileggere queste produzioni per convincersi che siano esattamente le riduzioni che si ottengono dal parsing bottom-up della parola data; si faccia attenzione all'ordine in cui si incontrano le riduzioni, questo è proprio l'ordine che ci si aspetta dalla procedura bottom-up, ovvero l'ordine inverso della procedura che dallo start symbol ci porta alla parola desiderata.

Vediamo ora come viene creato in memoria l'Abstract Syntax Tree mentre stiamo svolgendo il parsing della stringa. Ogni volta che incontriamo una riduzione andiamo ad aggiungere un nodo all'AST che stiamo creando. Quale forma avrà questo nodo ci è dato dalle funzioni di attribuzione dei nodi dell'AST. Osserviamo quindi quale forma viene ad ottenere l'abero di sintassi astratta della nostra parola $id - num + id$ mentre incontriamo le riduzioni passo dopo passo.

1. La prima riduzone che incontriamo è $T \rightarrow id$, che ha come funzione di attribuzione $\{T.node = newLeaf(id, id.entry)\}$; la rappresentazione grafica di quello che si ottiene in questo caso è la seguente:

Figura 10.14: Creaizone del nodo foglia per l'identificatore che referenzia la variabile a

2. La seconda riduzione che incontriamo, $E \rightarrow T$, ci porta ad un semplice passaggio di rinominazione di un nodo dell'AST, in quanto la sua azione semantica è $\{E.node = T.node\}$.

Figura 10.15: Rinominazione del $T.node$ in $E.node$

3. Il terzo passo ci porta ad usare la riduzione $T \rightarrow num$, con annessa funzione di attribuzione $\{T.node = newLeaf(num, num.lexval)\}$, che crea una nuova foglia con valore numerico associato.

Figura 10.16: Creazione di una nuova foglia con valore numerico associato

4. La prossima riduzione che incontriamo è $E \rightarrow E_1 - T$, con azione semantica $\{E.node = newNode('-', E_1.node, T.node)\}$, che viene rappresentata graficamente creando un nuovo nodo intermedio per l'AST, la cui forma può essere osservata nella figura sottostante.

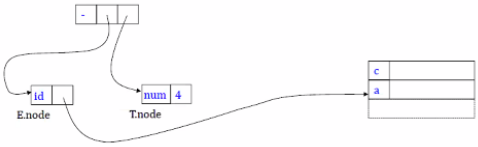


Figura 10.17: Inserimento del nodo intermedio che rappresenta la sottrazione

5. In seguito ci scontriamo di nuovo in una riduzione $T \rightarrow id$, che ha come funzione di attribuzione $\{T.node = newLeaf(id, id.entry)\}$, sappiamo già come funziona dal primo punto di questa lista. La rappresentazione grafica del nostro AST a questo stadio è la seguente:

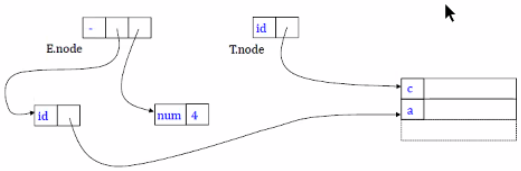


Figura 10.18: Creazione di un nodo foglia per id associato a c

6. Infine applichiamo la riduzione $E \rightarrow E_1 + T$, che prevede l'azione semantica $\{E.node = newNode('+', E_1.node, T.node)\}$ e va a completare il nostro AST che possiamo finalmente osservare in tutta la sua bellezza:

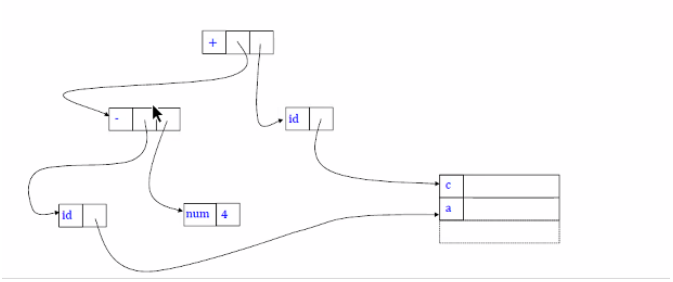


Figura 10.19: Inserimento del nodo intermedio per la rappresentazione della somma

Riassumendo Il goal che ci siamo fissati all'inizio di questo esercizio era di costruire *on the fly* l'albero di sintassi astratta durante la fase di parsing per la parola $id - num + id$ derivata grammatica LALR per le espressioni aritmetiche (10.8). Anche se abbiamo barato, utilizzando un parsing già fatto, è facile accorgersi che le funzioni di attribuzione che abbiamo usato ora le avremmo potute usare in contemporanea all'esecuzione del parsing, usandole direttamente ogni volta che avremmo incontrato una riduzione.
Per quanto riguarda le riduzioni con soli terminali nel body (che generano le foglie) abbiamo utilizzato questi attributi sintetizzati:

Id.entry per le variabili;

Num.lexval per i valori numerici.

Mentre per le riduzioni con non-terminali nel body abbiamo utilizzato delle funzioni di attribuzione semantica che prevedono di inserire i seguenti attributi sintetizzati:

E-node per le riduzioni con driver E;

T-node per le riduzioni con driver T.

Abbiamo visto quindi che si può ricavare l'AST mentre si esegue il nostro caro parsing bottom-up. Ma non è sempre così, l'albero di sintassi astratta in questo caso è stato costruito su una grammatica LALR con un SDD con tutti attributi sintetizzati (vedi 10.9, non ci sono attributi ereditati), è questo che ci ha permesso di costruire l'AST mentre costruivamo l'albero di parsing.

Sottolineamo questo aspetto per dire che nel caso in cui abbiamo un SDD L-attribuito non è detto che si possa compiere la costruzione dell'AST mentre si fa il parsing. Per aver ragion di ciò presentiamo subito un esempio in cui compaiono appunto attributi ereditati.

10.2.3 Esempio di creazione di un AST con attributi ereditati

L'esercizio in questo caso ci richiede di calcolare gli attributi per definire le funzioni di attribuzione per le produzioni della grammatica così definita:

$$\begin{aligned} E &\to TE' && \{...\} \\ E' &\to +TE'_1 && \{...\} \\ E' &\to -TE'_1 && \{...\} \\ E' &\to \varepsilon && \{...\} \\ T &\to (E) && \{...\} \\ T &\to id && \{...\} \\ T &\to num && \{...\} \end{aligned} \tag{10.10}$$

In questo caso dobbiamo chiamare in causa gli attributi ereditati, e la motivazione è presto detta. Per visualizzare la situazione possiamo andare a rivedere l'SDD che abbiamo costruito per le operazioni $+$ e $*$ nella prima parte riguardo l'analisi semantica (Fig.10.11, ma si può osservare tranquillamente la figura qui sotto per rendersi conto). In quel caso i valori sintetizzati in sottoalberi fratelli devono essere utilizzati da altri sottoalberi fratelli per ricavare il loro proprio valore, per esser chiari riportiamo qui la figura.

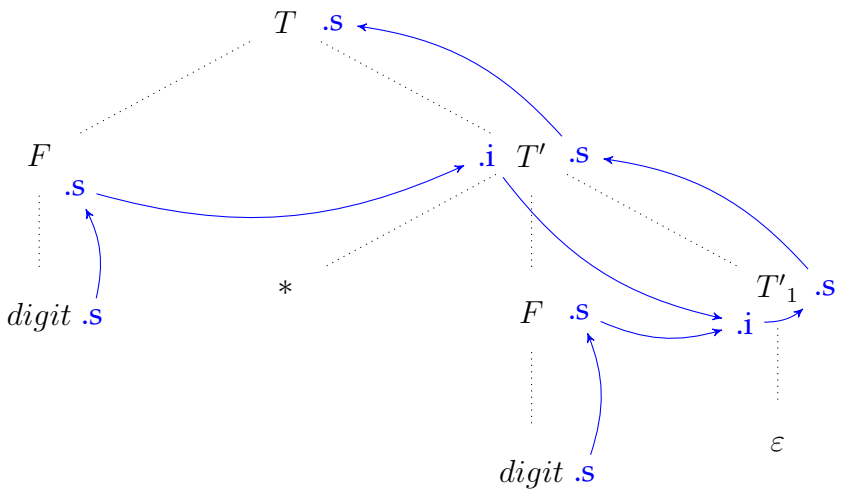


Figura 10.20: Pezzo del grafo delle dipendenze di $3 * 5$

Osservando quest'immagine è esemplice notare che per risolvere il valore di T' abbiamo bisogno dei valori nel sottoalbero di T' ma anche del valore di F che è fratello sinistro di T', quindi abbiamo bisogno di introdurre attributi ereditati.

Lo stesso avviene per la grammatica che stiamo considerando ora, quella per gli operatori $+$ e $-$ di cui dobbiamo completare l'SDD (10.10).

Il problema in questo caso specifico è che non possiamo sintetizzare tutte le informazioni che ci servono perchè, uno degli elementi necessari per ricavare il valore dell'espressione e per la formazione dell'abstract syntax tree, non è presente nello stesso sottoalbero nel quale troviamo l'operatore aritmetico e l'altro operando coinvolto nell'operazione.

Utilizzando gli attributi ereditati non si riesce ad eseguire in maniera immediata tutte le operazioni durante il parsing della grammatica: un'opzione, come già citato più volte, potrebbe essere quella di agire sulla grammatica in modo che, durante un'esecuzione del parsing bottom-up, si possa ottenere lo stesso risultato di una grammatica S-attribuita. Senza soffermarci troppo sui dettagli, tale operazione passerebbe per l'aggiunta di marker e ε-transizioni in modo che gli attributi vengano sintetizzati e non ereditati.

In questo caso per poter trasportare le informazioni da un ramo dell'albero ad un altro è necessario utilizzare degli attributi ereditati: in modo simile a quanto fatto per calcolare il valore delle espressioni aritmetiche, in questo caso assegniamo dei riferimenti a dei nodi.

Per poter capire al meglio che tipo di regole devono essere applicate è necessario partire da un parse tree: prendiamo dunque come esempio particolare la parola $id - num$, ottenuta dall'analisi lessicale di $a - 4$; senza troppa fatica dovremmo poterci accorgere che usando una derivazione di tipo leftmost i passi di derivazione sono

$$E \Rightarrow TE' \Rightarrow idE' \Rightarrow id - TE' \Rightarrow id - numE' \Rightarrow id - num$$

Il nostro *parse tree* in questo particolare caso è così costruito

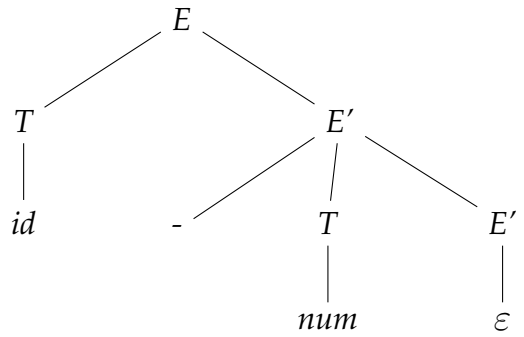


Figura 10.21: Parse tree attribuiti ereditati

Dal parse tree possiamo osservare in modo più esplicito che il calcolo del sottoalbero destro (con radice in E') necessita del sottoalbero sinistro (con radice in T) e quindi avremo bisogno degli attributi ereditati (quindi non potremo eseguire il parsing e la costruzione dell'abstract syntax tree contemporaneamente).

Facendo delle osservazioni di questo tipo siamo in grado di costruire la prossima struttura che ci servirà, cioè il *dependency graph*:

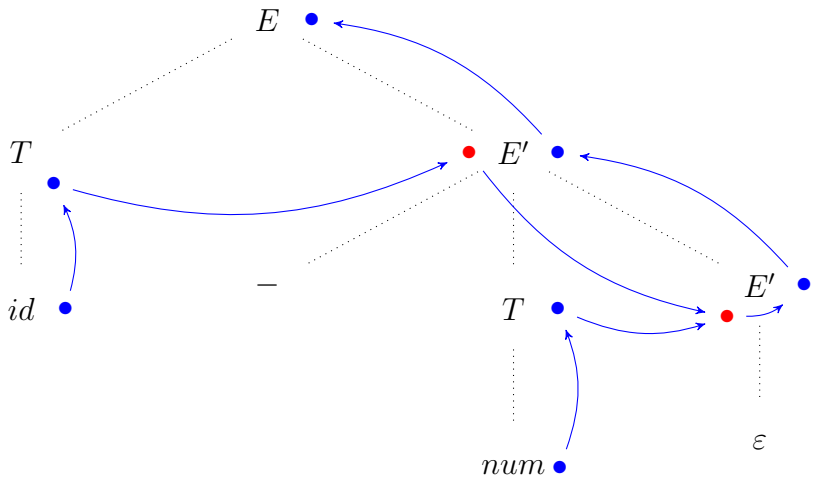


Figura 10.22: Dependency graph attributi ereditati

Può sembrare difficile ottenere questa struttura dal nostro parse tree, tuttavia i suoi archi rappresentano semplicemente le dipendenze tra gli attributi (i.e. quali sono necessari per poter effettuare una certa operazione); nella figura possiamo inoltre vedere quali fra gli attributi sono sintetizzati (in blu) e quali ereditati (in rosso).

Per rendere più chiaro il grafo utilizziamo un esempio: il sottoalbero sinistro possiede un arco da id a T (coincidente con quello del parse tree) perché sottolineiamo che, una volta effettuata l'operazione di riduzione $T \to id$, T mantenga un riferimento a id; visto che id è contenuto nel sottoalbero di T (i.e. è uno dei suoi figli) tale attributo è sintetizzato.

La situazione cambia se vogliamo esaminare l'arco esattamente successivo: l'arco da T ad E' infatti non coincide con nessun altro arco del parse tree e il suo scopo è quello di trasmettere i riferimenti di T al fratello E'. Questa operazione viene fatta perchè, per poter calcolare in questo caso la differenza, è necessario che E' riceva le informazioni del fratello T: dovremo dunque servirci di un attributo ereditato.

A questo punto dobbiamo definire le **funzioni di attribuzione** associate alle produzioni: tale operazione come sempre non è facile da eseguire ma vedremo di analizzarla passo per passo.

Cominciamo inserendo le azioni semantiche delle produzioni finali della grammatica proposta: queste definiscono le foglie del nostro parse tree e dovranno occuparsi di generare gli elementi finali dell'sbstract syntax tree.

$$T \to id \qquad \{T.node = newLeaf(id, id.entry)\}$$
$$T \to num \qquad \{T.node = newLeaf(num, num.lexval)\}$$

Queste produzioni ci permettono di creare due foglie:

- una foglia con etichetta id e valore $id.entry$ (i.e. quindi un riferimento alla sua entry nella symbol table per poter recuperare successivamente le informazioni);
- una seconda foglia con etichetta num e valore $num.lexval$ (i.e. il valore ottenuto dall'analisi lessicale)

Come si può intuire, il nostro scopo con tali regole è fare in modo che in T.node siano memorizzate le informazioni della foglia che abbiamo creato in modo da poterle recuperare in seguito.

Come avevamo già potuto mostare in precedenza, il sottoalbero sinistro ha bisogno di passare le proprie informazioni al sottoalbero destro; visto che stiamo parlando di nodi fratelli non possiamo ottenere queste informazioni in altro modo se non ricorrendo agli attributi ereditati: ci servirà dunque una regola del tipo $E'.i = T.node$.

Associamo dunque per il momento la regola descritta alla produzione corretta

$$E \to TE' \qquad \{E'.i = T.node\}$$

Volendo chiarire subito alcuni possibili dubbi, sottolineo il fatto che questa non sarà l'unica regola che sarà associata a tale produzione (si è voluto optare per un aggiunta graduale) e che le produzioni a cui è stata aggiunta l'azione semantica sono quelle dove T ed E' compaiono nel body.

A questo punto possiamo passare alle produzioni che si occupano delle operazioni aritmetiche, nel nostro caso della produzione $E' \to -TE'$: per via del nostro esempio potremmo limitarci semplicemente ad eseguire il risultato dell'espressione $E'.i - T.node$ (ricordiamoci che precedentemente abbiamo fatto in modo che $E'.i$ contenesse il valore del sottoalbero sinistro), tuttavia dobbiamo

considerare anche i casi per cui vi potrebbero essere altre operazioni in seguito (e.g. id + id - num + id - ...). Per questo motivo facciamo in modo di passare ad E' il risultato di tale operazione in modo che possa utilizzarlo nel caso di produzioni successive: questo potrebbe essere visto come un esempio di ricorsione dove di fatto stiamo "mandando avanti" i risultati delle operazioni e dovremo in seguito capire qual è il nostro caso base. Aggiungiamo dunque una regola che fa al caso nostro

$$E' \to -TE'_1 \qquad \{E'_1.i = newNode('-', E'.i, T.node)\}$$

Grazie a questa azione semantica siamo in grado di generare un nodo con etichetta $-$ e con un riferimento per ogni operando: il riferimento di questo nodo viene passato poi ad $E'_1.i$, in questo modo riuscirà a svolgere le operazioni successive.

Come abbiamo detto precedentemente, dovremo prima o poi interrompere tale ricorsione e quindi necessitiamo di un caso base: banalmente smetteremo di fare nuove chiamate nel momento in cui non avremo più operatori e quindi nel caso in cui utilizzeremo la produzione $E' \to \varepsilon$. In questo momento dovremo riportare il risultato di tutta la nostra computazione (i.e. i nostri riferimenti): per fare ciò semplicemente andiamo ad assegnare a $E'.node$ i riferimenti ottenuti fino a quel momento

$$E' \to \varepsilon \qquad \{E'.node = E'.i\}$$

A questo punto abbiamo trattato il nostro caso base ma è necessario ritornare sulle produzioni già analizzate in modo da poter restituire il tutto al "chiamante": aggiungiamo dunque

$$\begin{aligned} E &\to TE' && \{E.node = E'.node\} \\ E' &\to -TE'_1 && \{E'.node = E'_1.node\} \end{aligned}$$

A questo punto possiamo riassumere le regole come segue

$E \to TE'$	$\{E.node = E'.node; E'.i = T.node\}$
$E' \to +TE'_1$	$\{E'.node = E'_1.node; E'_1.i = newNode(+, E'.i, T.node)\}$
$E' \to -TE'_1$	$\{E'.node = E'_1.node; E'_1.i = newNode(-, E'.i, T.node)\}$
$E' \to \varepsilon$	$\{E'.node = E'.i\}$
$T \to (E)$	$\{T.node = E.node\}$
$T \to id$	$\{T.node = newLeaf(id, id.entry)\}$
$T \to num$	$\{T.node = newLeaf(num, num.lexval)\}$

Nonostante alcune delle produzioni della grammatica non vengano utilizzate nel nostro caso, non vuol dire che non vi possano essere stringhe passate in input che ne possano usufruire: ricordiamoci che nonostante le strutture che stiamo utilizzando fanno riferimento ad un particolare esempio, è necessario che la grammatica sia generale, allo stesso modo devono essere generali anche le relative azioni semantiche.

Per questo motivo, possiamo osservare che la produzione $E' \to +TE'$ svolge lo stesso ruolo di quella che abbiamo già potuto analizzare mentre, $T \to (E)$, viene utilizzata per poter passare un riferimento al padre.

A questo punto abbiamo bisogno dell'*evaluation order*, ottenuto facendo l'ordinamento topologico del dependency graph; si tenga a mente che un qualsiasi ordinamento topologico del grafo può andare bene. Ricordiamo anche che non è detto che si riesca a trovare un ordinamento topologico: in questo caso non è possibile valutare il grafo ottenuto.

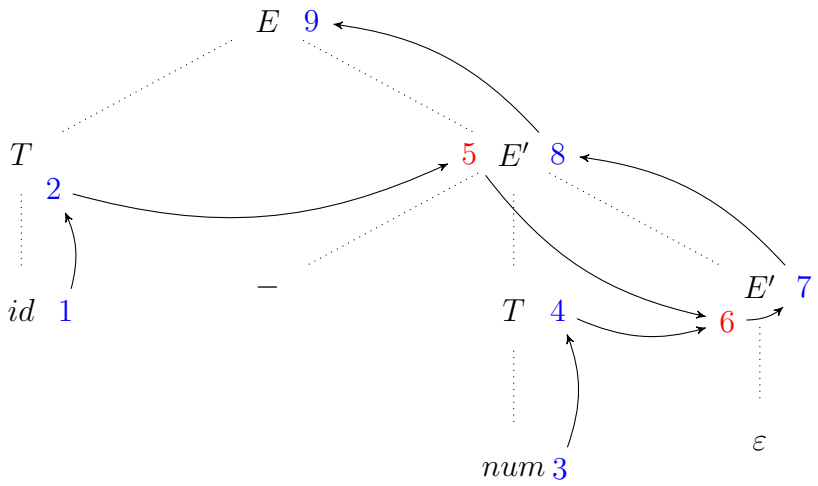


Figura 10.23: Dependency Graph attributi ereditati con Evaluation Order

Finalmente abbiamo la possibilità di osservare passo per passo come costruire il nostro abstract syntax tree seguendo l'evaluation order:

1. recuperiamo le informazioni relativamente ad id (i.e. a) dalla symbol table mediante il riferimento presente in $id.entry$;
2. generiamo una foglia con etichetta id e un riferimento ad a nella symbol table), tale operazione viene eseguita dalla regola $\{T.node = newLeaf(id, id.entry)\}$ per $T \to id$;
3. recuperiamo il valore di num ottenuto dall'analisi lessicale mediante $num.lexval$ (i.e. 4);
4. generiamo la foglia per il valore di num, tale operazione viene eseguita dalla regola $\{T.node = newLeaf(num, num.lexval)\}$ per $T \to num$;
5. creiamo un riferimento alla foglia di a utilizzando la regola $\{E'.i = T.node\}$ per $E \to TE'$. In questo passaggio dobbiamo recuperare $T.node$ (che contiene un riferimento alle informazioni del sottoalbero sinistro) e assegnarlo a $E'.i$, in modo che i passi successivi possano usufruirne;
6. generiamo il nodo per la sottrazione $a - 4$: tale operazione viene eseguita dalla regola $\{E'_1.i = newNode(-, E'.i, T.node)\}$ per $E' \to -TE'_1$; anche in questo caso abbiamo bisogno di passare il riferimento ai passi successivi;
7. creiamo un riferimento al nodo della sottrazione appena creato utilizzando la regola $\{E'.node = E'.i\}$ per $E' \to \varepsilon$;
8. creiamo un riferimento allo stesso nodo utilizzando la regola $\{E'.node = E'_1.node\}$ per $E' \to -TE'_1$;
9. infine creiamo un riferimento ancora allo steso nodo (esato, è proprio questo che si fa quando si ritorna da una ricorsione) utilizzando la regola $\{E.node = E'.node\}$ per $E \to TE'$.

Il risultato finale è dunque rappresentabile nel seguente modo:

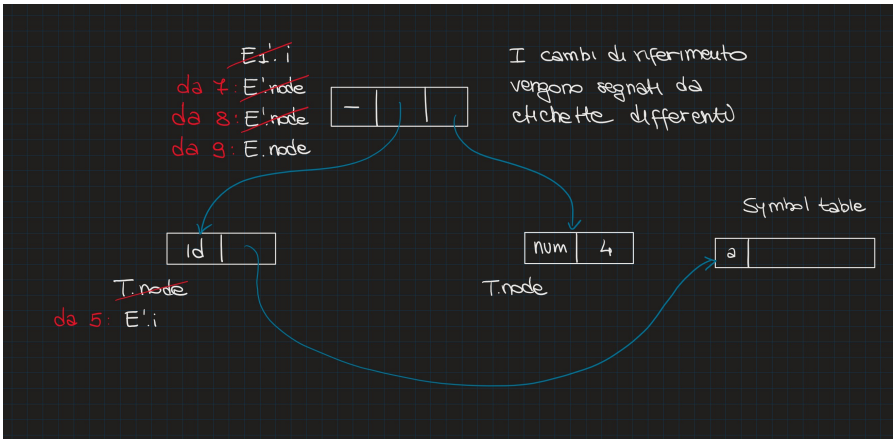


Figura 10.24: Abstract syntax tree attributi ereditati

Es attribuzione tipo degli identificatori

Consideriamo la seguente grammatica

$$
\begin{aligned}
D &\to TL \\
T &\to int \\
T &\to float \\
L &\to L_1, id \\
L &\to id
\end{aligned}
$$

L'obbiettivo è quello di associare alle produzioni descritte delle regole per cui si possa aggiungere il tipo nella entry della symbol table degli identificatori dichiarati. Per poter aggionrare la tabella è necessario utilizzare la funzione `addtype(table_entry, type_instance)`.

Anche in questo caso partiamo da un esempio che può aiutarci per la definizione delle regole: supponiamo di avere la parola *int id, id*, in questo caso per una derivazione leftomost i passi di derivazione che dobbiamo compiere sono

$$D \Rightarrow TL \Rightarrow int\ L \Rightarrow int\ L, id \Rightarrow int\ id, id$$

dunque il parse tree può essere così rappresentato

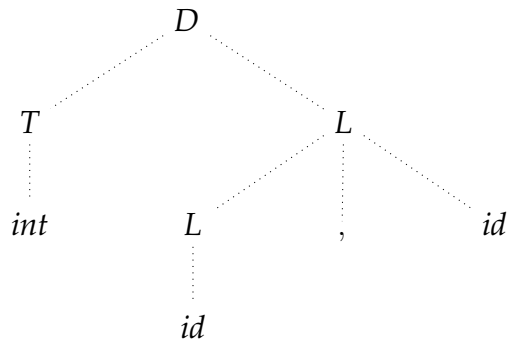


Figura 10.25: Parse tree aggiornamento tipo identificatori

Partiamo dalle produzioni più semplici, cioè quelle che si occupano di definire il tipo degli attributi: in questo caso banalmente possiamo passare l'attributo sintetizzato al driver della produzione

$$\begin{aligned} T &\rightarrow int & \{T.type = int\} \\ T &\rightarrow float & \{T.type = float\} \end{aligned}$$

Guardando il parse tree osserviamo che la risoluzione del sottoalbero destro dipende dal sottoalbero sinistro: una volta scoperto il tipo delle nostre entry abbiamo bisogno di fornire questa informazione alla funzione nell'altro sottoalbero, ma ciò è possibile solamente utilizzando degli attributi ereditati. Per questo motivo possiamo introdurre la seguente regola

$$D \rightarrow TL \qquad \{L.i = T.type\}$$

Una volta che abbiamo trovato il modo per poterci trasportare il tipo dobbiamo effettivamente utilizzarlo per le nostre entry: a questo scopo possiamo andare a valorizzare le regole per le produzioni rimaste nel seguente modo

$$\begin{aligned} L &\rightarrow L_1, id & \{addtype(id.entry, L.i);\ L_1.i = L.i\} \\ L &\rightarrow id & \{addtype(id.entry, L.i)\} \end{aligned}$$

Come è possibile notare, entrambe le produzioni le entry della tabella aggiungendo il tipo; nel primo caso viene aggiunta una regola per garantire la

produzione degli attributi per i sottoalberi successivi. Riassumendo, le regole per la tipizzazione sono le seguenti:

$$
\begin{array}{ll}
D \to TL & \{L.i = T.type\} \\
T \to int & \{T.type = int\} \\
T \to float & \{T.type = float\} \\
L \to L_1, id & \{addtype(id.entry, L.i);\ L_1.i = L.i\} \\
L \to id & \{addtype(id.entry, L.i)\}
\end{array}
$$

Parte III

Aftermath

Capitolo 11

La generazione del codice intermedio

11.1 Riepilogo sulla fase frontend della compilazione

Durante il nostro percorso abbiamo avuto modo di attraversare quasi tutte le fasi che costituiscono il cosiddetto *frontend* della compilazione:

- l'analisi lessicale ha il compito di distinguere i vari lessemi in un dato input e successivamente restituire una stringa di tokens, ossia di elementi che possono essere correttamente letti da un analizzatore sintattico; inoltre, ha il compito di iniziare a popolare la tabella dei simboli;
- l'analisi sintattica, su cui non ci dilunghiamo troppo in questa sede dal momento che costituisce circa $\frac{1}{3}$ di questo elaborato già da sola, verifica se la stringa ottenuta dall'analisi lessicale appartiene al linguaggio di una data grammatica (restituendo in caso positivo l'albero di derivazione); è interessante specificare che, in molti casi, a questa fase vengono accorpate ulteriori operazioni, come ad esempio la stessa analisi semantica;
- l'analisi semantica, che si occupa di fare controlli statici, ad esempio circa la compatibilità di operandi e operatori nei vari statements (è qui che si scoprono errori come tentativi di somma tra un `int` e un `float` o tentativi di conversione di tipo quando si fa uso di un operatore che consente operazioni tra tipi misti), o anche altri riguardanti la validità di istruzioni condizionali e di controllo (`if`, `while` e `break`).

A valle di tutto questo troviamo l'ultima fase del frontend della compilazione, la generazione del codice intermedio; al pari dell'analisi semantica, è possibile che anche quest'ultima fase venga implementata come parte integrante dell'analisi sintattica, e in particolare ci sono degli schemi di traduzione che vanno a sfruttare gli alberi di sintassi astratta (AST) che abbiamo visto quando abbiamo parlato di analisi semantica.

11.2 Il codice intermedio

Il codice intermedio è una rappresentazione del nostro codice di partenza, abbastanza astratta da nascondere alcune specifiche che sono tipiche del codice macchina e che ne rendono la lettura poco agevole (come ad esempio i movimenti di informazioni tra registri o tra memoria e registri), ma comunque più specifica del linguaggio originale da cui siamo partiti.

11.2.1 Rappresentazione del codice intermedio

Non abbiamo una sola possibilità per rappresentare il codice intermedio; a seguito presenteremo quelle più comunemente utilizzate.

Strutture a grafo Una prima via è quella di utilizzare delle strutture a grafo partendo dai parse trees, come ad esempio gli stessi AST o dei grafi diretti aciclici (DAG). Questi ultimi sono spesso più succinti degli AST: immaginiamo che un AST abbia due foglie con uno stesso identificatore a: in un DAG avremmo lo stesso nodo con più archi entranti. Gli AST sono a loro volta più succinti dei parse trees: questi spesso devono ricorrere a cammini molto lunghi per rappresentare certe derivazioni, rallentando di molto le operazioni che ci troveremo a fare in seguito.

Codice a tre indirizzi Una possibilità, leggermente migliore, è anche quella di utilizzare il cosiddetto *codice a tre indirizzi*, così chiamato perché non prevede la possibilità di referenziare più di tre valori in un singolo statement (che quindi avrà forma `x = y op z`), limitando enormemente la complessità degli stessi.

Codice in un altro linguaggio Una terza strada percorribile è quella di utilizzare un altro linguaggio di programmazione come vero e proprio codice intermedio. Questo approccio ci risparmia inoltre il problema di implementare da noi un backend per la compilazione, perché da questo momento in poi possiamo tranquillamente utilizzare la catena di compilazione del linguaggio che stiamo usando come codice intermedio. In questi casi la scelta favorita è il C, perché è un linguaggio incredibilmente flessibile e dispone dell'eccellente compilatore `gcc`.

11.2.2 Il codice a tre indirizzi

In un certo senso, il codice a tre indirizzi è una rappresentazione testuale di un AST. Per capire di cosa stiamo parlando, è utile andare subito a vedere un

esempio di traduzione da AST a codice a tre indirizzi. Consideriamo il seguente AST:

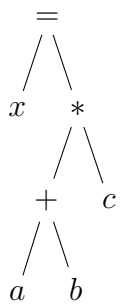


Figura 11.1: AST per l'espressione `x = (a + b) * c`

In questo caso abbiamo l'AST per una banale espressione aritmetica (con un minimo di parentesizzazione), abbastanza semplice da ricostruire partendo dalle foglie dei livelli più bassi. L'equivalente rappresentazione dell'AST nel codice a tre indirizzi può essere ottenuta salvando l'esito delle operazione intermedie in dei registri temporanei, come segue:

$$t_1 = a + b$$
$$t_2 = t_1 * c$$
$$x = t_2$$

Avere una rappresentazione testuale di un AST, facendo uso del codice a tre indirizzi, è comodo perché semplifica la generazione di linguaggio macchina e la sua ottimizzazione; tuttavia questa tecnica non è molto frequente, perché è dipendente dalla macchina per cui stiamo scrivendo il backend (machine dependent).

11.2.3 Esempio di generazione del codice intermedio

Supponiamo di partire con un'espressione come questa[1]:

```
if ((x < 100) || (x > 200 && x != y)) x = 0;
```

Una prima approssimazione di codice intermedio che possiamo ottenere è questa:

[1]Originariamente l'espressione era stata scritta come `if (x < 100 || x > 200 && x != y) x = 0;`, ma noi abbiamo deciso di aggiungere quelle parentesi perché è così che è stata effettivamente valutata nella traduzione, almeno se applichiamo il corretto ordine di precedenza degli operatori logici (`&&` $\preceq$ `||`).

```
    if x < 100 goto L2
    goto L3
L3: if x > 200 goto L4
    goto L1
L4: if x!= y goto L2
    goto L1
L2: x = 0
L1:
```

Possiamo osservare come in questa traduzione siano state sfruttate le regole di cortocircuitazione: se infatti `x < 100` viene valutata come vera, allora si esce subito dal controllo. Tuttavia, in quella traduzione abbiamo numerosi `goto` ridondanti; ci permettiamo dunque di presentare una traduzione leggermente migliore:

```
    if x < 100 goto L2
    iffalse x > 200 goto L1
    iffalse x!= y goto L1
L2: x = 0
L1:
```

il controllo sulla falsità di `x` ci permette di ridurre il numero di righe e labels utilizzate, nonché sfruttare meglio la corticircuitazione, caratteristica fondamentale per un buon compilatore. Si tenga sempre a mente che non sempre le condizioni vengono valutate nello stesso esatto ordine in cui sono state scritte dal programmatore.

11.3 Statement per il controllo di flusso

A questo punto possiamo andare a vedere più da vicino la traduzione in codice intermedio di alcuni statements. In particolare ci interessano i seguenti:

$$
\begin{aligned}
&P \to S \\
&S \to if(B)S_1 \\
&S \to while(B)S_1 \\
&\quad \dots \\
&B \to \texttt{true} \\
&B \to \texttt{false} \\
&B \to B_1 \,||\, B_2 \\
&B \to B_1 \,\&\&\, B_2
\end{aligned}
$$

In questo schema P sta per *program*, S sta per *statement* e B sta per *boolean*.

11.3.1 Istruzioni condizionali

Diamo un'occhiata alla generazione del codice intermedio di un'istruzione condizionale come potrebbe essere un comune costrutto `if-then`. Possiamo schematizzarne la struttura in questo modo:

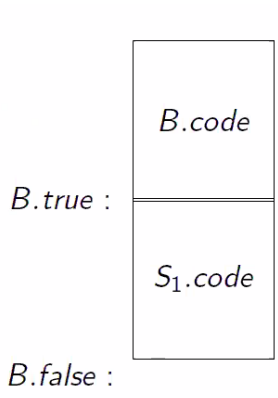


Figura 11.2

dove l'etichetta $B.true$ punta al body dell'istruzione condizionale (perché naturalmente vuol dire che la condizione è stata verificata), mentre l'etichetta $B.false$ punta alla prima istruzione successiva alla chiusura del blocco `then`.

Attributi Andiamo a vedere di preciso quali attributi abbiamo in questo processo:

- $S.next$, attributo ereditato: ci dice qual è la posizione della primissima istruzione da eseguire dopo aver concluso l'esecuzione del blocco di codice S; conservare l'etichetta è utile, perché S potrebbe avere degli altri statements di controllo annidati (ad esempio un ciclo `while` o una catena di `if`), per cui è importante poter sempre sapere qual è l'istruzione che deve essere eseguita dopo lo statement padre;
- $S.code$, attributo sintetizzato: è il codice intermedio che implementa lo statement che stiamo considerando; termina con un istruzione di salto a $S.next$;
- $B.true$, attributo ereditato: questa etichetta punta alla prima istruzione da eseguire se la condizione di B risulta vera;
- $B.false$, attributo ereditato: per analogia a $B.true$, questo attributo punta alla prima istruzione da eseguire nel caso in cui B sia valutato come falso;

- $B.code$, attributo ereditato per gli statements, sintetizzato se guardiamo solo l'SDD delle condizioni booleane: è la sequenza dei passi di codice intermedio che implementano la condizione booleana e determinano se passare a $B.true$ o $B.false$.

Nota sui booleani In questo è bene puntualizzare un distinguo sui due diversi ruoli che possono assumere i booleani:

- alterare il flusso di esecuzione, e quindi operare come *condizioni booleane*;
- calcolare il valore di espressioni logiche, e quindi comportarsi come *espressioni booleane* (e quindi come r-values).

Quella che noi consideriamo oggi è la prima funzione; nel secondo caso si comportano esattamente come delle normali espressioni aritmetiche, che lavorano con valori e operatori logici anziché numerici. Molto spesso le grammatiche si trovano ad avere non-terminali diversi per questi due ruoli.

11.3.2 Il programma come statement

Andiamo ora a vedere da vicino la prima istruzione dalla nostra lista, $P \to S$; ne approfitteremo per dare un'occhiata anche a quale tipo di notazioni e regole di sintassi (più o meno) utilizzeremo in questo processo.

$$P \to S \qquad \begin{aligned} S.next &= newlabel() \\ P.code &= S.code \triangleright label(S.next) \end{aligned}$$

Il senso stesso dell'espressione $P \to S$ sembra suggerire che il programma P possa essere visto globalmente come uno statement.

- $newlabel()$ è una funzione che, una volta invocata, genera una nuova etichetta;
- $\triangleright$ è un opereatore che denota la concatezione di due (o più) segmenti di codice intermedio;
- $label(L)$ assegna l'etichetta L alla prossima istruzione a tre indirizzi generata;
- $S.next$, come intuibile, è l'etichetta che denota la prima istruzione da eseguire dopo la fine di S; se ipoteticamente volessimo uscire da S prima della chiamata a $S.next$, allora dovremmo esplicitamente eseguire un `goto`.

Inoltre possiamo vedere come l'attributo sintetizzato $P.code$ venga definito con l'ausilio dell'operatore $\triangleright$.

11.3.3 Semplice blocco `if-then`

Vediamo adesso nello specifico come tradurre uno statement S che si compone di un'istruzione di `if-then` semplice, senza un blocco `else`.

$$S \to if(B)S_1 \qquad \begin{aligned} B.true &= newlabel() \\ B.false &= S_1.next = S.next \\ S.code &= B.code \triangleright label(B.true) \triangleright S_1.code \end{aligned}$$

La prima cosa che facciamo è creare una nuova label $B.true$ che etichetti l'inizio del blocco di codice da eseguire nel caso in cui la condizione B sia verificata. Nel caso in cui B risulti invece falsa, assegnamo all'etichetta di $B.false$ il blocco successivo ad S_1 (quindi successivo al blocco `then`); dal momento che in questo caso non abbiamo anche un `else`, il codice successivo a S_1 sarà quello successivo a tutto lo statement S.

Complessivamente, il valore che assegnamo al nostro statement S sarà quindi costituito dal codice della condizione booleana $B.code$, l'etichetta $B.true$ e il blocco di codice da eseguire nel caso in cui l'istruzione sia verificata ($S_1.code$). Nel caso in cui la condizione booleana non fosse valutata come vera, il programma eseguirebbe, come detto in precedenza, la prima istruzione contenuta in $B.false$.

11.3.4 Blocco `if-then-else`

Supponiamo di voler tradurre un'istruzione condizionale che comprende anche un blocco `else`. Questo naturalmente cambia leggermente la traduzione, poiché l'etichetta $B.false$ non punterà più alla prima istruzione successiva al blocco `then`, bensì dovrà essere creata ex novo, e punterà alla prima istruzione del blocco `else`.

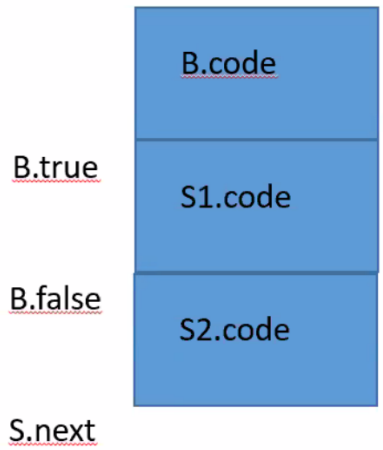


Figura 11.3

Entrambe le etichette $S_1.next$ e $S_2.next$ punteranno all'istruzione successiva all'intero statement.

$$
\begin{aligned}
S \rightarrow if(B)S_1 else S_2 \quad B.false &= newlabel() \\
B.true &= newlabel() \\
S_1.next &= S.next \\
S_2.next &= S.next \\
S.code &= B.code \triangleright label(B.true) \triangleright S_1.code \triangleright \\
&\quad gen(goto\ S.next) \triangleright label(B.false) \triangleright S_2.code
\end{aligned}
$$

Il codice intermedio dello statement S è dato quindi dalla concatenazione di tutti questi blocchi; particolare riguardo all'istruzione $gen(goto\ S.next)$, che deve essere eplicitata per poter saltare all'istruzione contenuta in $S.next$.

11.3.5 Traduzione di un ciclo `while`

Prendiamo adesso visione del codice intermedio di un ciclo.

- La prima cosa che facciamo è assegnare due $newlabels()$: una la chiameremo $begin$, e punterà all'inizio del ciclo (quindi all'espressione $B.code$); l'altra $B.true$, e indicherà l'inizio del corpo del ciclo S_1 e a cui andiamo quindi in caso l'espressione contenuta in $B.code$ sia verificata.

- Nelle istruzioni condizionali viste prima ci siamo abituati ad vedere $S_1.next$ puntare a $S.next$, poiché al termine del corpo di un'istruzione condizionale normalmente proseguiamo con l'esecuzione del codice; in questo caso, invece, al termine del corpo dell'istruzione dovremmo tornare ad eseguire il controllo $B.code$ e, eventualmente, rieseguire il corpo del ciclo stesso, per cui $S_1.next$ dovrà puntare a $begin$.

A questo punto, la composizione dell'attributo sintetizzato $S.code$ dovrebbe seguire naturalmente.

$$
\begin{aligned}
S \rightarrow while(B)S_1 \qquad begin &= newlabel() \\
B.true &= newlabel() \\
B.false &= S.next \\
S_1.next &= begin \\
S.code &= label(begin) \triangleright B.code \triangleright label(B.true) \triangleright \\
&\quad S_1.code \triangleright gen(goto\ begin)
\end{aligned}
$$

11.3.6 Traduzione di operazioni booleane

Per concludere, diamo un'occhiata rapida alla traduzione delle espressioni logiche. Lo faremo andando a confrontare la diversa traduzione di due semplici operazioni di `OR` e `AND`:

$$B \rightarrow B_1 \,||\, B_2 \qquad\qquad B \rightarrow B_1 \,\&\&\, B_2$$

Andiamo a confrontare il codice intermedio di queste due espressioni, riga per riga, in modo da apprezzarne le analogie e il modo in cui sfruttano la cortocircuitazione:

- l'etichetta $B_1.true$ rimanda direttamente all'etichetta $B.true$ dell'intero statement nel caso dell'`OR` (perché naturalmente è sufficiente che un elemento sia $true$ per rendere valida l'intera espressione), mentre invece per `AND` viene semplicemente creata una nuova label;

- la situazione è assolutamente analoga, ma a parti invertite, per l'etichetta $B_1.false$;

- nelle due righe successive andiamo a definire le etichette per il secondo elemento delle nsotre espressioni e, in entrambi i casi, queste ultime punteranno alle etichette di uscita dell'intero statement; questo accade perché sono l'ultimo elemento dell'espressione ed il loro valore andrà a determinare quello dell'intera espressione;

- la differenza nella composizione di $B.code$ sta quindi esclusivamente nell'etichetta che si frappone fra il codice del primo e del secondo elemento: nel caso dell'`OR`, infatti, il secondo elemento verrà valutato solo se il primo risulta falso (e qui infatti mettiamo la nuova etichetta $B_1.false$); viceversa nell'`AND` valutiamo il secondo solo se il primo è vero, e quindi l'etichetta frapposta è $B_2.false$.

$$
\begin{aligned}
B_1.true &= B.true \\
B_1.false &= newlabel() \\
B_2.true &= B.true \\
B_2.false &= B.false \\
B.code &= B_1.code \triangleright \\
&\quad label(B_1.false) \triangleright B_2.code
\end{aligned}
\qquad
\begin{aligned}
B_1.true &= newlabel() \\
B_1.false &= B.false \\
B_2.true &= B.true \\
B_2.false &= B.false \\
B.code &= B_1.code \triangleright \\
&\quad label(B_1.true) \triangleright B_2.code
\end{aligned}
$$

Una precisazione

Se a questo punto avete l'impressione che tutto questo capitolo sia molto sconnesso e senza una conclusione (o in realtà del vero e proprio contenuto)... beh, sappiate che è la stessa sensazione che ha provato l'autore.

Tutta la squadra ha avuto l'impressione che l'argomento non sia stato per nulla approfondito e in realtà abbiamo il sospetto che la professoressa sia una grande fan di Speedy Gonzales, ma... Hey, questo è quello che passa il convento e pare che questi argomenti nemmeno siano presenti all'esame. Mai dire mai però, quindi occhi aperti e guardia alta.

Capitolo 12

Sintesi e conclusioni

In questo ultimo capitolo ripercorreremo sommariamente tutte le tappe che abbiamo attraversato nel nostro cammino verso la padronanza della compilazione frontend; l'obiettivo è fornire una visione d'insieme di tutto il nostro percorso, dal momento che quando si approfondisce un argomento è molto facile perdere il senso di come questo si relazioni a tutti gli altri.

Inoltre approfitteremo di questo per presentare dei piccoli approfondimenti trasversali, curiosità, divagazioni un po' casuali, barzellette e anche qualche battutaccia, il tutto per mantenere vivo l'interesse dello stanchissimo lettore che ha attraversato pianure, mari, monti, la città sotterranea di Agharti, il regno di Prete Gianni, l'ultima Thule e l'isola di Mu, tutto durante queste 300 pagine, il tutto viaggiando in un camioncino della verdura, il tutto per imparare qualcosa su quelle strane bestie che sono i linguaggi formali e i compilatori.

Partiamo quindi con il ripetere, per l'ennessima volta, in maniera schematica le fasi della compilazione ed il loro scopo:

Analisi lessicale In questa fase si prende la stringa in ingresso e la si trasforma in una sequenza di token, che diamo in pasto all'analizzatore sintattico.

Analisi sintattica A questo punto verifichiamo (attraverso il parsing) se la sequenza di token aderisce alla specifica grammatica del linguaggio di programmazione che stiamo utilizzando, se è così ricaviamo un albero di parsing per la stringa data in input.

Analisi semantica Qui vengono considerate caratteristiche del linguaggio che non possono essere descritte agilmente dalla grammatica; inoltre, si va a valorizzare e ad assegnare attributi a tutti gli elementi riconosciuti nelle fasi precedenti.

Generazione del codice intermedio In questa fase si genera, come suggerito dal nome, una rappresentazione intermedia della stringa data in input e si va a compiere possibili ottimizzazioni (molte delle quali indipendenti dal linguaggio finale, target code).

Generazione del codice macchina A questo punto si traduce il codice intermedio in linguaggio macchina con l'eventuale aggiunta di ottimizzazioni legate all'architettura su cui si sta operando.

Noi abbiamo visto tutte queste fasi separatamente, ma ricordiamo che spesso e volentieri vengono eseguite in simultanea per ottimizzare i tempi. È arrivato il momento di rispettare le tradizioni, quindi ora introdurremo un esempio che ci permetterà di andare a rianalizzare più approfonditamente tutte le fasi della compilazione.

12.1 Analisi lessicale

$$position = initial + rate * 60 \tag{12.1}$$

Data in input la stringa rappresentata in 12.1, l'analizzatore lessicale si occupa di ricavare i token ed inserire tutte le informazioni nella symbol table, ottenendo un risultato simile a quello che si può osservare in seguito. Riportiamo la lista di identificatori.

$$< id, 1 > assign < id, 2 > sumop < id, 3 > mulop < num, 60 > \tag{12.2}$$

Riportiamo ora la tabella dei simboli (Tab.12.1).

Reference	Name	Other attributes
1	position	...
2	initial	...
3	rate	...

Tabella 12.1: Symbol table ricavata dall'analisi lessicale

In questo caso specifico, l'analizzatore lessicale va a riconoscere $position$, $initial$ e $rate$ come identificatori (nota che nella lista di token, Eq.12.2, sono presenti i riferimenti alle entry della symbol table per gli identificatori); allo stesso modo, vengono riconosciuti gli altri terminali come gli operatori utilizzati e, in questo caso particolare, il numero (creando un token che ne specifica il terminale, cioè *num*, e il valore).

La stringa che andremo ad analizzare nell'analisi sintattica si dimenticherà poi dei riferimenti agli identificatori, delle parentesi angolari e dei valori numerici, quindi sostanzialmente avrà questa forma:

$$id = id + id * num \tag{12.3}$$

L'analizzatore lessicale deve saper riconoscere le parole chiave del linguaggio, differenziandole da eventuali identificatori ambigui (ad esempio $while22$); la strategia più utilizzata per risolvere tale problematica è quella del *longest match*: si assegna la categoria di token che identifica la versione più lunga possibile della stringa che si riesce a matchare.

Per svolgere il loro compito, gli analizzatori lessicali utilizzano automi a stati finiti, sia in versione deterministica che non deterministica. In qualsiasi caso, i tipi di linguaggi che gli analizzatori lessicali vanno ad approcciare sono tutti linguaggi regolari, derivati quindi da grammatiche regolari.

Di questa fase, durante il nostro percorso, abbiamo visto tutti i passaggi: abbiamo ricavato l'automa per l'analisi lessicale da una determinata grammatica, poi abbiamo visto come minimizzare tale automa ed infine il suo utilizzo per riconoscere un linguaggio regolare. Ora possiamo quindi passare in input all'analizzatore sintattico la lista di token che abbiamo ricavato e gustarci il prossimo flashback (*vietnam intensifies*).

12.2 Analisi sintattica

Nella fase di analisi sintattica vogliamo innanzitutto capire se la lista di token in ingresso aderisce alle regole della grammatica; da tale lista si ottiene, nel caso la stringa sia corretta, un albero di derivazione (o un albero di sintassi astratta, se sei proprio uno skillato). Nel nostro esempio, quello che andiamo ad ottenere dalla lista di token in Eq.12.2 è il seguente albero di parsing.

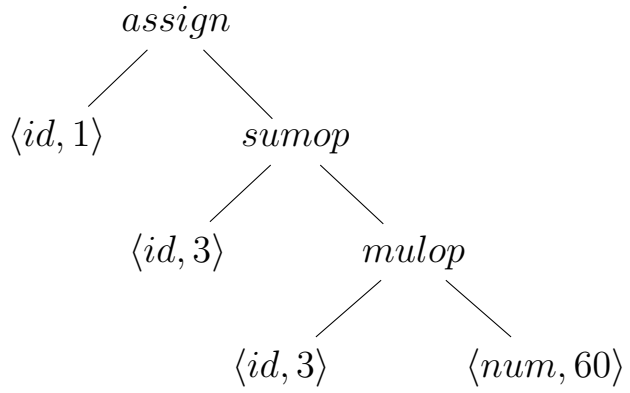


Figura 12.1: Parsing tree ricavato dal nostro esempio

Relativamente all'analisi sintattica abbiamo studiato ed utilizzato le due famiglie principali di metodi di parsing (basate su derivazioni leftmost in un caso e rightmost nell'altro). È da sottolineare però, che esistono parser adatti all'analisi di una qualsiasi grammatica libera, tuttavia questi hanno complessità minima n^3 (con n lungehezza dell'input) e questo li esclude dalla nostra wishlist. Non dimentichiamoci però che, concentrandoci principalmente sulle grammatiche regolari, i parser che abbiamo esaminato in questo corso riescono a ricavare un albero di derivazione con complessità *LINEAREH*.

Le due, ormai ben note, tipologie principali di parsing, sono:

- parsing top-down;
- parsing bottom-up.

12.2.1 Parsing top-down
Quello bello

Quando ci troviamo al cospetto di grammatiche $LL(1)$, riusciamo a ricavare il parse tree senza bisogno di backtrack utilizzando il parsing top-down. La strategia di analisi del parsing top-down prevede di partire dalla derivazione dello start symbol e poi, utilizzando la derivazione di tipo leftmost, costruire il parse tree. Se la grammatica è $LL(1)$ va tutto liscio ed otteniamo l'albero di parsing in men che non si dica, se non lo è dobbiamo ricorrere al backtrack, ma noi non vogliamo che questo succeda, Larry (se nemmeno voi, come il revisore, avete capito... sorridete e annuite: non si contraddice mai un pazzo).

Abbiamo quindi visto delle caratteristiche che vogliamo evitare per assicurarci di lavorare sempre con grammatiche $LL(1)$:

- Le grammatiche *ricorsive sinistre* **non** sono $LL(1)$, abbiamo visto però come si può, in alcuni casi, trasformarle per renderle tali;

- Le grammatiche *fattorizzabili a sinistra* **non** sono $LL(1)$, ma come per il punto precedente aabbiamo visto come è possibile provare ad eliminare questo problema;
- Le grammatiche *ambigue* **non** sono $LL(1)$, perché esiste almeno una stringa nel linguaggio che può essere derivata in 2 modi diversi (entrambi leftmost).

12.2.2 Parsing bottom-up
Quello tosto

L'altra grossa famigliola di grammatiche che abbiamo visto contiene quelle grammatiche che possono essere analizzate tramite il parsing bottom-up, di questa famiglia siamo andati ad indagare queste tipologie di grammatiche:

- SLR
- $LR(1)$
- $LALR$

Il goal dell'analisi bottom-up è sempre lo stesso, ma il modo in cui si va a ricostruire l'albero di derivazione è opposto al parsing top-down: si parte a ricostruire l'albero delle derivazioni dalle foglie e si continua fino alla radice.

Indifferentemente dal tipo di grammatica che stiamo analizzando, il modo per ricavare il parse tree prevede sempre l'utilizzo dell'*algoritmo shift-reduce*. L'algoritmo di shift-reduce è anche'esso indipendente dal tipo di grammatica che si va ad utilizzare, tuttavia necessita della parsing table per quella particolare tipologia di grammatica.

Mentre eseguiamo l'algoritmo andiamo ad analizzare la lista di token in input: controllando token per token ed usando la parsing table come mappa, ci spostiamo nel buffer di lettura (mosse di shift) e, quando si incontra una stringa che può essere derivata da una certa produzione della grammatica, si va ad effettuare una mossa di reduce.

Se non vi sono entry multiple defined possiamo terminare l'agloritmo con successo ed ottenere alla fine il parsing tree che tanto desideriamo.

Le 3 grammatiche viste per il parsing bottom-up, si differenziano per la precisione con cui si decide in quali caselle della tabella di parsing va inserita una determinata riduzione.

La differenza tra un tipo di grammatica e l'altro è la seguente:

- utilizzando grammatiche e parsing SLR otteniamo un automa semplice e di veloce computazione, ma povero di informazioni (rischiamo di avere dei conflitti)

- d'altro canto, il parsing $LR(1)$ ci fornisce degli automi molto più precisi, con dei lookahead set che ci danno molte più informazioni su quando applicare le riduzioni, ma che ci potrebbero portare ad una sovrabbondanza che porterà via molto tempo alla loro computazione (e anche spazio)
- tra SLR e $LR(1)$ troviamo grammatiche $LALR$, che ci offrono una tabella di parsing delle dimensioni di una tabella SLR, ma hanno una complessità di calcolo leggermente superiore alle grammatiche SLR permettendoci di avere gli stessi lookahead set (quindi la stessa precisione) del parsing $LR(1)$.

In realtà le possibilità non terminano qui, infatti si potrebbero ottenere delle grammatiche molto più semplici da parsare aumentantdo il numero di informazioni che si usano prima delle riduzioni, sostanzialmente incrementando il lookahead set: tali grammatiche potrebbero ad esempio essere $LL(2)$ ed $LR(2)$, ma la complessità per l'analisi della lista di token aumenterebbe parecchio in tal caso. La scelta privilegiata è quindi sforzarsi di trovare una grammatica $LL(1)$ o $LR(1)$ prima, per avere poi performance migliori.

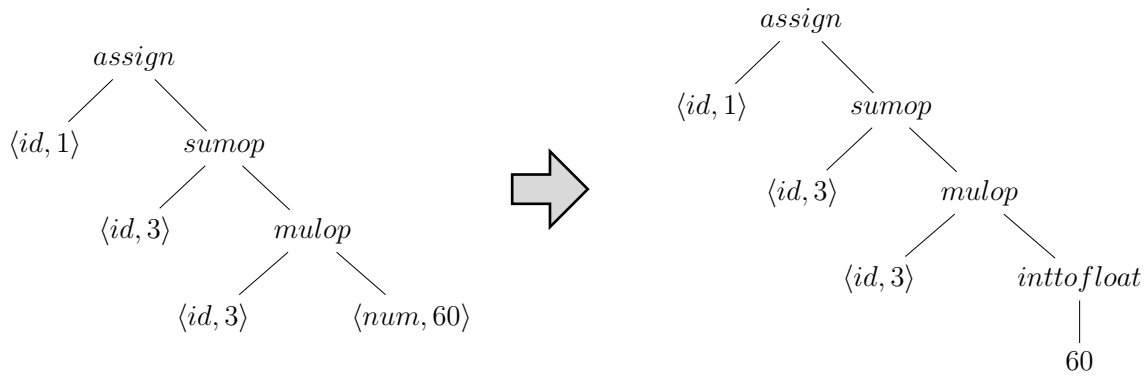
12.2.3 Quando finisce il parsing

Se l'analisi sintattica dimostra che la stringa non è riconosciuta dal linguaggio, vi è la possibilità, nei compilatori moderni, di prevedere dei meccanismi di riconoscimento degli errori: spesso sono i compilatori stessi che suggeriscono all'utente dove si potrerebbero trovare gli errori sintattici all'interno del codice.

In caso l'analisi sia andata a buon fine, si ottiene un parse tree che indica quali riduzioni sono state effettuate ed in quale ordine.

12.3 Analisi semantica

Questa è la fase in cui si va a valorizzare la sequenza di token che si ricava dall'analisi lessicale; nel nostro caso quello che otterremo dall'analisi semantica ha questa forma:



Symbol table

1	*position*	float	.....
2	*initial*	float	.....
3	*rate*	float	.....

Figura 12.2: Output dell'analisi semantica

Si noti che il $< num, 60 >$ è stato trasformato in `inttofloat` con nodo figlio 60: questo è avvenuto perchè l'analisi semantica è quella che si occupa di attribuire un valore (e di conseguenza anche un tipo) ai token. Proprio per questo motivo, gli effetti dell'analisi semantica non si vedranno solo nell'albero, come mostrato in Fig.**??**, ma anche nella tabella dei simboli che riportiamo qui:

Reference	Name	Type	Other attributes
1	position	float	...
2	initial	float	...
3	rate	float	...

Tabella 12.2: Symbol table aggiornata dall'analisi semantica

Si può immaginare, facendo qualche semplificazione, che da qualche parte nel codice, precedente a quella che stiamo analizzando noi nel nostro esempio (che ricordiamo è scritta in Eq.12.1), avessimo dichiarato le variabili: quando l'analisi semantica raggiunge tale dichiarazione effettua una tipizzazione degli identificatori, che è appunto riportata nella tabella appena mostrata.

In seguito alla tipizzazione degli identificatori, l'analizzatore semantico, vedendo che $rate$ è di tipo float, capisce che la moltiplicazione $rate * 60$ deve essere di tipo float e converte 60 con l'operazione `inttofloat`.

Aggiungiamo inoltre che la tipizzazione che viene effettuata in questa fase, serve anche a calcolare gli offset necessari per la memorizzazione delle variabili

che verrà effettuata al momento dell'esecuzione del programma: proprio grazie a questa manovra il compilatore riesce a descrivere che al tal programma servirà una quantità di memoria x per memorizzare una variabile di tipo y.

A questo punto si passa alla generazione del codice intermedio.

12.4 Generazione del codice intermedio

Un modo molto semplice per ottenere la generazione del codice intermedio consiste nell'attraversare l'albero sintattico, associando un temporaneo per ogni nodo intermedio. Si può osservare l'utilizzo di questa strategia applicata al nostro esempio: riportiamo dunque di seguito il codice intermedio generato partendo dal risultato dall'analisi semantica (Fig.**??**).

$$\begin{aligned} t1 &= inttofloat(60) \\ t2 &= id3 * t1 \\ t3 &= id2 + t2 \\ id1 &= t3 \end{aligned}$$

Una cosa a cui prestare attenzione è il fatto che gli identificatori `id1`, `id2` e `id3` sono riferimenti alle istanze locali degli identificatori: ciò vuol dire che si entra già nel territorio in cui si necessita di un meccanismo di *scope*, ma di questo parleremo in seguito.

A questo punto si passa all'ottimizzazione del codice intermedio, fase che ha le sue basi teoriche in solidissime e sofisticate prove matematiche.

12.5 Ottimizzazione del codice intermedio

Nell'esempio che stiamo trattando si riesce ad ottimizzare il codice riducendo subito il 60 in float ed eliminando operazioni temporanee non strettamente necessarie. Vediamo subito come risulterebbe un'ottimizzazione fatta secondo queste prerogative:

$$\begin{aligned} t1 &= id3 * 60.0 \\ id1 &= id2 + t2 \end{aligned}$$

Questa fase di ottimizzazione sfrutta algoritmi su grafo facendo un'analisi di catene di *definition use*: tenendo conto del momento della dichiarazione e

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF R2, id3
MULF R2, R2, 60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

Tabella 12.3: Generazione codice target da codice intermedio ottimizzato

dell'utilizzo di una certa variabile, è possibile ad esempio decidere di eliminare o spostare tale dichiarazione.

Un esempio piuttosto comune è la *subexpression elimination*, che prevede di sintetizzare tutte le espressioni che vengono ripetute nel codice: se nel nostro programma andiamo a calcolare più volte l'espressione $\pi * e$, la subexpression elimination va a salvare in una variabile temporanea il risultato, così da evitare di calcolarlo più e più volte sprecando risorse.

Nota che tutte queste ottimizzazioni avvengono a tempo di compiliazione, nella fase statica, e sono di tipo conservativo. Questo significa che alcune occorrenze della stessa espressione possono essere eseguite in entrambi i rami di un else, quello che si fa in questo caso è assumere in modo conservativo che possano essere percorsi tutti i rami di tutte le condizioni.

In sostanza le ottimizzazioni che vengono effettuate in questa fase ottimizzano in modo cieco, senza guardare i possibili flussi di esecuzione della logica del programma.

Nota anche che nelle fasi successive tutte le variabili dovranno essere inserite nei vari registri: qui avverranno ulteriori ottimizzazioni, ad esempio legate ad un migliore utilizzo dei registri, che andranno ad utilizzare algoritmi di graph coloring e altre stregonerie simili.

A valle dell'ottimizzatore del codice intermedio abbiamo la generazione del codice target.

12.6 Generazione del codice target

A questo punto possiamo finalmente tradurre il codice intermedio in codice target. Riportiamo in seguito quello che otteniamo a seguito di questa fase dal nostro esempio.

Spiegamo qualche dettaglio: il suffisso `F` sta per float per sottolineare che tutte le operazioni coinvolgono dei float e, per questo motivo, è necessario ad esempio

trattarli in maniera distinta dagli int. Possiamo osservare che nella prima riga viene caricato il contenuto di `id3` come float nel registro `R2`, poi avviene una moltiplicazione tra float che coinvolge `R2` e 60; segue un'operazione di `load` di un altro float e così via.

L'ultima nota che aggiungiamo riguardo a questa fase è che, come già accennato precedentemente, anche qui intervengono meccanismi di ottimizzazione, in questo caso strettamente legati al codice macchina (e quindi alla macchina target).

12.7 Approfondimento sulla tabella dei simboli

Le tabelle dei simboli vengono utilizzate fin dell'analisi lessicale e vengono mantenute per l'intera durata del processo; si capisce bene che sono le strutture principali in un compilatore, seconde solo agli alberi di parsing.

Ma come sono implementate effettivamente?

Ci sono varie strategie in realtà, ma la più diffusa è sicuramente la struttura dati dizionario, che deve presentare queste operazioni:

- insert
- lookup
- delete

I dizionari possono essere ottenuti sia con l'utilizzo di liste che con alberi di ricerca, ma l'implementazione di gran lunga più diffusa si basa sulle hash table, che offrono un costo di gestione lineare.

In questo caso si ha il canonico array bucket, da cui è possibile accedere ad una lista di elementi che contengono tutte le entry (gli elementi della tabella dei simboli) per cui la funzione di hash restituisce lo stesso risultato.

Funzione di hash Com'è strutturata solitamente una funzione di hash per la tabella dei simboli di un compilatore? Questa funzione si occupa di trasformare ogni carattere in un intero non negativo, tipicamente sfruttando funzioni built-in, a cui applica ulteriori operazioni per poi arrivare all'hash.

Come si può definire una funzione di hash adeguata? Conoscendo il dominio di applicazione. Ad esempio, nel nostro caso si può notare che spesso nella scrittura dei programmi, il programmatore va a dare nomi molto simili alle variabili: questo significa che utilizzare il suffiso degli identificatori non è una buona idea perchè porterebbe a diverse collisioni.

Una scelta tipicamente buona è la seguente

$$(\sum_{i=1}^{n} \phi^{n-i} c_i) \; mod \; \texttt{size} \tag{12.4}$$

Utilizzando funzioni hash di questo tipo si possono evitare con buona probabililtà le problematiche legate alla scarsa fantasia del nostro amico programmatore.

Scope Ora che abbiamo discusso le metodologie di implementazione di una symbol table, andiamo ad analizzare una problematica con cui tutte devono fare i conti: lo *scoping*.

Le dichiarazioni di fatto possono essere di due tipi: locali o globali. Va quindi capito, nel caso vi siano più dichiarazioni di una variabile, a quale scope si riferiscano le sue occorrenze nel codice. Può essere utile immaginare un AST che rappresenta un certo programma in cui lo stesso nome è utilizzato sia per una variabile globale che per una variabile locale di una subroutine. In questo AST la variabile locale sarà in un sottoalbero dell'albero in cui è dichiarata la variabile globale; le dichiarazioni locali sono comprese in sottoalberi dell'AST.

Come si va a gestire la tabella dei simboli per rispettare lo scope delle dichiarazioni? Esistono due principali soluzioni per gestire queste dichiarazioni annidate:

- si usa un'unica symbol table e, nella lista a cui si accede dal bucket, si va a prendere il primo elemento con lo stesso nome di quello della variabile. In questo modo si è sicuri di ottenere la variabile con scope più vicino all'occorrenza che si sta analizzando, perché la entry che è stata inserita per ultima sarà la prima della lista del bucket; adottando questa strategia però si deve anche implementare un meccanismo di rimozione delle variabili di uno scope quando si esce da quest'ultimo;
- creiamo una tabella dei simboli diversa per ogni scope, in questo modo quando usciamo da uno scope ci basta cancellare il puntatore alla tabella dei simboli associata.

12.8 Note finali per progetti futuri

Il lettore deve sapere che esistono dei tool per la creazione di analizzatori sintattici, che richiedono venga fornito in input:

- una grammatica;

- la lista di token della grammatica;
- come questi token devono interagire con l'analizzatore lessicale.

Una volta che si danno questi elementi in pasto al tool, si ottiene un analizzatore sintattico bello che funzionante. Addirittura, spesso si trovano coppie di tool per la creazione di analizzatori sia lessicali che sintattici, come ad esempio la coppia di software Flex e Bison.

Ma non è finita qui.

In alcuni casi i tool di cui sopra accettano come input anche delle funzioni di attribuzione che ci permettono di specificare le azioni semantiche per le varie produzioni (tipicamente in qualche linguaggio come il C), il che ci permette di ottenere dei parser che compiono anche la valorizzazione semantica (grazie Bison).

Cosa significa tutto questo?

Mettiamo caso che ci inventiamo un linguaggio tutto nostro: se abbiamo un generatore di analizzatore sintattico ed un'implementazione di symbol table riusciamo a generare il nostro front-end del compilatore; poi siccome l'intermediate code può essere un qualsiasi linguaggio, possiamo scegliere il C come intermediate code e sfruttare il tool gcc come back-end del compilatore. Ecco spiegato come ottenere un compilatore tutto per noi in pochi semplici passi.

Alla fine quello che serve sono un generatore di analizzatori sintattici, una symbol table e . . .

. . .

. . .

bzz zzz

. . .

scusate, ci sono interferenze con zoom, ci sentiamo domani, 4rriz00m4rci

Appendice A

Prototipo di test di LFC, 2021

Benvenuti in questa meravigliosa sezione dove avremo modo di affrontare assieme l'esame fornito dalla Prof.ssa Quaglia per potersi preparare al meglio alla prova vera e propria. Cominciamo col dire che per affrontare tale esame è necessario:

1. avere ben chiaro quanto fatto fino ad ora: gli esercizi proposti spaziano dalle derivazioni leftmost trattate all'inizio del corso fino alle regole degli SDD; assicuratevi quindi di essere ben preparati.

2. trovare la strada più breve: il procedimento algoritmico sarà sempre a vostra disposizione avendo studiato, ma è pur sempre lungo. Arrivare al DFA minimo passando per Thompson, la Subset Construction e la Minimizzazione vi porterà via un sacco di tempo: all'esame avrete un'ora e mezza per cui cercate di perdere meno tempo possibile.

3. saper spaziare: soprattutto nell'ultimo esercizio verrà richiesto di avere fantasia e saper applicare quanto studiato in maniera differente dal semplice ragionamento.

Detto questo, cominciamo.

Esercizio 1

Se $\{ww \mid w \in \mathcal{L}((a \mid b)^*)\}$ è un linguaggio regolare rispondere "SI", altrimenti rispondere "NO".

Analizziamo il tutto a mente fredda: la domanda è se il linguaggio proposto è regolare, una sottoclasse dei linguaggi liberi. In questo caso non ha senso perdersi sulla definizione in quanto il linguaggio di partenza (i.e. quello a cui

appartiene w) è sicuramente regolare in quanto è denotato da un'espressione regolare. A questo sembrerebbe che la risposta sia a portata di mano: "Visto che i linguaggi regolari sono chiusi rispetto alla concatenazione allora il linguaggio fornito è regolare", giusto? NO!

Osserviamo meglio il tutto: se veramente valesse la proprietà di concatenazione potremmo scrivere il tutto come

$$\mathcal{L} = \{w_1 w_2 \mid w_1 \in \mathcal{L}_1 \wedge w_2 \in \mathcal{L}_2\}$$

dove L_1 e L_2 sono due linguaggi regolari.

Se questo fosse vero vorrebbe dire che $\mathcal{L}_1 = \mathcal{L}_2 = \mathcal{L}((a \mid b)^*)$ e quindi che $w' = ab \in \{w_1 w_2 \mid w_1 \in \mathcal{L}((a \mid b)^*) \wedge w_2 \in \mathcal{L}((a \mid b)^*)\}$, tuttavia $w' \notin \{ww \mid w \in \mathcal{L}((a \mid b)^*)\}$: non possiamo dunque sfruttare tale proprietà a nostro vantaggio.

L'ultima spiaggia è dunque utilizzare il *Pumpin Lemma* per poter arrivare ad una conclusione:

- Supponiamo che $\mathcal{L} = \{ww \mid w \in L((a \mid b)^*)\}$ sia un linguaggio regolare
- Prendiamo p intero positivo arbitrario
- Costruiamo la parola $z = a^p b^p a^p b^p$ (con $w = a^p b^p \in \mathcal{L}$); $\mid z \mid > p$
- $\forall u, v, w$ se $(z = uvw$ e $\mid uv \mid \leq p$ e $\mid v \mid > 0) \Rightarrow (\exists i \in N : uv^i w \notin \mathcal{L})$

Dovrebbe essere infatti abbastanza facile osservare che in questo caso, per non sbilanciare la stringa, dovrebbe essere necessario poter modificare contemporaneamente due porzioni che si trovano a distanza maggiore di p.

Dunque tale linguaggio non è regolare e la risposta corretta è **"NO"**. Visto che non è richiesta alcuna dimostrazione un modo per potersi accorgere di ciò è immaginare un NFA o DFA che rappresenti il linguaggio: ciò è impossibile perchè dovrebbe memorizzare il valore di w per poi ripeterlo. Questo vorrebbe dire creare infiniti stati per cui, letta una parola, l'automa obblighi a crearne una identica di seguito.

Esercizio 2

Se la seguente affermazione è vera rispondere "VERO", altrimenti rispondere "FALSO": "Se i linguaggi $\mathcal{L}_1$ e $\mathcal{L}_2$ sono entrambi regolari allora $\mathcal{L}_1 \cup \mathcal{L}_2$ è regolare."

In questo caso per rispondere è sufficiente accorgersi del fatto che l'affermazione corrisponde alla *proprietà di chiusura dei linguaggi regolari rispetto all'unione*: infatti, dati $\mathcal{L}_1$ e $\mathcal{L}_2$ linguaggi regolari, si ha che

$$\mathcal{L} = \{w \mid w \in \mathcal{L}_1 \vee w \in \mathcal{L}_2\}$$

e quindi $\mathcal{L}$ è ancora un linguaggio regolare.
La risposta all'esercizio è dunque **"VERO"**.

Esercizio 3

Sia $r = b^* \mid b^*a(\varepsilon \mid a \mid b)^*$e sia $\mathcal{D}$ il DFA minimo per il riconoscimento di $\mathcal{L}(r)$. Dire quanti stati ha $\mathcal{D}$ e quanti di questi stati sono finali.

Per poter risolvere l'esercizio è possibile applicare la Thompson's Construction per ottenere un NFA, successivamente la Subset Construction per trasformarlo in un DFA ed infine la DFA Minimization per ottenere $\mathcal{D}$. Il procedimento, seppur efficacie, è troppo lungo da svolgere (i.e. o saltate l'esercizio e ci ritornate dopo oppure trovate un metodo furbo per farlo): per questo motivo è meglio osservare direttamente la regex. Il linguaggio denotato è $\{b^k \mid k \geq 0\} \cup \{b^iaa^jb^k \mid i \geq 0, j \geq 0, k \geq 0\}$: se riuscissimo a trovare un DFA che ci permette di ottenere questo linguaggio poi potremmo ridurlo.

Partiamo dal primo caso: vogliamo avere la possibilità di generare 0 o più ripetizioni di b, dunque abbiamo bisogno che lo stato iniziale sia anche lo stato finale (in questo modo si ha la possibilità di ottenere la stringa vuota) e che vi sia un self-loop con etichetta b (per poter considerare le parole del tipo b^k).

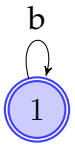


Figura A.1: Es 1: DFA Parziale b^*

Passiamo ora al secondo insieme di parole riconosciute: essendo che la prima parte è identica possiamo riciclare lo stato creato precedentemente, con la differenza che in questo caso non dovrà essere terminale; infatti dobbiamo avere almeno un'occorrenza di a. Proprio per questo motivo aggiungiamo un arco con etichetta a che porta in un nuovo stato terminale. A questo punto non ci resta che gestire il caso delle 0 o più occorrenze di a e di b: per fare ciò basterà come visto precedentemente aggiungere un self-loop per ognuno dei terminali che devono essere ripetuti.

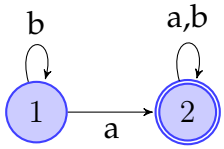


Figura A.2: Es 1: DFA Parziale $b^*a(\varepsilon \mid a \mid b)^*$

A questo punto abbiamo quasi concluso, la cosa che ci manca è unificare questi due DFA in modo che ci diano la possibilità di scegliere fra le due tipologie di parole denotate dalla nostra espressione regolare. Ancora una volta, una tecnica potrebbe essere quella di applicare l'operazione di unione prevista dalla Thompson's Construction e procedere come citato all'inizio. In questo caso però basta osservare che, per poter considerare valide anche le parole del tipo $\{b^k \mid k \geq 0\}$, basta semplicemente considerare lo stato 1 del secondo automa come terminale:

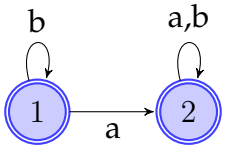


Figura A.3: Es 1: DFA Parziale $b^* \mid b^*a(\varepsilon \mid a \mid b)^*$

A questo punto possiamo applicare la minimizzazione del DFA:

- Separiamo gli stati terminali da quelli non terminali ottenendo $B_1 = \emptyset$ e $B_2 = \{1, 2\}$
- Visto che B_2 contiene la totalità degli stati che costituiscono il DFA, non è possibile che vi siano archi che vadano in altri stati al di fuori di B_2
- Il DFA minimo è costituito dunque da B_2

Il DFA minimizzato $\mathcal{D}$ è dunque così costruito:

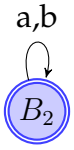


Figura A.4: Es 1: DFA Minimizzato $b^* \mid b^*a(\varepsilon \mid a \mid b)^*$

$\mathcal{D}$ possiede dunque **1 stato** e **1 stato finale**

Esercizio 4

Chiamiamo $\mathcal{D}$ il DFA ottenuto da $\mathcal{N}_1$ per subset construction e Q_l lo stato iniziale di $\mathcal{D}$. Dire a quale sottoinsieme degli stati di $\mathcal{N}_1$ corrisponde $Q[ab]$.

Per maggiore chiarezza possiamo rappresentare l'automa per intero:

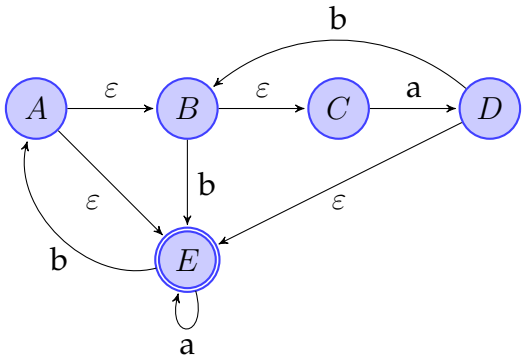


Figura A.5: Es 1: NFA di partenza

Eseguiamo dunque la Subset Construction:

states	a	b
$T_1 = \{A, B, C, E\}$	$\varepsilon\text{-}closure(\{D, E\}) = T_2$	$\varepsilon\text{-}closure(\{A, E\}) = T_1$
$T_2 = \{D, E\}$	$\varepsilon\text{-}closure(\{E\}) = T_3$	$\varepsilon\text{-}closure(\{A, B\}) = T_1$
$T_3 = \{E\}$	$\varepsilon\text{-}closure(\{E\}) = T_3$	$\varepsilon\text{-}closure(\{A\}) = T_1$

Tabella A.1: Es 4: Subset Construction

$\mathcal{D}$ risulta dunque così costruito:

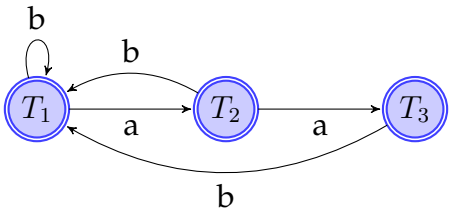


Figura A.6: Es 1: DFA risultante dalla Subset Construction

A questo punto, partendo dallo stato iniziale T_1, è possibile spostarsi allo stato indicato seguendo il percorso ab: lo stato di destinazione è dunque lo stato T_1, quindi la risposta è corretta è **{A, B, C, E}**

Esercizio 5

Chiamiamo $\mathcal{D}_m$ il DFA ottenuto per minimizzazione di $\mathcal{D}_1$ e P_1 o stato iniziale di $\mathcal{D}_m$. Dire a quale sottoinsieme degli stati di $\mathcal{D}_1$ corrisponde $P[abab]$

L'esercizio proposto è identico a quello svolto durante una lezione del corso: la sua risoluzione si può trovare in 5.2.3 *Esercizi sulla minimizzazione DFA, Esercizio 2*.

Esercizio 6

Scrivere l'intera riga della tabella di parsing $LL(1)$ per $\mathcal{G}_1$ relativa al non-terminale B.

Riportiamo $\mathcal{G}_1$ per praticità

$$
\begin{aligned}
S &\to AaB \mid b \\
A &\to BcBaA \mid \varepsilon \\
B &\to \varepsilon
\end{aligned}
$$

Per poter calcolare la tabella di parsing $LL(1)$ è necessario calcolare innanzitutto first e follow per ogni driver della grammatica $\mathcal{G}_1$.

	first	follow
S	$\{c, b, \varepsilon\}$	$\{\$\}$
A	$\{c, \varepsilon\}$	$\{a\} \cup follow(A) = \{a\}$
B	$\{\varepsilon\}$	$\{a, c\} \cup follow(S) = \{a, c, \$\}$

Tabella A.2: Es 6: First e Follow $\mathcal{G}_1$

A questo punto è possibile applicare l'algoritmo per la computazione della parsing table $LL(1)$: visto che ci interessa solamente la riga relativa al non-terminale B concentriamoci solamente sulla produzione $B \to epsilon$

- $\forall b \in first(B) \setminus \varepsilon,$ add $B \to epsilon$ to $M[B, b]$ (che non comporta nessuna modifica nella tabella)
- Visto che $\varepsilon \in first(B)$, allora $\forall x \in follow(B),$ add $B \to epsilon$ to $M[B, x]$

La riga identificata dal non-terminale B, nonché soluzione dell'esercizio, sarà dunque così rappresentata:

	a	b	c	\$
B	$B \to epsilon$		$B \to epsilon$	$B \to epsilon$

Tabella A.3: Es 6: Riga Parsing Table $LL(1)$

Esercizio 7

Siano: I lo stato iniziale dello $LR(1)$-automa per $\mathcal{G}_1$; T la tabella di parsing $LR(1)$ per G_1. Se T non contiene alcun conflitto nello stato $I[BcBa]$, rispondere "NO CONFLICT". Altrimenti, per ciascuna X tale che la entry $T(I[BcBa], X)$ contiene un conflitto, dire, specificando a quale X ci si riferisce: (i) di che tipo di conflitto si tratta; (ii) quale/i riduzione/i sono coinvolte.

Riportiamo $\mathcal{G}_1$ per praticità

$$
\begin{aligned}
&S \to AaB \mid b \\
&A \to BcBaA \mid \varepsilon \\
&B \to \varepsilon
\end{aligned}
$$

Nonostante sia possibile eseguire tutta la costruzione dell'automa caratteristico di tipo $LR(1)$ e la successiva computazione della tabella, anche in questo caso non è consigliabile procedere in quel modo: la cosa migliore è invece calcolare solo gli stati dell'automa caratteristico per poter arrivare in $I[BcBa]$ utilizzando le transizioni corrette. Se il lettore fosse interessato, può trovare lo svolgimento completo dell'esercizio nel capitolo 9.3.2 *Costruzione di una tabella di parsing LALR(1)*: per coerenza qui verranno utilizzati gli stessi numeri per gli stati riportati nell'esercizio indicato.

1. Inizializziamo lo stato 0; il suo kernel è

   $$S' \to \cdot S, \{\$\}$$

   Di questo kernel devo calcolare la chiusura:

   $$
   \begin{aligned}
   &S \to \cdot AaB, \{\$\} \\
   &S \to \cdot b, \{\$\} \\
   &A \to \cdot BcBaA, \{a\} \\
   &A \to \cdot, \{a\} \\
   &B \to \cdot, \{c\}
   \end{aligned}
   $$

Essendo questi gli item per lo stato 0, possiamo identificare quattro possibili transizioni (e quindi quattro possibili nuovi stati): $\tau(0, S) = 1$, $\tau(0, A) = 2$, $\tau(0, b) = 3$ e $\tau(0, B) = 4$.

Interessante notare che le produzioni del tipo $A \to \varepsilon$ vengono convertite in reducing item $A \to \cdot$.

2. Visto che siamo interessati a una transizione con etichetta B, computiamo lo stato 4; il suo kernel è

$$A \to B \cdot cBaA, \{a\}$$

La chiusura di tale stato non porta nuovi elementi ma è comunque possibile definire la transizione $\tau(4, c)$ allo stato 6

3. Seguiamo l'arco con etichetta a uscente da 4 e calcoliamo dunque il kernel per lo stato 6

$$A \to Bc \cdot BaA, \{a\}$$

la cui chiusura corrisponde a

$$B \to \cdot, \{a\}$$

Facendo notare la presenza di un reducing item, ci appuntiamo la transizione $\tau(6, B) = 8$

4. Utilizzando l'arco etichettato B ci spostiamo in 8 che ha kernel

$$A \to BcB \cdot aA, \{a\}$$

e possiede una transizione $\tau(8, a)$ allo stato 9

5. Seguendo l'arco con etichetta a arriviamo finalmente in 9, stato target dell'esercizio; il suo kernel è

$$A \to BcBa \cdot A, \{a\}$$

di cui possiamo calcolare la chiusura ottenendo

$$\begin{aligned} A &\to \cdot BcBaA, \{a\} \\ A &\to \cdot, \{a\} \\ B &\to \cdot, \{c\} \end{aligned}$$

Che contiene due reducing item e possiede due transizioni: la prima è $\tau(9, A) = 10$ e ha come target un nuovo stato mentre la seconda è $\tau(9, B) = 4$ che ha come target uno stato che fa già parte del nostro automa caratteristico.

Mediante le informazioni contenute nello stato 10 del nostro automa caratteristico siamo in grado di ricavare la seguente riga della tabella di parsing $LR(1)$:

	a	b	c	$	S	A	B
...							
9	r4		r5			10	4
...							

Tabella A.4: LR(1) & LALR(1) Parsing Table

dove $r4 = A \to \varepsilon$ e $r5 = B \to \varepsilon$.

Visto che la riga relativa allo stato 9 della tabella di parsing $LR(1)$ per $\mathcal{G}_1$ non presenta conflitti, allora la risposta corretta è **"NO CONFLICT"**.

Esercizio 8

Sia J lo stato iniziale dello $LR(1)$-automa per $\mathcal{G}_1$. Elencare gli item LR(1) che appartengono a $J[Aa]$.

Riportiamo $\mathcal{G}_1$ per praticità

$$
\begin{aligned}
&S \to AaB \mid b \\
&A \to BcBaA \mid \varepsilon \\
&B \to \varepsilon
\end{aligned}
$$

Come nel caso dell'esercizio precedente, adottiamo una tecnica che ci permetta di calcolare il minimo numero di stati possibili dell'automa caratteristico per poter calcolare quanto richiesto:

1. Inizializziamo lo stato 0; il suo kernel è

$$S' \to \cdot S, \{\$\}$$

   Di questo kernel devo calcolare la chiusura:

$$
\begin{aligned}
&S \to \cdot AaB, \{\$\} \\
&S \to \cdot b, \{\$\} \\
&A \to \cdot BcBaA, \{a\} \\
&A \to \cdot, \{a\} \\
&B \to \cdot, \{c\}
\end{aligned}
$$

Essendo questi gli item per lo stato 0, possiamo identificare quattro possibili transizioni (e quindi quattro possibili nuovi stati): $\tau(0, S) = 1$, $\tau(0, A) = 2$, $\tau(0, b) = 3$ e $\tau(0, B) = 4$.

2. A differenza del caso precedente, qui siamo interessati a percorrere l'arco etichettato A: analizziamo dunque lo target 2; il suo kernel è

$$S \to A \cdot aB, \{\$\}$$

Nemmeno in questo caso è necessario calcolare la sua chiusura in quanto il marker si trova davanti ad un non terminale: aggiungiamo dunque la transizione $\tau(2, a) = 5$ e proseguiamo.

3. Seguiamo l'arco uscente da 2, dirigiamoci nello stato 5, i.e. lo stato target del nostro esercizio; questo ha kernel

$$S \to Aa \cdot B, \{\$\}$$

La sua chiusura risulta invece essere pari a

$$B \to \cdot, \{\$\}$$

Gli item $LR(1)$ che appartengono allo stato target 5 del nostro automa caratteristico sono dunque:

$$S \to Aa \cdot B, \{\$\}$$
$$B \to \cdot, \{\$\}$$

Esercizio 9

Siano: H lo stato iniziale dell'automa caratteristico per il parsing $LALR(1)$ di $\mathcal{G}_1$; T la tabella di parsing $LALR(1)$ per $\mathcal{G}_1$. Se non ci sono conflitti nello stato $H[BcBaBc]$ di T, rispondere "NO CONFLICT". Altrimenti, per ciascuna X tale che la entry $T(H[BcBaBc], X)$ contiene un conflitto, dire, specificando a quale X ci si riferisce: (i) di che tipo di conflitto si tratta; (ii) quale/i riduzione/i sono coinvolte.

Riportiamo $\mathcal{G}_1$ per praticità

$$S \to AaB \mid b$$
$$A \to BcBaA \mid \varepsilon$$
$$B \to \varepsilon$$

Prima di procedere alla risoluzione dell'esercizio è necessario fare una precisazione: visto che uno svolgimento molto simile per per un automa caratteristico $LR(1)$ è già stato eseguito per l'esercizio 7 di questa appendice, il metodo più efficiente per arrivare alla risposta è, vista anche la grande similitudine tra gli item impiegati nel parsing $LALR(1)$, quello di orientare la risoluzione di questo esercizio basandosi sui risultati già ottenuti in precedenza. Per poter seguire meglio il ragionamento, andiamo a riportare gli stati incontrati fino ad ora, sostituendo il loro lookahead set con le variabili tipiche del parsing $LALR(1)$

1. Ricominciamo nuovamente dallo stato 0; il suo kernel è

   $$S' \to \cdot S, \{x_0\}$$

   e ha chiusura:

   $$\begin{aligned} &S \to \cdot AaB, \{x_0\} \\ &S \to \cdot b, \{x_0\} \\ &A \to \cdot BcBaA, \{a\} \\ &A \to \cdot, \{a\} \\ &B \to \cdot, \{c\} \end{aligned}$$

   Essendo questi gli item per lo stato 0, possiamo identificare quattro possibili transizioni (e quindi quattro possibili nuovi stati): $\tau(0, S) = 1$, $\tau(0, A) = 2$, $\tau(0, b) = 3$ e $\tau(0, B) = 4$. Segniamoci nel mentre da qualche parte che $x_0 = \{\$\}$.

2. Da 0 ci eravamo spostati seguendo l'arco con etichetta B nello stato 4; il suo kernel è

   $$A \to B \cdot cBaA, \{x_1\}$$

   Questo è caratterizzato dalla transizione $\tau(4, c) = 6$. Continuiamo a salvare le informazioni sulle variabili, in questo caso $x_1 = \{a\}$

3. Percorrendo l'arco etichettato a eravamo giunti in 6, il quale ha kernel

   $$A \to Bc \cdot BaA, \{x_2\}$$

   e chiusura

   $$B \to \cdot, \{a\}$$

   L'elemento di rilevanza è la transizione $\tau(6, B) = 8$. Appuntiamoci che $x_2 = x_1$

4. Sfruttando l'arco con label B ci spostiamo in 8; il suo kernel è

   $$A \to BcB \cdot aA, \{x_3\}$$

   e possiede una transizione $\tau(8, a)$ allo stato 9. Come prima (più di prima) segniamo che $x_3 = x_2$

5. Infine l'arco con etichetta a ci aveva portato in 9, con kernel

   $$A \to BcBa \cdot A, \{x_4\}$$

   e computando la sua chiusura avevamo ottenuto

   $$A \to \cdot BcBaA, \{x_4\}$$
   $$A \to \cdot, \{x_4\}$$
   $$B \to \cdot, \{c\}$$

   Che avevamo asserito contenere due reducing item e possedere due transizioni: $\tau(9, A) = 10$ e $\tau(9, B) = 4$. Inutile dire che è necessario memorizzare da qualche parte il fatto che $x_4 = x_3$.

   Per il momento di fatto non è cambiato nulla, se non la presenza dei lookahead set tipici del parsing $LALR(1)$; tuttavia è da questo momento che bisogna prestare attenzione: infatti, possiamo notare che il prossimo stato a cui dovremmo dirigerci è in realtà uno stato che abbiamo già incontrato in precedenza, cioè lo stato 4 (questo perchè possiede la stessa componente LR(0) dell'item LR(1)). Tornando in uno stato già visitato, dobbiamo andare a segnarci che $x_1 = \{a\} \cup x_4$.

6. Percorriamo proprio l'arco all'indietro etichettato B e torniamo in 4: di fatto non è necessario ricalcolare tutto, l'unica cosa che ci interessa è la presenza di una transizione $\tau(4, c) = 6$ per poter giungere a destinazione.

7. Seguendo infine l'arco già noto possiamo raggiungere lo stato 6, target di questo esercizio

A questo punto è il momento di tirare le dovute somme, iniziando dal riportare lo stato target 6 per fare maggior chiarezza:

- Kernel:

  $$A \to Bc \cdot BaA, \{x_2\}$$

- Chiusura:

  $$B \to \cdot, \{a\}$$

Come è possibile osservare, lo stato presenta una transizione $\tau(6, B) = 8$ e un reducing item: ciò si traduce nella presenza di un'operazione di `goto` 8 in corrispondenza del non-terminale B e di `reduce` $B \to \varepsilon$ in corrispondenza di a. Apparentemente non sembrerebbero esserci conflitti in questo stato ma siamo sicuri di non dimenticarci nulla? La risposta banalmente è sì, questo perchè:

1. abbiamo solo un reducing item, per cui non è possibile un conflitto `r/r`
2. non abbiamo operazioni di shift, dunque non è possibile nemmeno un conflitto `s/r`

La risposta corretta per l'esercizio è dunque **"NO CONFLICT"**.

Esercizio 10

Sia $\mathcal{G}$ la grammatica con produzioni nell'insieme $\{S \to SS+ \mid SS* \mid a\}$ e sia $w = aaa*+$. Se $w \notin L(\mathcal{G})$ rispondere "NON APPARTIENE". Altrimenti fornire una derivazione rightmost di w.

Se siete piacevolmente sorpresi dalla facilità dell'esercizio, sappiate che non siete gli unici.

La soluzione all'esercizio dunque è

$$S \Rightarrow SS+ \Rightarrow SSS*+ \Rightarrow SSa*+ \Rightarrow Saa*+ \Rightarrow aaa*+ \tag{A.1}$$

Esercizio 11

Sia P lo stato iniziale del parser $LALR(1)$ per la grammatica dello SDD $\mathcal{V}_1$. Il parser ha 4 conflitti shift/reduce: uno in $(P[EaE], a)$, uno in $(P[EaE], b)$, uno in $(P[EbE], a)$ e uno in $(P[EbE], b)$. Supponiamo che tutti e 4 i conflitti siano risolti a favore dello shift. Supponiamo inoltre che l'attributo $n.lexval$ del terminale n sia il numero intero rappresentato da n. Se l'input $2a3b4$ non è riconosciuto, rispondere "ERROR". Altrimenti dire quale valore viene valutato per $S.v$ su input $2a3b4$.

Riportiamo $\mathcal{V}_1$ per praticità

$S \to E$	$\{S.v = E.v\}$
$E \to n$	$\{E.v = n.lexval\}$
$E \to E_1 a E_2$	$\{E.v = E_1.v * E_2.v\}$
$E \to E_1 b E_2$	$\{E.v = E_1.v + E_2.v\}$

Iniziamo innanzitutto a tradurre l'input in un qualcosa che possa essere riconosciuto dall'analisi sintattica, trasformando la stringa $2a3b4$ in $nanbn\$$, operazione effettuata dall'analisi lessicale.

A questo punto dobbiamo capire come si comporta il nostro parser $LALR(1)$, ovviamente potremmo calcolare tutto l'automa caratteristico e la relativa tabella di parsing, ma questo porterebbe via molto tempo. Cerchiamo invece di eseguire il lavoro del parsing a mente, notando che:

- ogni volta che leggiamo una n possiamo eseguire una riduzione del tipo $E \to n$ ed applicare la regola corrispondente
- i conflitti si verificano in $(P[EaE], a)$, $(P[EaE], b)$, $(P[EbE], a)$ e $(P[EbE], b)$: questo vuol dire che, una volta finito di leggere un'operazione tra due numeri interi, il parser non sa se continuare a leggere oppure effettuare una riduzione e calcolare il valore; fortunatamente ci viene in aiuto il testo che ci dice di dare priorità all'operazione di `shift`

Caliamoci dunque nei panni del parser e immaginiamo come dovremmo considerare la nostra stringa $nanbn\$$

1. Come detto precedentemente, i terminali n comportano `reduce` $E \to n$ e ciò accade indipendentemente dal carattere letto dall'input buffer, questo perchè nel nostro linguaggio abbiamo la possibilità di fare parole del tipo $n,\ n+n,\ n*n,\ n+n*n,\ n*n+n$
2. Per questo motivo non dovrebbero esserci problemi ad arrivare al punto in cui $sySt = EaE$; essendo che abbimo eseguito implicitamente delle reduce ricordiamoci che le $E.v = n.lexval$
3. A questo punto avremmo un conflitto di shift/reduce citato in precedenza, tuttavia ci viene detto di dare priorità all'operazione di `shift`
4. Eseguendo tale operazione ci ritroviamo nella situazione analoga dove non si dovrebbero avere troppi problemi nell'arrivare al punto in cui $sySt = EaEbE$
5. A questo punto, leggendo in input \$, non è possibile fare altro se non eseguire un'operazione di `reduce` $E \to E_1 b E_2$, per cui $sySt = EaE$ e, applicando la regola associata alla riduzione, otteniamo $E.v = E_1.v + E_2.v = 3 + 4 = 7$
6. Sempre leggendo in input \$ applichiamo `reduce` $E \to E_1 a E_2$, per cui $sySt = E$ e otteniamo $E.v = E_1.v + E_2.v = 2 * 7 = 14$

7. Eseguiamo infine `reduce` $S \to E$, per cui $sySt = S$ e otteniamo $S.v = E.v = 14$

La risposta corretta risulta dunque essere $S.v = E.v = 14$

Esercizio 12

Sia P lo stato iniziale del parser $LALR(1)$ per la grammatica dello SDD $\mathcal{V}_1$. Il parser ha 4 conflitti shift/reduce: uno in $(P[EaE], a)$, uno in $(P[EaE], b)$, uno in $(P[EbE], a)$ e uno in $(P[EbE], b)$. Alcuni di questi conflitti dipendono dal fatto che la grammatica non modella la precedenza dell'operatore di moltiplicazione (operatore a) sull'operatore di somma (operatore b). Si dica quali conflitti sono dovuti alla suddetta carenza della grammatica e si dica come risolvere ciascuno di essi per fare in modo che a abbia precedenza su b.

Riportiamo $\mathcal{V}_1$ per praticità

$$
\begin{array}{ll}
S \to E & \{S.v = E.v\} \\
E \to n & \{E.v = n.lexval\} \\
E \to E_1 a E_2 & \{E.v = E_1.v * E_2.v\} \\
E \to E_1 b E_2 & \{E.v = E_1.v + E_2.v\}
\end{array}
$$

Per poter imporre la precedenza di un operatore sull'altro è possibile risolvere manualmente i conflitti all'interno della tabella decidendo che cosa fare: analizzando i conflitti, ci si accorge che sono dovuti a ambiguità non avendo definito associatività e precedenza degli operatori (e.g. nei casi in cui si abbia già letto EbE e nell'input buffer vi sia una a, è necessario calcolare il valore - eseguendo una `reduce` - oppure dare la precedenza all'operazione che ha a come operatore - eseguendo uno `shift` - e posticipare l'operazione corrente?). Per poter uscire da tale situazione, garantendo la precedenza di a rispetto a b, basterebbe accorgersi che:

- nel caso in cui si abbia letto fino ad ora EaE, si vorrà fare un'operazione di `reduce` leggendo b in input e garantendo di conseguenza la precedenza della a
- se invece avessimo letto EbE, avremmo dovuto effettuare uno `shift` nel caso avessimo letto una a in input per garantire la precedenza di quest'ultima

Per poter implementare correttamente ciò sarà necessario fornire al parser le direttive del caso per poter gestire autonomamente la situazione

I conflittti generati dalla precedenza sono duqnque $(P[EaE], b)$, che viene risolto effettuando la `reduce` $E \to E_1 a E_2$, e $(P[EbE], a)$, risolto effettuando lo `shift`. I restanti conflitti sono dovuti alla mancata associatività degli operatori per la grammatica definita.

Se il lettore fosse interessato, può trovare una spiegazione più dettagliata su un esempio analogo nella sezione 8.4.1 *Gestione dei conflitti*

Esercizio 13

Sia $\mathcal{G}$ la seguente grammatica:

$$S \to Aa \mid Bb$$
$$A \to aAb \mid ab$$
$$B \to aBbb \mid abb$$

Evitando di ricorrere alla computazione della tabella di parsing, spiegare perchè $\mathcal{G}$ certamente non è $LR(1)$.

Pur non essendo ambigua la grammatica presenta il problema che, nel momento in cui finisce di leggere la sequenza di a, non si ha idea se sia necessario eseguire un'operazione `reduce` $A \to ab$ oppure di `shift` in modo da poter eseguire successivamente `reduce` $B \to abb$. Per poter sapere che cosa compiere si dovrebbe conoscere l'ultimo non terminale della stringa: nel caso in cui fosse una a si saprebbe di dover applicare il primo caso descritto, se fosse una b il secondo.