Linguaggi di Programmazione

Silvio Peroni

17 marzo 2006

Licenza

Copyright © 2006 Silvio Peroni.
Si garantisce il permesso di riprodurre, distribuire, comunicare al pubblico, esporre in pubblico il presente documento e di creare opere derivate da esso sotto i termini di "Creative Commons: Attribuzione - Non Commerciale - Condividi allo stesso modo 2.0" consultabile al sito

http://www.creativecommons.it/Licenze/LegalCode/by-nc-sa

Introduzione

Questo documento è stato preparato sulla base delle lezioni di *Linguaggi di Programmazione* tenute dal prof. *Cosimo Laneve* nell'anno accademico 2004/2005, presso l'Università di Bologna, corso di Laurea in Informatica. Come supporto a tali lezioni è stato utilizzato il manuale *Compilers: Principles, Techniques, and Tools* di *Alfred V. Aho*, *Ravi Sethi* e *Jeffrey D. Ullman* (Prentice-Hall), da cui è stato tratto la quasi totalità del contenuto informativo del presente documento.

Il presente documento **non è in alcun modo sostitutivo** alle lezioni tenute ed agli appunti realizzati dal prof. Cosimo Laneve.

Indice

I Dall'analisi lessicale al parsing **9**

1 Introduzione ai compilatori **11**
1.1 I compilatori 11
1.2 Analisi del programma sorgente 12
1.2.1 Errori 14
1.3 Le fasi di un compilatore 14

2 Analisi lessicale **17**
2.1 Ruolo dell'analizzatore lessicale 17
2.2 Grammatiche, sentenze, linguaggi e espressioni regolari 18
2.3 Automa a stati finiti 20
2.3.1 Dall'espressione regolare al NFA 21
2.3.2 Dall'NFA al DFA 23
2.4 Flex 24

3 Analisi sintattica **27**
3.1 Grammatiche libero dal contesto 27
3.2 Parsing top-down 28
3.2.1 Parsing top-down a discesa ricorsiva 28
3.2.2 Parsing top-down LL(1) 30
3.2.2.1 Funzione First 30
3.2.2.2 Funzione Follow 31
3.2.2.3 Tabella LL(1) 32
3.3 Parsing bottom-up 32
3.3.1 Elementi LR(0), automi LR(0) e tabella *azione-goto* . . . 34
3.3.1.1 NFA e DFA con elementi LR(0) 34
3.3.1.2 Costruzione di un DFA diretta 36
3.3.1.3 Costruzione della tabella LR(0) 37
3.3.2 Parser Simple LR(1) {SLR(1)} 38
3.3.3 Parser LR(1) 39
3.3.3.1 NFA e DFA con elementi LR(1) 40
3.3.3.2 Costruzione di un DFA per un parser LR(1) diretta 40
3.3.3.3 Costruzione della tabella LR(1) 41
3.3.4 Il parser LookAhead LR(1) {LARL(1)} 42
3.3.4.1 Automa DFA e tabella LALR(1) 43
3.3.5 Grammatiche ambigue 43
3.3.6 Bison 44
3.4 Error Recovery 46

3.5 Classificazione delle grammatiche 48

II Dall'analisi semantica alla generazione del codice intermedio 49

4 Grammatiche con attributi 51
4.1 Attributi *sintetizzati* ed *ereditati* 51
4.2 Albero di sintassi astratta 54
4.3 Grammatiche con S-attributi e L-attributi 56

5 Tabella dei simboli 57

6 Type checking 59
6.1 Nozioni 59
6.1.1 Type checking delle Dichiarazioni e dei Tipi 59
6.1.2 Type checking di Espressioni 60
6.1.3 Type checking di Statement 60
6.1.4 Type Checking di Funzioni 61
6.2 Equivalenza 61
6.3 Conversioni di tipo 62
6.4 Overloading di funzioni e operatori 62
6.5 Funzioni polimorfe 63
6.5.1 Type checking di funzioni polimorfe 64

7 Ambienti di run-time 69
7.1 A cosa servono 69
7.2 Organizzazione della memoria 71
7.3 Tecniche di allocazione della memoria 73
7.3.1 Allocazione statica 73
7.3.2 Allocazione dinamica sullo stack 73
7.3.3 Allocazione dinamica sull'heap 77

8 Generazione del codice intermedio 81
8.1 Codice a tre indirizzi 81
8.2 Tabella dei simboli e albero di sintassi astratta 82
8.3 Dichiarazioni 82
8.4 Comandi di assegnamento 86
8.5 Espressioni booleane 90
8.6 Backpatching 96
8.7 Invocazioni di procedure 98

Parte I

Dall'analisi lessicale al parsing

Capitolo 1

Introduzione ai compilatori

1.1 I compilatori

Un *compilatore* é un programma che legge un programma scritto in *codice sorgente* e lo traduce in un altro programma equivalente scritto in un altro linguaggio, il *linguaggio di target*, riportando opportunamente eventuali errori presenti nel condice sorgente.

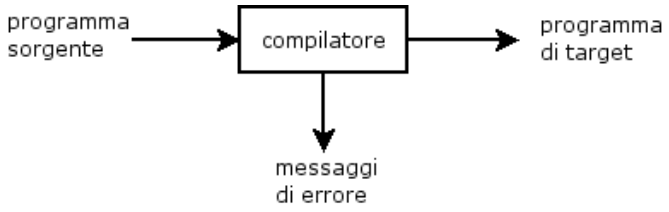


Figura 1.1: Un compilatore

La compilazione, come attività, viene divisa in due parti ben distinte:

1. l'*analisi*: ha il compito di spezzare il programma sorgente in "pezzi" separati, in modo da riuscire a creare una rappresentazione *intermedia* del programma sorgente. Durante questa fase, le operazioni richieste dal programma sorgente sono descritte da strutture gerarchiche chiamate *alberi*. In particolare, viene usato un tipo di albero, conosciuto come *albero sintattico* (dall'inglese, *syntax tree*), nel quale ogni nodo rappresenta un'operazione e i figli del nodo rappresentano gli argomenti dell'operazione.

2. la *sintesi*: ha il compito di costruire il programma di target a partire dalla rappresentazione intermedia fornita come prodotto dell'*analisi*.

Esistono molti tool in grado da manipolare il programma sorgente al fine di facilitare, e performare, ogni tipo di analisi. Di questi ricordiamo:

- gli *editor strutturati*, in grado di prendere in input una sequenza di comandi al fine di creare il programma sorgente. Sono in grado, inoltre, di analizzare il testo del programma, creando un opportuna struttura gerarchica sul programma sorgente.

- i *pretty printers*, che analizzano un programma e ne visualizzano, a video, la sua struttura.

- i *checker statici*, che, leggendo e analizzando il programma (quest'ultimo **non** viene eseguito), sono in grado di trovare potenziali bug.

- gli *interpreti*, che invece di produrre un programma di target come effetto di una traduzione, esegue le operazioni contenute nel programma sorgente. In sintesi, a differenza di un compilatore che *trasforma* i programmi, l'interprete *valuta* i programmi.

Qui di seguito viene mostrata una tipica sequenza di compilazione:

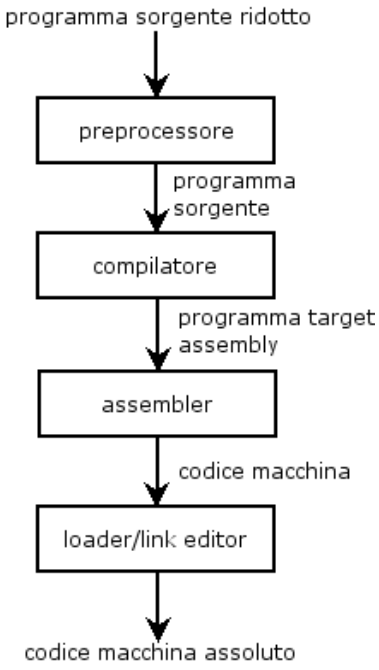


Figura 1.2: Sequenza di compilazione

1.2 Analisi del programma sorgente

Nella compilazione, l'analisi e' composta da tre fasi:

1. l'*analisi lineare*, in cui il programma sorgente viene letto da sinistra verso destra e viene raggruppato in *token*, ovvero sequenze di caratteri che hanno un significato comune.

2. l'*analisi gerarchica*, in cui i caratteri o i token vengono ragruppati gerarchicamente in gruppi con significati comuni.

3. l'*analisi semantica*, in cui, attraverso dei *check* affidabili, viene garantito che le componenti del programma sono legate fra loro da un significato corretto.

In un compilatore, l'analisi lineare viene chiamata *analisi lessicale* o *scanning*. In questa analisi, gli spazi bianchi, i *new line*, le tabulazioni, i commenti, vengono eliminati. ***analisi lessicale***

L'analisi gerarchica, invece, viene spesso indicata come *analisi sintattica* o *parsing*. Essa ha il compito di ragruppare i token del programma sorgente in determinate frasi grammaticali, usate poi dal compilatore per generare l'output. Queste frasi grammaticali sono generalmente rappresentate da un *albero di parsing*. ***analisi sintattica***

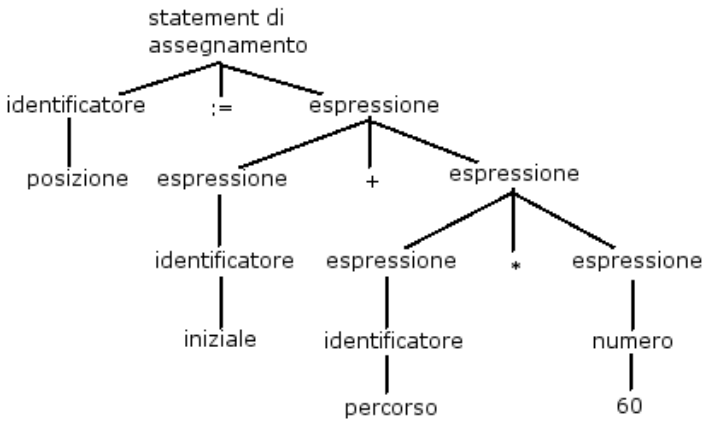


Figura 1.3: Albero di parsing per "posizione := iniziale + percorso * 60"

Solitamente, la struttura gerarchica é espressa tramite regole *ricorsive*. Per formalizzare le regole ricorsive vengono usate le *grammatiche libere dal contesto* (dall'inglese, *context-free grammars*). Mentre l'albero descritto nella figura 1.3 descrive la struttura sintattica dell'input, una piu' comune, e interna, rappresentazione di questa struttura sintattica viene fornita dall'*albero sintattico* in figura 1.4.

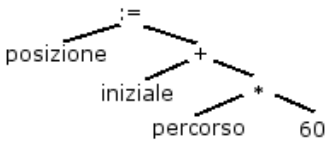


Figura 1.4: Albero sintattico per "posizione := iniziale + percorso * 60"

Un albero sintattico é una rappresentazione "compressa" dell'albero di parsing, in cui gli operatori appaiono come nodi interni e gli operandi sono i figli del nodo relativo a quell'operatore.

Nella fase di *analisi semantica*, si cerca se nel programma sorgente se sono presenti errori semantici e si raccolgono informazioni per la successiva fase di ***analisi semantica***

generazione del codice. Essa usa la struttura gerarchica, prodotto dell'analisi sintattica, per identificare operatori ed operandi di espressioni e statement e da come output un nuovo albero sintattico con la relativa *tabella dei simboli*, una struttura contenente un *record*, formato da più *campi*, per ogni identificatore. Uno degli organi più importanti dell'analisi semantica é il *type checking*, grazie al quale il compilatore controlla che ogni operatore sia associato ad operandi in modo legale, ovvero l'associazione deve seguire le specifiche del linguaggio sorgente.

1.2.1 Errori

Ogni frase scritta con il linguaggio sorgente può contenere errori. Solitamente, un compilatore, una volta individuato un errore, lo segnala e procede nella scansione del codice alla ricerca di eventuali altri errori. Un compilatore che, trovato il **primo** errore blocca la sua esecuzione uscendo e segnalando solo questo **non é** di gran aiuto al programmatore. Gli errori possibili individuabili variano a seconda della fase in cui si ci trova. Nella fase lessicale si possono individuare errori dove i caratteri rimasti nell'input non formano un token del linguaggio. Gli errori in cui una serie di token violano le regole strutturali (o sintattiche) del linguaggio sono determinati dall analisi sintattica. Durante l'analisi semantica il compilatore prova ad individuare costrutti con una corretta struttura sintattica ma senza un corretto significato relativo alle operazioni da intraprendere (ad esempio, se si provano a sommare due identificatori, uno relativo al nome di un array mentre l'altro relativo al nome di una procedura).

1.3 Le fasi di un compilatore

Concettualmente, un compilatore opera suddividento il suo lavoro in 6 principali *fasi*, descritte in figura 1.5.

generazione codice intermedio

Dopo le analisi sintattica e semantica, precedentemente descritte (vedi la sezione 1.2), molti compilatori generano un esplicito *codice intermedio* derivato dal programma sorgente. Il codice intermedio, che può essere pensato come una rappresentazione di un programma per una macchina astratta, gode di due importanti proprietà: é facile da produrre ed é facile da tradurre nel programma di target. Ovviamente, la rappresentazione intermedia può avere svariate forme (gerarchica, lineare, ad albero, a triple, ecc).

ottimizzazione del codice

La fase di *ottimizzazione del codice* cerca di migliorare il codice intermedio in modo da redere più performante il codice macchina ritornato dalla compilazione.

generazione del codice

La fase finale di un compilatore é la generazione del codice di target, normalmente composto da codice macchina o codice assembly. Una aspetto cruciale di questa fase é l'assegnazione delle variabili, contenute nel codice intermedio, ai *registri*.

Ma cosa rende un compilatore "buono"? Si può dare una breve descrizione per punti relativi alla qualità del compilatore in base a:

- correttezza
- performace dei programmi tradotti
- scalabilità del compilatore

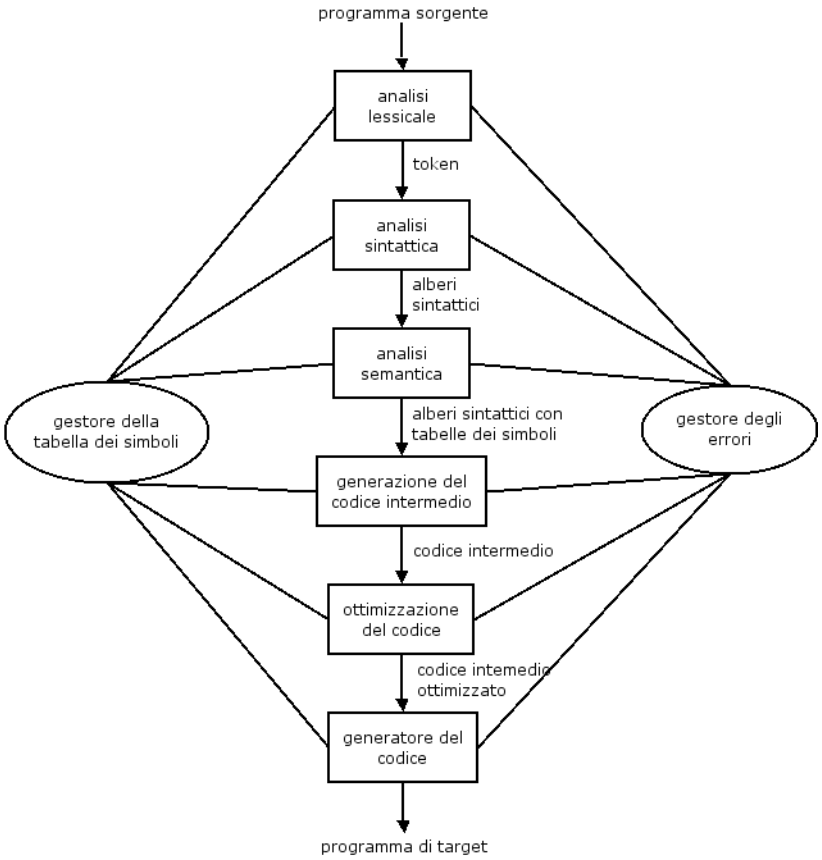


Figura 1.5: Fasi di un compilatore

1. il compilatore é veloce
2. compilazione separata

- facile da modificare
- ausilio alla programmazione

1. messaggi di errore intelligibili
2. supporto per il debugging

Capitolo 2

Analisi lessicale

2.1 Ruolo dell'analizzatore lessicale

L'*analisi lessicale* é la prima fase di un processo di compilazione. Il compito principale di questa fase é quello di leggere una serie di caratteri in input al fine di produrre, come output, una sequenza di token, la cui specifica viene data dalle *espressioni regolari* opportunamente strutturate in regole all'interno dell'analizzatore, che verranno usati dal *parser* nell'*analisi sintattica*. Questa interazione, schematizzata nella figura 2.1, é l'implementazione utilizzata più comunemente e lega strettamente al parser l'analizzatore lessicale, come se quest'ultimo fosse una *subroutine* del primo.

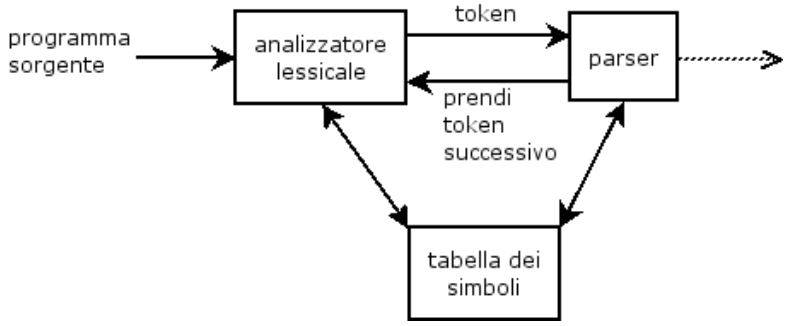


Figura 2.1: Interazione tra l'analizzatore lessicale e il parser

L'analizzatore lessicale svolge altri due compiti fondamentali: elimina dal programma sorgente, al fine di portare a termine le sue analisi, tutti i commenti e tutti gli spazi bianchi (black, tab, newline, ecc)[1], e genera, nel caso se ne rilevasse la presenza, messaggi di errore opportuni che verranno restituiti al termine della compilazione. Oltre a ciò, un analizzatore lessicale deve essere **completo**, ovvero deve ritornare qualcosa in ogni caso (funzionamento corretto

[1] Ovviamente, questa modifica non si ripercuote sul programma sorgente creato dall'utente, ma sulla "copia" di tale programma utilizzata dal processo di compilazione. In generale, quindi, l'analizzatore lessicale scorre il file contenente il programma sorgente come se fosse scritto tutto su una sola riga esente di spazi.

o errore), e deve risolvere i **casi di ambiguità**, risoluzione resa possibile da due regole fondamentali:

1. *match più lungo*: la sottostringa iniziale dell'input più lunga che corrisponde ad un'espressione regolare, relativa alle regole dell'analizzatore, é quella scelta come prossimo token.
2. *priorità delle regole*: se la più lunga sottostringa corrisponde a più regole, allora la regola scelta é la prima della sequenza.

In alcuni casi, gli analizzatori lessicali vengono divisi in due fasi svolte a catena: la fase di *scanning*, responsabile dei compiti più semplici e facilmente risolvibili, e la fase di *analisi lessicale*, a cui fanno capo le operazioni di risoluzioni più complesse.

2.2 Grammatiche, sentenze, linguaggi e espressioni regolari

grammatica

Una *grammatica* G é una quadrupla $G = (V, T, S, P)$, dove:

- l'insieme V é finito e rappresenta il *vocabolario*
- l'insieme T é finito, contiene i *simboli terminali* della grammatica ed ha come restrizione $T \subseteq V$. Ogni terminale della grammatica é anche un *token lessicale*, proprio del processo di compilazione.
- S é un simbolo *non terminale* ($S \in V - T$) detto *simbolo iniziale* della grammatica.
- P é l'insieme delle *produzioni* della grammatica, le quali hanno forma $\alpha \rightarrow \beta$, $\alpha \in V^+ \; \beta \in V^*$.

sentenza

Considerando una qualunque alfabeto **finito**, una *sentenza* (o anche *stringa*) é una sequenza finita di simboli che compaiono nell'alfabeto. La stringa *vuota* viene indicata con il simbolo ϵ. Le stringhe, relative ad un alfabeto, godono di determinate proprietà quali la *concatenazione* (se x e y sono stringhe, la concatenzione di x e y é la stringa formata aggiungendo y al termine di x, e si scrive xy) e l'*elemento neutro* ϵ *destro e sinistro* per la concatenazione (consideriamo la stringa s, allora $s\epsilon = \epsilon s = s$). Inoltre una stringa può essere suddivisa in *parti*, definite nella tabella 2.2.

linguaggio

Un *linguaggio* L associato ad una grammatica G, e si scrive L_G, é l'insieme così definito: $L_G = \{w \in T^* | s \rightarrow^* w\}$, dove il $\rightarrow^*$ indica infiniti passi di *derivazione* (vedere a tal proposito la sezione 3.1). Le operazioni fondamentali su un linguaggio sono quattro e sono così definite:

1. *unione* di due linguaggi, L e M: $L \cup M = \{s | s \in L \; or \; s \in M\}$.
2. *concatenazione* di due linguaggi, L e M: $LM = \{st | s \in L \; and \; t \in M\}$.
3. *chiusura di Kleene* di un linguaggio L: $L^* = \bigcup_{i=0}^{\infty} L^i$(zero o più concatenazioni di L).

Termine	Definizione
prefisso di s	Una stringa ottenuta come risultato della rimozione di zero o più simboli contigui, a partire dall'inizio, sulla stringa s; *c*, *ci*, *cia* sono prefissi della stringa *ciao*.
suffisso di s	Una stringa ottenuta come risultato della rimozione di zero o più simboli contigui, a partire dalla fine, sulla stringa s; *o*, *ao*, *iao* sono suffissi della stringa *ciao*.
sottostringa di s	Una stringa ottenuta cancellando un prefisso e un suffisso dalla stringa s; ad esempio, *ia* é una sottostringa di *ciao*. Ogni prefisso ed ogni suffisso sono una sottostringa di s, ma **non é vero** che ogni sottostringa di s é anche un suffisso o un prefisso di s. Per ogni stringa s, sia s stessa sia ϵ sono prefissi, suffissi e sottostringhe di s.
prefisso, suffisso e sottostringa *propria* di s	Ogni stringa non vuota x che é rispettivamente un prefisso, un suffisso o una sottostringa di s e che $x \neq s$.
sottosequenza di s	Ogni stringa ottenuta come risultato della rimozione di zero o più simboli, non necessariamente contigui, da s; *ca* é una sottosequenza di *ciao*.

Tabella 2.2: Parti di una stringa

4. *chiusura "positiva"* di un linguaggio L: $L^+ = \bigcup_{i=1}^{\infty} L^i$(una o più concatenazioni di L).

Le *espressioni regolari* su un alfabeto A sono definite dalle seguenti regole: ***espressioni regolari***

1. ϵé un'espressione regolare che denota il linguaggio $\{\epsilon\}$.
2. se $a \in A$, a é un'espressione regolare che denota il linguaggio $\{a\}$.
3. supponiamo r e s espressioni regolari che denotano rispettivamente i linguaggi L_re L_s:
   (a) $r|s$ é un'espressione regolare che denota il linguaggio $L_r \cup L_s$.
   (b) rs é un'espressione regolare che denota il linguaggio $L_r L_s$.
   (c) r^* é un'espressione regolare che denota il linguaggio L_r^*.
   (d) r é un'espressione regolare che denota il linguaggio L_r.

Un linguaggio denotato da un'espressione regolare é anche detto *insieme regolare*. Le operazioni viste nel punto 3 precedente vanno considerate con la seguente priorità (dalla più alta alla più bassa): stella di Kleene, concatenazione e |. Per quel che concerne le proprietà algebriche delle espressioni regolari, ne viene data una visione globale nella tabella 2.3.

A volte, nella parte destra delle produzioni grammaticali, si possono usare anche espressioni regolari per velocizzarne la scrittura e la comprensibilità. Queste "scoraciatoie" possono essere di quattro tipi fondamentali:

Assioma	Descrizione
$r\|s = s\|r$	$\|$ é commutativa
$r\|(s\|t) = (r\|s)\|t$	$\|$ é associativa
$(rs)t = r(st)$	la concatenazione é associativa
$r(s\|t) = rs\|rt$ e $(s\|t)r = sr\|tr$	la concatenazione é distributiva rispetto a $\|$
$\epsilon r = r\epsilon = r$	ϵ é l'elemento neutro destro e sinistro per la concatenazione
$r^* = (r\|\epsilon)^*$	relazione tra * e ϵ
$r^{**} = r^*$	* gode dell'idempotenza

Tabella 2.3: Proprietà algebriche delle espressioni regolari

1. *Una o più istanze.* L'operatore unario "+" posto come apice di una espressione regolare r (r^+) assume il significato di "una o più istanze di r". L'espressione regolare così formata denota il linguaggio L_G^+. Inoltre l'operatore "+" gode della stessa precedenza e dell'associatività che caratterizzano l'operatore "*" (stella di Kleene).

2. *Zero o una istanza.* L'operatore unario "?" posto di seguito ad una espressione regolare r ($r?$) assume il significato di "zero o una istanza di r". Questo èattern é una "scorciatoia" all'espressione regolare $r|\epsilon$.

3. *Generico carattere.* L'operatore "." può essere chiamato, forse impropriamente, espressione regolare e denota una qualsiasi carattere dell'alfabeto, escluso il *new line*. Questo operatore é tipico nel caso preatico in cui si stiano utilizzando espressioni regolari su un calcolatore, ma non viene considerato nella teoria delle espression regolari.

4. *Classi di caratteri.* La notazione "[abc]", dove "a", "b" e "c" sono simboli dell'alfabeto, denota l'espressione regolare $a|b|c$. Un'abbreviazione di una di queste classi di caratteri, come ad esempio "[a-z]", denota l'espressione regolare $a|b|c|\ldots|z$. In generale tutti questi gruppi possono essere formati sia da caratteri sia da numeri, a meno di utilizzare le forme contratte, nelle quali in primo simbolo e il secondo simbolo devono essere dello stesso tipo.

2.3 Automa a stati finiti

Un *riconoscitore* per un linguaggio é un programma che prende in input una stringa x e che rispondi "sì" se la stringa appartiene al linguaggio, "no" altrimenti. Le espressioni regolari, che, ricordiamo, definiscono un linguaggio e, più nello specifico, delineano le regole di un analizzatore lessicale, possono essere trasformate in determinati riconoscitori chiamati *automi*. Gli automi sono riconoscitori di token lessicali e possono essere *deterministici*, ovvero automi nei quali ogni stato non ha mai due archi uscenti etichettati con lo stesso carattere e non ha ma ϵ-transizioni, e *non deterministici*, in caso contrario. In ognuno dei due casi, la definizione formale di un automa é

$$M = (Q, T, q_i, F)$$

dove Q é l'insieme degli stati dell'automa, q_i é lo stato iniziale dell'automa ($q_i \in Q$), F é l'insieme degli stati finali dell'automa ($F \subseteq Q$) e T é l'insieme composto dalle transizioni stato-carattere-stato della forma $(q_{partenza}, a, q_{arrivo})$ con a generico simbolo dell'alfabeto preso in cosiderazione e $q_{partenza}, q_{arrivo} \in Q$. Sono gli elementi contenuti in quest'ultimo insieme a caratterizzare la forma deterministica o meno dell'automa. Una stringa si dice **accettata** dall'automa se, partendo dallo stato iniziale dell'automa e saltando negli stati successivi a seconda dei caratteri letti nella stringa di input, dopo n transizioni, dove n é la lunghezza della stringa, l'automa si trova in uno **stato finale**; in caso contrario la stringa non é accettata.

Il *linguaggio* riconosciuto da un automa rappresenta l'insieme delle stringhe accettate. Notare che si fa sempre rifermento ad automi generali senza renderne esplicito il tipo. Questo perchè gli automi a stati finiti deterministici, DFA, e non determisitici, NFA, hanno *lo stesso potere espressivo*, ovvero avendo un NFA é sempre possibile costruirne uno DFA riconoscitore dello stesso identico linguaggio e viceversa. Negli analizzatori lessicali si fa largo uso dei DFA, tralasciando completamente i NFA a causa della complessità: al contrario del deterministico, che ha costo *lineare*, un NFA ha costo *esponenziale*, dovuto al necessario *backtracking* utilizzato per capire se una stringa é accettata o meno.

Mentre graficamente si usa la rappresentazione standard degli automi (una freccia entrante sullo stato iniziale, stati con doppio bordo per gli stati finali, ecc), in un analizzatore lessicale l'implementazione di un DFA é data attraverso delle matrici, matrici che necessitano di uno *stato pozzo*, non finale, indicato dalla cifra "0" utile a codificare l'assenza di un arco (in questo stato l'automa cicla per qualunque input), e di un *array per gli stati finali*, che mappa ogni stato alle azioni corrispondenti.

2.3.1 Dall'espressione regolare al NFA

Si può definere un algoritomo (1) in grado di trasformare ogni espressione regolare nell'NFA corrispondente.

Algorithm 1 (Costruzione di Thompson) Da una espressione regolare ad un NFA

Input. Un'espressione regolare r su un alfabeto A.
Output. Un NFA N che accetti il linguaggio L_r.
Metodo. Si scorre r suddividendola in sottoespressioni. Poi, usando le regole (1) e (2), descritte sotto, si costruisce un NFA per ogni simbolo base in r (ovvero per ogni simbolo dell'alfabeto o ϵ). Fatto ciò, seguendo la struttura sintattica dell'espressione regolare r, si combinano *induttivamente* gli NFA finora generati usando la regola (3). Gli NFA intermedi così prodotti, nei vari passi di costruzione, corrispondono a sotto espressioni di r e hanno un'importante e fondamentale proprietà: ci sono esattamente un solo stato iniziale ed un solo stato finale.

Le regole usate per la trasformazione in NFA di espressioni regolari, a cui si fa riferimento anche dentro l'algoritmo 1, sono cinque, due casi base e tre regole induttive:

1. Per il simbolo ϵ il relativo NFA é:

Figura 2.2: NFA per il simbolo ϵ

2. Per ogni simbolo dell'alfabeto $a \in A$ il relativo NFA é:

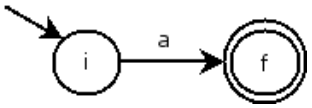


Figura 2.3: NFA per un qualsiasi simbolo dell'alfabeto a

3. Sopponiamo N_s e N_t due NFA rispettivamente per le espressioni regolari s e t. Allora si possono costruire i seguenti NFA:

   (a) Per l'espressione regolare $s|t$ il relativo NFA é:

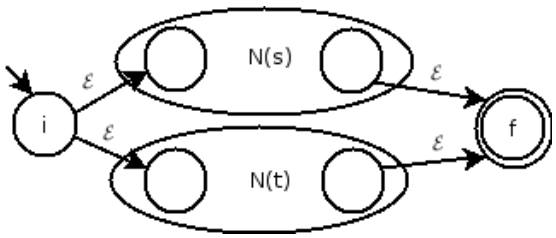


Figura 2.4: NFA per l'espressione regolare $s|t$

   (b) Per l'espressione regolare st il relativo NFA é:

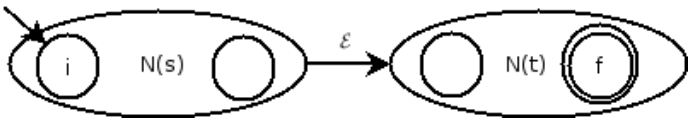


Figura 2.5: NFA per l'espressione regolare st

   (c) Per l'espressione regolare s^* il relativo NFA é:

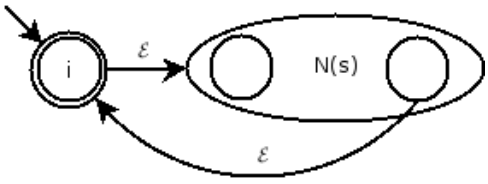


Figura 2.6: NFA per l'espressione regolare s^*

2.3.2 Dall'NFA al DFA

Si può definere un algoritomo (2) in grado di trasformare ogni NFA in un DFA corrispondente.

Algorithm 2 Costruzione di un DFA dato un NFA

Input. Un NFA *N*.
Output. Un DFA *D* che accetta lo stesso linguaggio di *N*.
Metodo. Si utilizzano i due insiemi, *InsStati* e *TranStati*, il primo inizializzato con il solo stato iniziale dell'automa DFA non marcato, dato dalla ϵ-chiusura di s_0 (dove s_0 é lo stato iniziale dell'NFA), il secondo inizializzato vuoto. Questi due insiemi rappresentano, rispettivamente, gli stati dell'automa DFA e le relative transizioni stato-carattere-stato.

while c'é uno stato q in *InsStati* non marcato
  marca q;
  for ogni simbolo di input a
    $q^{'}$:= ϵ-chiusura(muovi(q,a));
    if $q^{'}$ non é in *InsStati* **then**
      aggiungi $q^{'}$ come stato non marcato in *InsStati*;
    TranStati[q, a] := $q^{'}$;

Le funzioni *ϵ-chiusura* e *muovi* sono semplicemente definite come segue:

- *ϵ-chiusura(Q)* dove Q é un insieme di stati: é l'insieme composto da tutti gli stati di Q, chiamiamoli $q_1 \dots q_n$, più tutti quegli stati raggiungibili a partire da ogni q_i utilizzando soltanto ϵ-transizioni. Ovvero: *ϵ-chiusura(Q)* é il più piccolo insieme che soddisfa le seguenti due regole:
  1. se $q \in Q \Rightarrow q \in$ *ϵ-chiusura(Q)*.
  2. se $q \in$ *ϵ-chiusura(Q)* **e** $q \rightarrow^* q^{'} \in T \Rightarrow q^{'} \in$ *ϵ-chiusura(Q)*.
- $muovi(Q, a)$ dove Q é o uno stato o un insieme di stati e a é un simbolo dell'alfabeto: é l'insieme composto da tutti gli stati raggiungibili a partire da ogni $q_i \in Q$ utilizzando soltanto transizioni etichettate a. Ovvero: $muovi(Q, a) = \{q^{'} \mid q \rightarrow^a q^{'} \in T, q \in Q\}$.

In principio, il numero degli stati che un DFA potrebbe avere é 2^n, dove n é il numero degli stati dell'NFA. Di solito, però, il numero degli stati del DFA

é proporzionale ad n (lineare). C'é inoltre da ricordare che l'automa prodotto con questo algoritmo **non** é il minimo possibile. Questo, per i compilatori, non é significativo poichè il tempo di compilazione di un programma usato quando si vuole anche minimizare l'automa é largamente superiore a quello usato per completare la compilazione con l'automa non minimizzato.

2.4 Flex

Flex é un programma per la construzione di analizzatori lessicali partendo da una particolare notazione basata su espressioni regolari.

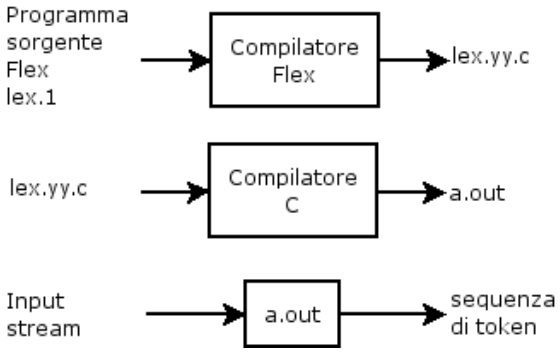


Figura 2.7: Creazione di un analizzatore lessicale con Flex

Come si può vedere dalla figura 2.7, Flex prende in input un file in formato speciale e genera un nuovo file C, *lex.yy.c*, che definisce una funzione *yylex()*. Il file viene poi compilato e linkato con la libreria "-lfl" in modo che venga generato un eseguibile che, quando eseguito, analizza la stringa di input e man mano che trova espressioni regolari descitte nel codice, esegue il comando corrispondente.

Un programma Flex é suddiviso in tre parti:

definizioni
%%
regole
%%
codice utente

definizioni

La sezione *definizioni* contiene le *dichiarazioni di nomi* la cui sintassi é

nome definizione

dove *nome* é un identificatpre C e *definizione* é un espressione regolare. Ad esempio, inserendo

DIGIT [0-9]
ID [a-z][a-z0-9]*

nelle *definizioni*, é possibile fare riferimento a quelle espressioni regolari richiamandole grazie al loro *nome*, lasciando perdere la loro *definizione*.

Alcune *definizioni* vengono dette di *start conditions* e sono un meccanismo per attivare in maniera condizionata le regole. Se, ad esempio, compare la start conditions "%s example", le regole prefissate da "<example>" saranno attive.

La sezione *regole* contiene una sequenza di regole della forma ***regole***

pattern azione

dove, i tipi di *pattern* più usati sono:

x	il carattere *x*
"*"	il carattere speciale *
"+"	il carattere speciale +
"-"	il carattere speciale -
"/"	il carattere speciale /
"|"	il carattere speciale |
\n,\b,\t	i caratteri newline, blank, tab
.	ogni carattere eccetto il newline
[xyz]	una classe di caratteri (rappresenta l'espressione x|y|z)
[aj-oZ]	una classe di caratteri con range
r*	zero o più *r*, *r* espressione regolare
r+	una o più *r*, *r* espressione regolare
r?	zero o una *r*, *r* espressione regolare
{NAME}	l'espansione di NAME (dichiarazione)
(r)	espressione regolare *r*, le parentesi per la precedenza
rs	espressione regolare *r* seguita da *s*

Le *azioni* sono comandi in C e rappresentano le operazioni che vengono eseguite quando un certo pattern é riconosciuto.

La sezione *codice utente* contiene le funzioni ausiliarie in C che possono essere chiamate dallo scanner. Questa porzione é opzionale. ***codice utente***

Risulta importante fare alcune osservazioni. Nelle sezioni *definizioni* e *regole*, ogni testo racchiso tra "%{" e "%}" é copiato nel file .c prodotto da Flex. Questo serve a definire *variabili globali*, *istruzioni su di esse*, *warnings*, ecc. In aggiunta a ciò, Flex utilizza un certo numero di variabili che possono essere referenziate attraverso il loro nome. Le più usate sono *yytext*, memorizza il testo riconosciuto in input (tipo stringa), e *yyleng*, memorizza la lunghezza del testo (tipo intero).

Vediamo ora un esempio di file di input per Flex (scanner di espressioni aritmentiche):

```
%{
#include <stdio.h>
%}
%%
[\t\n]+   printf(“carattere bianco, lunghezza %i\n”,yyleng);
“*”   printf(“per\n”);
“/”   printf(“diviso\n”);
“+”   printf(“più\n”);
“-”   printf(meno\n”);
“(“   printf(“par-sinistra\n”);
“)”   printf(“par-destra\n);
0|([1-9][0-9]*)   printf(“intero %s\n”,yytext);
[a-zA-Z][a-zA-Z0-9]*   printf(“identificatore %s\n”,yytext);
.   printf(“carattere illegale %s\n”,yytext);
%%
main() {
  yylex();
}
yywrap() {
  return(1);
}
```

Si ricordi, inoltre, che per risolvere i **casi di ambiguità**, come precedentemente descritto, Flex utilizza due tecniche principali:

1. *match più lungo*: la sottostringa iniziale dell'input più lunga che corrisponde ad un'espressione regolare, relativa alle regole dell'analizzatore, é quella scelta come prossimo token.

2. *priorità delle regole*: se la più lunga sottostringa corrisponde a più regole, allora la regola scelta é la prima della sequenza.

Un ultimo appunto va riservato ai possibili errori generabili in questa fase. Come risulta logico, gli unici errori rilevabili sono quelli relativi alla scansione lessicale. Se sono presenti errori *sintattici* o *semantici*, questi non vengono rilevati in questa fase ma nelle relative seguenti fasi di *analisi sintattica* e *analisi semantica*.

Capitolo 3

Analisi sintattica

3.1 Grammatiche libero dal contesto

La sintassi dei linguaggi di programmazione é descritta attraverso grammatiche *libere dal contesto*, anche dette *BNF* (*Backus-Naur Form*). Queste grammatiche offrono notevoli vantaggi ai linguaggi da esse generate:

- Una grammatica da una *precisa*, *facile* da capire, descrizione sintattica ad un linguaggio di programmazione.
- Per certe classi grammaticali, si possono costruire efficienti *parser* che determinano se un programma sorgente é sintatticamente corretto.
- Una grammatica strutturata opportunamente fornisce una struttura al linguaggio di programmazione che risulta utile sia per tradurre il programma sorgente nel corretto codice oggetto sia per rilevare eventuali errori.

Come già discusso nella sezione 2.2, anche le grammatiche libere dal contesto sono della forma

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

ma, al contrario delle generali, hanno produzioni della forma

$$A \to \alpha,\ A \in (V - T)\ \alpha \in V^*$$

derivazione

Una **derivazione** é un processo di riscrittura in cui ogni stringa viene ottenuta dalla precedente riscrivendo un simbolo non terminale A con la parte destra di una produzione che ha a sinistra A. In ogni caso, la stringa di partenza é sempre il *simbolo iniziale* della grammatica. Due sono i principali metodi di derivazione utilizzati: il *lestmost*, dove il simbolo non terminale che viene rimpiazzato é quello più a sinistra, e *rightmost*, dove il simbolo non terminale che viene rimpiazzato é quello più a destra.

linguaggio

Un **linguaggio** che é generato da una grammatica libera dal contesto viene chiamato *linguaggio libero dal contesto* ed é definito $L_G = \{w \in T^* | s \to^* w\}$, dove il $\to^*$ indica infiniti passi di riscrittura (derivazione). In poche parole, il linguaggio definito da una grammatica é l'insieme delle frasi terminali (contenenti

soltanto simboli terminali) che sono derivabili con derivazioni della grammatica. Se due grammatiche generano lo stesso linguaggio, le grammatiche vengono dette *equivalenti*.

albero sintattico

Un **albero sintattico**, o *parse tree*, é una rappresentazione grafica per una derivazione in cui ogni nodo interno dell'albero é etichettato con un non terminale $A \in (V - T)$ ed i relativi figli sono etichettati, da sinistra verso destra, con i simboli presenti nella parte destra della produzione relativa ad A. Le foglie dell'albero sono etichettate da simboli terminali e, lette da sinistra verso destra, costituiscono una *sentenza*.

ambiguità

Un grammatica si dice **ambigua** quando, data una frase s ben definita, esistono due alberi sintattici differenti o, equivalentemente, due derivazioni *leftmost* per la stessa frase. Le ambiguità non sono sempre pericolose e sono, a volte, estremamente difficili da risolvere: non esiste un algoritmo generale che, data una grammatica ambigua, ne generi un'altra, equivalente alla precedente, non ambigua.

3.2 Parsing top-down

Gli algoritmi di *parsing top-down* analizzano una stringa di input cercando di ricostruire i passi di una derivazione *leftmost*. Sono detti top-down perche sottintendono una visita anticipata dell'albero sintattico (dalla radice alle foglie). Verranno discussi due tipi di parsing top-down: quelli a *discesa ricorsiva* e quelli *LL(1)*.

3.2.1 Parsing top-down a discesa ricorsiva

L'idea che sta alla base dei parsing a *discesa ricorsiva* é quella che vuole le regole grammaticali per un non terminale A come costituenti una procedura che riconosce A:

- la parte destra di ogni regola specifica la struttura del codice per questa procedura.
- la sequenza di terminali e non terminali nelle varie regole corrisponde ad un controllo che i terminali siano presenti sia nell'input sia nelle invocazioni delle procedure dei simboli non terminali.
- la presenza di diverse regole per A é modellata da *case* o *if*.

Per fare un esempio, consideriamo la seguente grammatica (S simbolo iniziale):

S→begin S L | print E
L→end | ; S L
E→num

Il parser a discesa ricorsiva per questa grammatica ha una funzione per ogni terminale e una clausola per ogni produzione:

```
enum token {BEGIN, PRINT, END, SEMI, NUM};
extern enum token getToken();
enum token tok;
```

```
void avanza() {
  tok = getToken();
}
void mangia(enum token t) {
  if (tok == t)
    avanza();
  else
    error();
}
void S() {
  if (tok == BEGIN) {
    mangia(BEGIN); S(); L();
  }
  else if (tok == PRINT) {
    mangia(PRINT); E();
  }
  else
    error();
}
void L() {
  if (tok == END) {
    mangia(END);
  }
  else if (tok == SEMI) {
    mangia(SEMI); S(); L();
  }
  else
   error();
}
void E() {
  mangia(NUM);
}
```

L'albero sintattico relativo é construito a partire dal sembolo iniziale della grammatica ed espandendo sempre il non terminale più a sinistra (*leftmost*). Il parsing a discesa ricorsiva funziona soltanto quando:

1. leggendo il primo simbolo in input, si riesce a conprendere la produzione da applicare. Nel caso, nella grammtica considerata, ci siano due o più produzioni, relative ad un certo non terminale A, inizianti tutte e due con lo stesso simbolo si applica la *fattorizzazione a sinistra* della produzione. Ovvero, supponiamo che alla grammatica considerata precedentemente si aggiunge la produzione $E \rightarrow num + num$; questa nuova produzione entra in "collisione" con $E \rightarrow num$. La fattorizzazione a sinistra sposta le parti non in comune in un nuovo simbolo, chiamiamolo N, in modo che $E \rightarrow num\, N$ e $N \rightarrow +num \,|\, \epsilon$.

2. non ci sia, in nessuna produzione, la *ricorsione sinistra*, ovvero tutte le produzione **devono** iniziare per un simbolo terminale. Nel caso si presentino alcune ricorsioni sinistre nella grammatica, si può seguire l'algo-

ritmo 3, che ne causa la sostituzione con ricorsioni destre, queste ultime ammissibili.

Algorithm 3 Eliminazione della ricorsione sinistra

Input. Una grammatica G senza cicli o ϵ-produzioni.
Output. Una grammatica equivalente senza ricorsione sinistra.
Metodo. Sistemiamo i non terminali secondo un ordine: $A_1, A_2, \ldots, A_n$.

for i:=1 **to** n **do begin**
  for j:=1 **to** i-1 **do begin**
    rimpiazza ogni produzione della forma $A_i \rightarrow A_j\gamma$ con le produzioni $A \rightarrow \delta_1\gamma|\delta_2\gamma|\ldots|\delta_k\gamma$, dove $A_j \rightarrow \delta_1|\delta_2|\ldots\delta_k$ sono tutte le produzioni correnti di A_j.
  end
end

Per quel che riguarda la rilevazione di errori, ci sono due modi per trattare la situazione di errore: o si *solleva* immediatamente l'errore, fermando il parser (poco utile al programmatore), o si *segnala* l'errore, continuando il parsing alla ricerca di altri eventuali errori.

3.2.2 Parsing top-down LL(1)

Il termine *LL(1)* ha il seguente significato: la prima L sottende che l'input é analizzato da sinistra verso destra; la seconda L significa che il parser deriva una derivazione *leftmost* per la stringa di input; il numero *1* descrive che l'algoritmo utilizza soltanto un solo simbolo dell'input per risolvare le scelte del parser (ci sono varianti con k simboli). Per definizione, una grammatica *LL(1)* non é ambigua.

Mentre nei parser a discesa ricorsiva l'analisi sintattica viene effettuata attraverso una cascata di chiamate ricorsive, nel parser LL(1) la *pila* delle chiamate ricorsive viene esplicitata, non facendo più uso della ricorsione. Per la costruzione della tabella finale del parser, vengono usate due funzioni, dette *First* e *Follow*.

3.2.2.1 Funzione First

Se α é una stringa di simboli grammaticali, $First(\alpha)$ é l'insieme dei terminali che iniziano la stringa derivata da α. Se inoltre $\alpha \rightarrow^* \epsilon$, allora anche $\epsilon \in First(\alpha)$. Per calcolare $First(X)$ per ogni simbolo grammaticale X (terminale o non terminale), bisogna reiterare le seguenti 4 regole per ogni terminale finchè nessun'altro simbolo venga aggiunto a nessun *First*:

1. se X é un terminale o ϵ allora $First(X) = \{X\}$.
2. se $X \rightarrow \epsilon$ é una produzione allora $\epsilon \in First(X)$.

3. se X é un non terminale e $X \rightarrow Y_1 Y_2 \dots Y_k$ é una produzione, allora $First(Y_1) - \{\epsilon\} \in First(X)$. Poi:

```
i = 1;
while i ≤ k
    if ε ∈ First(Y_i)
        if i ≠ k
            aggiungi First(Y_{i+1}) − {ε} a First(X);
        else
            aggiungi ε a First(X);
        i++;
    else
        break;
```

4. se γ é una srtinga di terminali e non terminali $\gamma = X_1 X_2 \dots X_k$ allora $First(X_1) - \{\epsilon\} \in First(\gamma)$. Poi:

```
i = 1;
while i ≤ k
    if ε ∈ First(X_i)
        if i ≠ k
            aggiungi First(X_{i+1}) − {ε} a First(γ);
        else
            aggiungi ε a First(γ);
        i++;
    else
        break;
```

Viene definito *nullable* un non terminale X se $X \rightarrow \epsilon$, ovvero se é possibile derivare la stringa vuota da esso. Inoltre, un non terminale X é *nullable* **se e solo se** $\epsilon \in First(X)$.

3.2.2.2 Funzione Follow

Dato un non terminale X, $Follow(X)$ é l'insieme dei simboli terminali, eventualmente con *$*, definito dalle seguenti tre regole:

1. se X é un simbolo iniziale allora $\$ \in Follow(X)$.
2. se c'é una produzione $A \rightarrow \alpha X \gamma$ allora $First(\gamma) - \{\epsilon\} \in Follow(X)$.
3. se c'é una produzione $A \rightarrow \alpha X \gamma$ per cui $\epsilon \in First(\gamma)$ allora $Follow(A) \in Follow(X)$.

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

3.2.2.3 Tabella LL(1)

La tabella per una grammatica *LL(1)* é data dal prodotto cartesiano tra i non terminali (righe) e i terminali (colonne) della grammatica presa in considerazione. La sua costruzione é data da due semplici regole:

1. per ogni regola $X \rightarrow \gamma$ della grammatica G, si inserisce nella casella *(X,t)* la regola $X \rightarrow \gamma$ per ogni t tale che $t \in First(\gamma) - \{\epsilon\}$.
2. per ogni regola $X \rightarrow \gamma$ della grammatica G, per cui $\epsilon \in First(\gamma)$, si inserisce nella casella *(X,t)* la regola $X \rightarrow \gamma$ per ogni t tale che $t \in Follow(\gamma)$.

Per fare un esempio pratico, consideriamo la seguente grammatica (S simbolo iniziale):

$S \rightarrow int\, O\, int \mid real$
$O \rightarrow + S + \mid *$

Il *First* e il *Follow* di questa grammatica sono i seguenti:

First	
S	{int,real}
O	{+,*}
int	{int}
real	{real}
+	{+}
*	{*}

Follow	
S	{$,+}
O	{int}

La tabella risultante per la grammatica considerata é:

	int	**real**	+	*
S	$S \rightarrow int\, O\, int$	$S \rightarrow real$		
O			$O \rightarrow + S +$	$O \rightarrow *$

Una grammatica che genera una tabella con più di un elemento in corrispondenza di qualche *(X,t)* é ambigua e di conseguenza non é *LL(1)*. Le grammatiche ambigue corrispondenti sono dette *LL(K)*.

Come per i parser a discesa ricorsiva, trattati nella sezione 3.2.1, anche il parsing LL(1) é affetto dai problemi di *ricorsione sinistra* e *fattorizzazione a sinistra*. Le tecniche usate per la risoluzione di questi problemi sono le stesse di quelle trattate nella sezione 3.2.1 relativa ai parser a discesa ricorsiva.

3.3 Parsing bottom-up

In questa sezione verrà introdotta analisi sintattica *bottom-up*, conosciuta anche come *shift-reduce parsing*. Questa tecnica ha il compito di costruire un albero sintattico per una stringa di input partendo dalle foglie (bottom) e risalendo verso la radice (up). Si può pensare questo processo come una *riduzione* di una stringa w al simbolo iniziale della grammatica. Ad ogni passo di riduzione, se c'é una sottostringa che fa *match* con la parte destra di una produzione grammaticale é sostituita dal simbolo a sinistra della stessa produzione e, se ad ogni passo le sottostringhe vengono scelte in maniera opportuna, viene prodotta una derivazione *rightmost* rovesciata.

Una tecnica di analisi sintattica bottom-up, usata per fare il parsing di gran parte delle grammatica libere dal contesto, é la *LR(k)*: la L indica che l'input é analizzato da sinistra verso destra; la R indica che nel processo di parsing viene prodotta una derivazione *rightmost*; il numero k indica quanti simboli di *lookahead*[1] sono usati. I parser LR sono usati per varie ragioni, ad esempio possono essere costruiti per riconoscere tutti i linguaggi di programmazione castruiti a partire da grammatiche libere dal contesto.

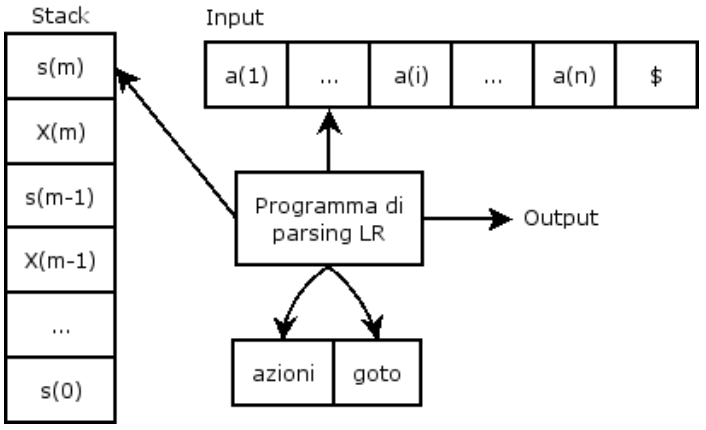


Figura 3.1: Modello per i parser LR

Uno schema di un parser LR é quello in figura 3.1: esso consiste di un input, un output, uno stack, un *programma di parsing* e una *tabella di parsing* divisa in due parti (*azioni* e *goto*). Il programma di parsing é lo stesso per ogni parsing LR; quello che cambia tra i vari parsing é solo la tabella di parsing. Il programma di parsing legge, uno per volta, dei caratteri da un buffer di input ed usa lo stack per memorizzare stringhe della forma $s_0X_1s_1X_2s_2 \ldots X_ms_m$, dove s_0 é lo stato iniziale, l'unico presente all'inizio della computazione sullo stack, mentre s_m é lo stato in cima allo stack. Ogni X_i é un simbolo grammaticale mentre ogni s_i é un simbolo che identifica uno *stato*. Ogni stato descrive l'informazione contenuta nello stack prima di esso; la combinazione tra lo stato in cima allo stack e il simbolo di input corrente viene usata per indicizzare la tabella di parsing e per determinare le decisioni di *shift-reduce* del parser.

Come già detto, la tabella di parsing é composta da due parti. Per la parte relativa alle *azioni* all'interno della tabella, quelle possibili sono quattro e sono calcolabili grazie alla funzione *azione*:

azioni

1. l'azione di *shift*, dove il prossimo simbolo di input viene spostato in cima allo stack.

2. l'azione di *reduce*, dove il parser scegli la h-esima[2] regola grammaticale $X \to \alpha_1 \ldots \alpha_k$, estrae (eliminandola, equivale ad un'azione di *pop*) la sequenza $\alpha_1 s_{h+1} \ldots \alpha_k s_{h+k}$ dallo stack, calcola il prossimo stato s_m a partire

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

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

dallo stato s_h in testa alla pila, dal simbolo X della regola grammaticale e dai k simboli (*lookahead*) all'inizio della stringa di input, e inserisce sullo stack Xs_m.

3. l'azione di *accept*, dove il parsing segnala di aver terminato con successo.
4. l'azione di *error*, dove il parsing segnala di aver terminato con un fallimento.

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

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

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

Il simbolo *$* indica **sempre** la fine della stringa. A tal scopo, se le produzioni di una certa grammatica non lo prevedono, bisogna aggiungere la produzione $S' \rightarrow S\$$, con S' nuovo simbolo iniziale della grammatica al posto di S. Questa configurazione rappresenta la stringa

$$X_1 X_2 \dots X_m a_i a_{a+1} \dots a_n \$$$

L'algoritmo 4 fornisce un metodo per fare parsing LR di una stringa ricevuta in input. Per calcolare l'ulteriore tabella fornita come input bisogna chiarire prima alcuni concetti.

3.3.1 Elementi LR(0), automi LR(0) e tabella *azione-goto*

elemento LR(0) Un *elemento LR(0)* di una grammatica G é una produzione di G con un punto nella parte destra, della forma $A \rightarrow \alpha.\beta$. Ad esempio, se $A \rightarrow XYZ$ é una produzione della grammatica, $A \rightarrow .XYZ$, $A \rightarrow X.YZ$, $A \rightarrow XY.Z$, $A \rightarrow XYZ.$ sono elementi LR(0). Una produzione come $A \rightarrow \epsilon$ genera l'elemento $A \rightarrow .$.

automa LR(0) Al fine di generare la tabella si rende necessaria la costruzione di un *automa LR(0)* utilizzando gli elementi LR(0) sopra descritti. Esistono due metodi principali, che verranno illustrati, per costruire l'automa: uno crea prima un automa non deterministico, utilizzando opportune regole, per poi passare, grazie alle regole standard, all'automa deterministico (vedere a tal proposito la sezione 2.3.2); l'altro crea direttamente un automa deterministico appoggiandosi alla funzione *chiusura* e di *goto*.

3.3.1.1 NFA e DFA con elementi LR(0)

Gli stati dell'automa NFA sono tutti gli elementi LR(0) creabili a partire dalla grammatica G considerata. Le transazioni dell'automa sono definite come segue: si consideri l'elemento $A \rightarrow \alpha.X\beta$. Allora esiste una trasizione etichettata X dallo stato $A \rightarrow \alpha.X\beta$ allo stato $A \rightarrow \alpha X.\beta$. In più, solo se X é un **non terminale**, a cui é associata una o più produzioni $X \rightarrow \gamma$, esistono delle transizioni etichettate ϵ dallo stato $A \rightarrow \alpha.X\beta$ agli stati $X \rightarrow .\gamma$.

Si possono effettuare le seguenti osservazioni:

Algorithm 4 Algoritmo di parsing LR

Input. Una stringa di input w e una tabella di parsing con le funzioni *azione* e *goto* per una grammatica G.
Output. Se w é il L_G, uno "scorrimento" bottom-up per w, altrimenti una segnalazione di errore.
Metodo. Inizialmente il parser ha il solo stato s_0(stato iniziale) sullo stack e $w\$$ sul buffer di input.

si fa puntare *ip* al primo simbolo di $w\$$;
repeat forever
  estraggo s stato in cima allo stack e a simbolo puntato da *ip*;
  if $azione[s, a] = shift\, s'$
    faccio una *push* di a seguita da una *push* di s' sullo stack;
    avanzo *ip* al prossimo simbolo in input;
  else if $azione[s, a] = reduce\, A \rightarrow \beta$
    faccio una *pop* di $2 * |\beta|$ simboli dello stack; //sia s' il nuovo stato in cima allo stack
    faccio una *push* di A seguita da una *push* di $goto[s', A]$ sullo stack;
    ritorno la produzione $A \rightarrow \beta$;
  else if $action[s, a] = accept$
    ritorno esito positivo;
  else
    ritorno errore;

- se X é un simbolo **terminale**, questa transizione corrisponde a fare uno *shift* del simbolo sull'input in cima allo stack.

- se X é un simbolo **non terminale**, questa transizione corrisponde a mettere X in testa alla pila. X, essendo non terminale, non é presente nell'input ma sicuramente sull'input vi sarà presente una stringa derivabile da X stesso. Per riconoscere questa stringa occorre ridursi al riconoscimento di un elemento LR(0) del tipo $X \rightarrow .\gamma$. In poche parole, in questo caso si é di fronte ad una *reduce*.

Gli stati iniziali dell'automa sono tutti quegli stati del tipo $S \rightarrow .\alpha\$$, dove S é il simbolo iniziale della grammatica. Nel caso in cui ce ne sia più di uno e/o la grammatica non preveda il terminatore *$*, si aumenta la grammatica con la produzione $S' \rightarrow S\$$, dove S' é un nuovo non terminale e, di conseguenza, il nuovo simbolo iniziale che fornisce, come unico stato iniziale, $S' \rightarrow .S\$$.

Gli stati finali non sono presenti perchè l'obiettivo dell'automa LR(0) é di ricordare lo stato del parser e non di accettare stringhe. Sarà il parser stesso ad accettare la stringa quando sulla pila c'é l' stato iniziale s_0 e l'input contiene solamente *$*.

Ottenuto l'automa non deterministico, si può ottenere quello deterministico applicando semplicemente il metodo descritto nella sezione 2.3.2.

3.3.1.2 Costruzione di un DFA diretta

Per costruire un DFA direttamente, ovvero senza passare per un NFA, bisogna prima descrivere il comportamento di due operazioni: la *chiusura* e il *goto*.

chiusura L'operazione di *chiusura* funziona nel seguente modo. Se E é un insieme di elementi per una grammatica G, allora *chiusura(E)* é l'insieme di elementi costruiti a partire da E attraverso le seguenti due regole:

1. Inizialmente, tutti gli elementi in E sono aggiunti a *chiusura(E)*.
2. Se $A \to \alpha.X\beta$ é in *chiusura(E)* e $X \to \gamma$ é un produzione, si aggiunge l'elemento $X \to .\gamma$ a E, se questo non é già presente. Questa regola si applica finchè nessun nuovo elemento viene aggiunto da *chiusura(E)*.

goto L'operazione di *goto*, che prende come argomenti un insieme E di elementi e un simbolo grammaticale X, é definita come segue:

$$goto(E, X) = chiusura(A \to \alpha X.\beta) \forall A \to \alpha.X\beta \in E$$

A questo punto, con l'algoritmo 5, si possono ottenere tutti gli stati con le relative transizioni.

Algorithm 5 Costruzione diretta di un DFA per una grammatica G

Input. Una collezione di di insiemi di elementi LR(0) derivati dalla grammatica G. Deve essere presente un solo stato iniziale (vedi la sezione 3.3.1.1).
Output. Un DFA relativo alla grammatica G, definito da un insieme degli stati C e un insieme di transizioni T (definita come $T[stato_{partenza}, SimboloGrammaticale] = stato_{arrivo}$).
Metodo. La costruzione é definita dal seguente algoritmo:

```
begin
    C := {chiusura({[S' → .S]})};
    repeat
        per ogni insieme di elementi E ∈ C
            per ogni simbolo grammaticale X tale
            che goto(E, X) non sia vuoto
              if goto(E, X) ∉ C
                 aggiungi goto(E, X) a C;
              T[E, X] = goto(E, X);
    until non si possono generare più elementi da inserire in
    C o nuove transizioni
end
```

Sia S il simbolo iniziale della grammatica e sia S' il nuovo simbolo creato per la costruzione dell'automa. Lo **stato iniziale** dell'automa LR(0) é $chiusura(\{S' \to .S\})$.

Per quel che riguarda l'algoritmo 5, ancora un'osservazione: $goto(E, X)$ non é vuoto quando esistono in E degli elementi definiti come $A \rightarrow \alpha.X\beta$. La riga "per ogni simbolo grammaticale X tale che $goto(E, X)$ non sia vuoto" può essere sostituita con "per ogni simbolo grammaticale X che compare in elementi di E formati da $A \rightarrow \alpha.X\beta$".

3.3.1.3 Costruzione della tabella LR(0)

La tabella ha come righe il numero degli stati e come colonne i simboli della grammatica disposti nel seguente ordine: terminali, *$*, non terminali. Per ogni riga della tabella:

- azioni *shift*: sia k lo stato dell'automa (riga) considerato. Per ogni arco uscente dallo stato k, etichettato con x simbolo **terminale**, si inserisce, nella posizione (k, x) della tabella lo stato di arrivo dell'arco $s_{T[k,x]}$.
- azione *goto*: sia k lo stato dell'automa (riga) considerato. Per ogni arco uscente dallo stato k, etichettato con X simbolo **non terminale**, si inserisce, nella posizione (k, X) della tabella lo stato di arrivo dell'arco $g_{T[k,X]}$.
- azioni *reduce*: ogni stato contenente un elemento LR(0) del tipo $X \rightarrow \gamma.$ deve avere una operazione di riduzione, indicata con r_i, dove i identifica la regola $X \rightarrow \gamma$ presa in considerazione (tutte le regole grammaticali devono essere numerate). Poichè il parser é LR(0) la riduzione deve essere fatta per ogni terminale della grammatica.
- azione *acc*: l'operazione di $reduce\, S' \rightarrow \alpha.\$$ quando in testa alla pila c'é *$* equivale ad accettazione. Di conseguenza, supponendo che l'elemento sia nello stato k, **non** bisogna effettuare la *reduce* ma inserire, in posizione $(k, \$)$, un *acc*.
- azione *error*: tutte le celle della tabella vuote sottintendono una situazione di errore.

Ora, facciamo un esempio. Consideriamo la grammatica G_{prova} con le seguenti produzioni:

1. $S \rightarrow E\$$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$

Nella figura 3.2 si può vedere il DFA relativo, ricavato utilizzando le regole spegate nella sezione 3.3.1.2, mentre la tabella 3.1 é stata ricavata seguendo le regole espresse precedentemente.

Una grammatica é LR(0) se, una volta costruita la tabella, in ogni cella non ci sono conflitti. La tabella 3.1, di conseguenza, é relativa ad una grammatica **non** LR(0). I conflitti possibili sono di due tipi: o *shift-reduce*, quando non si sa se shiftare o ridurre, o *reduce-reduce*, se ci sono più riduzioni possibili in uno stato. Il conflitto *shift-shift* non é considerato perchè, siccome l'automa é deterministico, da ogni stato non possono esserci due archi etichettati con lo

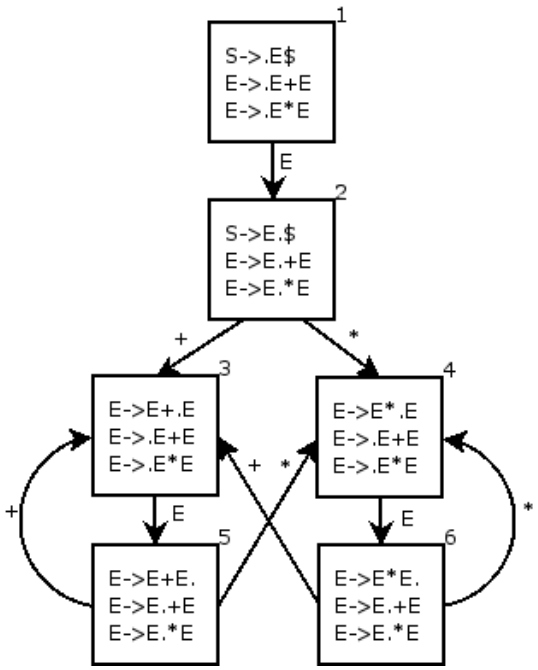


Figura 3.2: Costruzione del DFA data la grammatica G_{prova}

stesso simbolo e di conseguenza, siccome a maggior ragione la regola vale per i terminali della grammatica, in un qualunque stato non possono esistere due *shift* diversi per uno stesso simbolo.

3.3.2 Parser Simple LR(1) {SLR(1)}

Gli elementi utilizzati dal *parser SLR(1)* sono ancora quelli LR(0) definiti nella sezione 3.3.1. Le varie costruzioni dell'automa, definite nelle sezioni 3.3.1.1 e 3.3.1.2, sono identiche a quelle usate per i parser LR(0). L'unica cosa che cambia é la costruzione della tabella in relazione alla sola azione *reduce*. Per comodità si riportano qui di seguito tutte le regole relative alla costruzione della tabella:

- azioni *shift*: sia k lo stato dell'automa (riga) considerato. Per ogni arco uscente dallo stato k, etichettato con x simbolo **terminale**, si inserisce, nella posizione (k, x) della tabella lo stato di arrivo dell'arco $s_{T[k,x]}$.

- azione *goto*: sia k lo stato dell'automa (riga) considerato. Per ogni arco uscente dallo stato k, etichettato con X simbolo **non terminale**, si inserisce, nella posizione (k, X) della tabella lo stato di arrivo dell'arco $g_{T[k,X]}$.

- **azioni *reduce***: ogni stato contenente un elemento LR(0) del tipo $X \rightarrow \gamma$. deve avere una operazione di riduzione, indicata con r_i, dove i identifica la regola $X \rightarrow \gamma$ presa in considerazione (tutte le regole grammaticali devono essere numerate). L'operazione di riduzione, per uno stato k, deve essere

stati/simboli	+	*	**$**	**S**	**E**
1					g2
2	s3	s4	acc		
3					g5
4					g6
5	s3,r2	s4,r2	r2		
6	s3,r3	s4,r3	r3		

Tabella 3.1: Tabella relativa alla costruzione del DFA data la grammatica G_{prova}

fatta in ogni cella $(k, t)\, \forall t \in Follow(X)$, dove t é un simbolo terminale e $Follow(X)$ é la funzione definita nella sezione 3.2.2.2.

- azione *acc*: l'operazione di $reduce\, S' \rightarrow \alpha.\$$ quando in testa alla pila c'é *$* equivale ad accettazione. Di conseguenza, supponendo che l'elemento sia nello stato k, **non** bisogna effettuare la *reduce* ma inserire, in posizione $(k, \$)$, un *acc*.

- azione *error*: tutte le celle della tabella vuote sottintendono una situazione di errore.

Come per le LR(0), le grammatiche per cui la tabella di analisi prodotta dal parser SLR(1) non contiene conflitti in nessun cella (non ha ambiguità) sono dette *grammatiche SLR(1)*.

3.3.3 Parser LR(1)

Il *parser LR(1)* consente di aggirare molte ambiguità create dai parser SLR(1), al prezzo di una complessità più elevata dell'algoritmo. In realtà, in pratica é poco usato a causa, oltre che per la sua complessità, anche perchè esiste un parser, l'*LALR(1)* analizzato nella sezione 3.3.4, più semplice ed efficiente. Al contrario dell'SLR(1), che utilizzava il *lookahead* dopo la costruzione del DFA (infatti il *Follow* veniva applicato nella costruzione della tabella), nel parser LR(1) il *lookahead* é utilizzato durante la costruzione dell'automa.

elemento LR(1)

Un *elemento LR(1)* di una grammatica G é una produzione di G con un punto nella parte destra seguita da un token t (il *lookahead*), della forma $A \rightarrow \alpha.\beta, t$. Ad esempio, se $A \rightarrow XYZ, t$ é una produzione della grammatica, $A \rightarrow .XYZ, t$, $A \rightarrow X.YZ, t$, $A \rightarrow XY.Z, t$, $A \rightarrow XYZ, t.$ sono elementi LR(1). Una produzione come $A \rightarrow \epsilon$ genera l'elemento $A \rightarrow ., t$.

automa LR(1)

Al fine di generare la tabella si rende necessaria la costruzione di un *automa LR(1)* utilizzando gli elementi LR(1) sopra descritti. Esistono due metodi principali, che verranno illustrati, per costruire l'automa: uno crea prima un automa non deterministico, utilizzando opportune regole, per poi passare, grazie alle regole standard, all'automa deterministico (vedere a tal proposito la sezione 2.3.2); l'altro crea direttamente un automa deterministico appoggiandosi alla funzione *chiusura* e di *goto*, definite in maniera diversa da quelle elencate nella sezione 3.3.1.2 relative agli automi LR(0) e SLR(1).

3.3.3.1 NFA e DFA con elementi LR(1)

Gli stati dell'automa NFA sono tutti gli elementi LR(1) creabili a partire dalla grammatica G considerata. Le transazioni dell'automa sono definite come segue: si consideri l'elemento $A \to \alpha.X\beta, t$. Allora esiste una trasizione etichettata X dallo stato $A \to \alpha.X\beta, t$ allo stato $A \to \alpha X.\beta, t$. In più, solo se X é un **non terminale**, a cui é associata una o più produzioni $X \to \gamma$, esistono delle transizioni etichettate ϵ dallo stato $A \to \alpha.X\beta, t$ agli stati $X \to .\gamma, t'$ per ogni produzione $X \to \gamma$ e per ogni $t' \in First(\beta t)$, dove *First* é la funzione definita nella sezione 3.2.2.1.

Gli stati iniziali dell'automa sono tutti quegli stati del tipo $S \to .\alpha, t$, dove S é il simbolo iniziale della grammatica. Nel caso in cui ce ne sia più di uno e/o la grammatica non preveda il terminatore *$*, si aumenta la grammatica con la produzione $S' \to S\$$, dove S' é un nuovo non terminale , di conseguenza, il nuovo simbolo iniziale che fornisce, come unico stato iniziale, $S' \to .S, \$$.

Gli stati finali non sono presenti perchè l'obiettivo dell'automa LR(1) é di ricordare lo stato del parser e non di accettare stringhe. Sarà il parser stesso ad accettare la stringa quando sulla pila c'é l' stato iniziale s_0 e l'input contiene solamente *$*.

Ottenuto l'automa non deterministico, si può ottenere quello deterministico applicando semplicemente il metodo descritto nella sezione 2.3.2.

3.3.3.2 Costruzione di un DFA per un parser LR(1) diretta

Per costruire un DFA direttamente, ovvero senza passare per un NFA, bisogna prima descrivere il comportamento di due operazioni: la *chiusura* e il *goto*.

chiusura

L'operazione di *chiusura* funziona nel seguente modo. Se E é un insieme di elementi per una grammatica G, allora *chiusura(E)* é l'insieme di elementi costruiti a partire da E attraverso l'algoritmo 6.

Algorithm 6 Chiusura per il parser LR(1)

Input. Un insieme E di elementi LR(1) relativi ad una grammatica G_E.
Output. Un insieme C_E contenente le chiusure degli elementi presenti in E.
Metodo. Per calcolare $chiusura(E)$ si segue il seguente algoritmo:

begin
  inserisco in C_E tutti gli elementi di E;
  repeat
    per ogni elemento $A \to \alpha.X\beta, t \in C_E$
      per ogni produzione $X \to \gamma \in G_E$
        per ogni terminale $t' \in First(\beta t)$
          if l'elemento $X \to .\gamma, t' \notin C_E$
          then aggiungo $X \to .\gamma, t'$ a C_E;
  until non si possono generare più elementi da inserire in C_E
  return C_E
end

L'operazione di *goto*, che prende come argomenti un insieme E di elementi e un simbolo grammaticale X, é definita come segue: **goto**

$$goto(E, X) = chiusura(A \rightarrow \alpha X.\beta, t) \, \forall A \rightarrow \alpha.X\beta, t \in E$$

A questo punto, con l'algoritmo 7, si possono ottenere tutti gli stati con le relative transizioni.

Algorithm 7 Costruzione diretta di un DFA di un parser LR(1) per una grammatica G

Input. Una collezione di di insiemi di elementi LR(1) derivati dalla grammatica G. Deve essere presente un solo stato iniziale (vedi la sezione 3.3.3.1).
Output. Un DFA relativo alla grammatica G, definito da un insieme degli stati C e un insieme di transizioni T (definita come $T[stato_{partenza}, SimboloGrammaticale] = stato_{arrivo}$).
Metodo. La costruzione é definita dal seguente algoritmo:

begin
  $C := \{chiusura(\{[S' \rightarrow .S, \$]\})\}$;
  repeat
    per ogni insieme di elementi $E \in C$
      per ogni simbolo grammaticale X tale che $goto(E, X)$ non sia vuoto
        if $goto(E, X) \notin C$
          aggiungi $goto(E, X)$ a C;
        $T[E, X] = goto(E, X)$;
  until non si possono generare più elementi da inserire in C o nuove transizioni
end

Sia S il simbolo iniziale della grammatica e sia S' il nuovo simbolo creato per la costruzione dell'automa. Lo **stato iniziale** dell'automa LR(1) é $chiusura(\{S' \rightarrow .S, \$\})$.

Per quel che riguarda l'algoritmo 7, ancora un'osservazione: $goto(E, X)$ non é vuoto quando esistono in E degli elementi definiti come $A \rightarrow \alpha.X\beta, t$. La riga "per ogni simbolo grammaticale X tale che $goto(E, X)$ non sia vuoto" può essere sostituita con "per ogni simbolo grammaticale X che compare in elementi di E formati da $A \rightarrow \alpha.X\beta, t$".

3.3.3.3 Costruzione della tabella LR(1)

La tabella ha come righe il numero degli stati e come colonne i simboli della grammatica disposti nel seguente ordine: terminali, *$*, non terminali. Per ogni riga della tabella:

- azioni *shift*: sia k lo stato dell'automa (riga) considerato. Per ogni arco uscente dallo stato k, etichettato con x simbolo **terminale**, si inserisce, nella posizione (k, x) della tabella lo stato di arrivo dell'arco $s_{T[k,x]}$.

- azione *goto*: sia k lo stato dell'automa (riga) considerato. Per ogni arco uscente dallo stato k, etichettato con X simbolo **non terminale**, si inserisce, nella posizione (k, X) della tabella lo stato di arrivo dell'arco $g_{T[k,X]}$.

- azioni *reduce*: ogni stato contenente un elemento LR(1) del tipo $X \rightarrow \gamma., t$ deve avere una operazione di riduzione, indicata con r_i, dove i identifica la regola $X \rightarrow \gamma$ presa in considerazione (tutte le regole grammaticali devono essere numerate). Poichè il parser é LR(1) la riduzione deve essere soltanto quando il simbolo in testa all'input sarà *t*.

- azione *acc*: l'operazione di $reduce\, S' \rightarrow \alpha., \$$ quando in testa alla pila c'é *$* equivale ad accettazione. Di conseguenza, supponendo che l'elemento sia nello stato *k*, **non** bisogna effettuare la *reduce* ma inserire, in posizione $(k, \$)$, un *acc*.

- azione *error*: tutte le celle della tabella vuote sottintendono una situazione di errore.

Le grammatiche per cui la tabella di analisi prodotta dal parser LR(1) non contiene conflitti in nessun cella (non ha ambiguità) sono dette *grammatiche LR(1)*.

3.3.4 Il parser LookAhead LR(1) {LARL(1)}

L'ultimo parser che verrà discusso in questo capitolo é relativo al *parser LALR(1)* (*lookahead-LR*). Questo parser é molto usato in pratica perchè le tabelle da lui ottenute sono considerevolmente più piccole di quelle date come risultato del parser LR(1). La maggiorparte dei linguaggi di programmazione in circolazione sono ottenuti da grammatiche LALR. La stessa cosa vale per le grammatiche SLR, anche se ci sono alcuni costrutti che non é facile rendere al meglio utilizzando grammatiche SLR.

Per fare un confronto, le tabelle SLR e LALR per una stessa grammatica G hanno lo stesso numero di stati mentre la tabella costruita da un parser LR, basandosi anch'esso sulla grammatica G, ha molti più stati, pur riferendosi allo stesso linguaggio. In "soldoni", le tabelle di analisi LR(1) sono di solito 2 ordini di grandezza maggiori di quelle SLR(1): se una tabella SLR(1) di un linguaggio di programmazione é circa 10KB, una tabella LR(1) per lo stesso linguaggio é intorno ai MB, utilizzando il doppio della memoria per la costruzione.

Il parser LALR(1) identifica gli stati **uguali a meno del *lookahead*** creati con l'algoritmo del parser LR(1) (vedere a tal proposito la sezione 3.3.3.2), unisce fra loro questi stati fondendone i *lookahead*, con l'obiettico di ottenere un DFA LR(1) simile al DFA LR(0).

Gli *elementi LALR(1)* hanno la forma

$$A \rightarrow \alpha.\beta, \{t_1, \ldots, t_n\}$$

dove $A \rightarrow \alpha.\beta$ é un elemento LR(0) e $\{t_1, \ldots, t_n\}$ é un insieme di token (insieme di *lookahead*)[3].

[3]Gli insiemi di *lookahead* sono di solito più piccoli degli insiemi $Follow$. E' per questo che i parser LALR(1) sono migliori degli SLR(1), pur avendo la stessa dimensione di quelli LR(0).

Nella sezione 3.3.4.1 viene presentata la costruzione della tabella LALR(1) partendo dall'automa LR(1). E' ovvio che questo metodo, seppur semplice da applicare, é estremamente costoso a causa del passaggio obbligato relativo alla costruzione dell'automa LR(1) e sembrerebbe entrare in contrasto con quello detto all'inizio di questa sezione in relazione ai costi di costruzione. Nella realtà, l'algoritmo utilizzato, molto più complicato, evita il passaggio dall'automa LR(1), utilizzando, come punto di partenza, gli elementi LR(0).

3.3.4.1 Automa DFA e tabella LALR(1)

Per trovare l'automa DFA LALR(1), relativo ad una grammatica G, si segue l'algoritmo 8.

Algorithm 8 Costruzione dell'automa LALR(1)

Input. L'automa LR(1) per la grammatica G.
Output. L'automa LALR(1) per la grammatica G.
Metodo. Un **nucleo** di uno stato s dell'automa LR(1) é l'insieme degli elementi LR(0), ovvero gli elementi senza considerare il *lookahead*, di quello stato. In poche parole, il nucleo di uno stato dell'automa LR(1) é uno stato dell'automa LR(0). Si definiscono gli stati e le transizioni dell'automa LALR(1) come segue:

- uno *stato* dell'automa LALR(1) é definito
  1. prendendo tutti gli stati dell'automa LR(1) con lo stesso nucleo.
  2. fattorizzando gli elementi rispetto alla parte LR(0).
  3. considerando come *lookahead* l'unione dei *lookahead*.
- le *transizioni* dell'automa LALR(1) sono le stesse dell'automa LR(1). Se ci sono due stati, s_1 e s_2, del DFA LR(1) con lo stesso nucleo e c'é una transizione etichettata X che va da s_1 ad un'altro stato s_3, allora esiste s_4 tale che esiste una transizione etichettata X che va da s_2 a s_4, ed inoltre s_3 e s_4 hanno lo stesso nucleo.

Da come é costruito, si può osservare che l'automa LALR(1) é strutturalmente uguale a quello LR(0). La costruzione della relativa tabella é identica a quella indicata nella sezione 3.3.3.3 relativa ai parser LR(1). Se in nessuna delle celle della tabella LALR(1) così prodotta non compaiono conflitti, allora la grammatica é detta *grammatica LALR(1)*.

3.3.5 Grammatiche ambigue

Le grammatiche di molti linguaggi di programmazione possono avere alcune regole che generano casi di ambiguità grammaticale:

$$S \rightarrow if\ E\ then\ S\ else\ S$$
$$S \rightarrow if\ E\ then\ S$$
$$S \rightarrow \ldots$$

Le regole sopra elencate, per esempio, consentono di derivare la stringa "if *a* then if *b* then *s1* else *s2*". Questa stringa può essere analizzata in due modi possibili:

1. if *a* then {if *b* then *s1* else *s2*}
2. if *a* then {if *b* then *s1*} else *s2*

Solitamente, in tutti i linguaggi di programmazione, l'interpretazione corretta é quella fornita dall'analisi 1. Nella relativa tabella LR(1), costruita a partire da una grammatica del genere, si verrebbe a creare un conflitto *shift-reduce*. L'ambiguità può essere eliminata introducendo due non terminali, con l'aggiunta delle seguenti regole:

$S \rightarrow M$
$S \rightarrow U$
$M \rightarrow if\ E\ then\ M\ else\ M$
$M \rightarrow \ldots$
$U \rightarrow if\ E\ then\ S$
$U \rightarrow if\ E\ then\ M\ else\ U$

Invece di cambiare la grammatica come appena indicato, si potrebbe pensare di tollerare il conflitto *shift-reduce*, risolvendolo "a mano" a favore dello *shift* nella costruzione della tabella di analisi. Ovviamente le modifiche manuali vanno usate solo in casi estremi poichè, se in un parser LR(1) sonno presenti dei conflitti, spesso sono dovuti ad una scarsa attenzione in fase di progettazione della grammatica considerata.

3.3.6 Bison

La costruzione di un parser LR(1) o LALR(1) é basata su dei passaggi meccanici di facile automatizzazione. Gli esempi visti sono stati fatti con piccole grammatiche, che già da sole bastavano a generare un grosso quantitativo di stati. Se si considerassero le grammatiche utilizzate realmente nei linguaggi di programmazione attuali, diventerebbe impossibile calcolare "a mano" il relativo automa e la conseguente tabella, siano essi LR(1) o LALR(1). Per questo motivo, a causa della sua grande laboriosità, la costruzione del parser viene fatta fare da un generatore automatico.

Bison é uno di questi generatori. Il suo input é diviso in tre sezioni fondamentali:

```
%{
dichiarazioni C
%}
dichiarazioni di Bison
%%
regole grammaticali
%%
programmi
```

iarazioni di Bison La parte relativa alle *dichiarazioni di Bison* contiene la *lista dei simboli* (terminali, non terminali, simbolo iniziale), le regole di *associatività* degli operatori (utili per risolvere casì di ambiguità senza mettere mano la grammatica - vedere sezione 3.3.5), dichiarazione di *funzioni*. Queste linee sono caratterizzate da *parole chiave*:

%token indica la lista dei token del linguaggio
%start indica il non terminale iniziale
%left indica l'associatività a sinitra
%right indica l'associatività a destra
%nonassoc indica che certi operatori non possono essere usati in maniera associativa

La parte relativa alle *regole grammaticali* é formata da produzioni della forma ***regole grammaticali***

exp: exp PLUS exp {azione semantica}

a rappresentare la regola $exp \rightarrow exp + exp$ e dove l'*azione semantica* é un programma scritto in C che viene eseguito quando il parser effettua una *reduce* con questa regola.

La parte relativa al *programmi* può contenere del codice C. Spesso sono ***programmi*** inserite in questa parte la definizione di *yylex* o delle funzioni ausiliarie utilizzate nelle azioni semantiche delle regole grammaticali.

Portiamo ora, come esempio, il file di input per *Bison* in figura 3.3.

```
%{
#include<stdio.h>
int yylex();
void yyerror(char *s) {printf("parse error\n"); }
%}
%token ID WHILE BEGIN END DO IF THEN ELSE SEMI ASSIGN
%start prog
%%
prog      : stmlist
;
stmlist   : stm
          | stmlist SEMI stm
;
stm       : ID ASSIGN ID
          | WHILE ID DO stm
          | BEGIN ID DO stm
          | IF ID THEN stm
          | IF ID THEN stm ELSE stm
;
```

Figura 3.3: File di input per Bison

Memorizzando il codice in figura 3.3 nel fil *file.y* ed invocando *bison –verbose file.y* otteniamo il seguente warning: *file.y contains 1 shift/reduce conflict*. Con la chiamata appena effettuata, *Bison* ha generato due file: *file.tab.c*, un file C, e *file.output*, che descrive la tabella di analisi del parser. Analizzando quest'ultimo file ci si accorge, come discusso nella sezione 3.3.5, che il parser in relazione alla

regola $stm \rightarrow IF\ ID\ THEN\ stm$ tenta di fare una *reduce* mentre in relazione alla regola $stm \rightarrow IF\ ID\ THEN\ stm\ ELSE\ stm$ cerchi di fare uno *shift*.

Bison di default risolve i conflitti di tipi *shift-reduce* in favore dello *shift* mentre risolve i conflitti *reduce-reduce* in favore della rpima regola che si incontra nell'ordinamento delle regole. Di conseguenza, nell'esempio considerato, si può accettare il warning lasciando a *Bison* l'incombenza di risolverlo.

Per definizione, nessuna grammatica ambigua é LR(k). Ciò non toglie che le grammatiche ambigue sono utili, perchè semplici, se si riesce a trovare il modo di risolvere i conflitti, o inserendo nuove regole grammaticali o indicando *direttive di precedenza* opportune (ad esempio, * deve avere più precedenza di +). In un conflitto *shift-reduce* la **regola standard** seguita é di solito la seguente: una produzione prende la precedenza del simbolo terminale che compare più a destra. Quando la *reduce* e lo *shift* hanno la stessa precedenza allora si considerano le direttive definite nelle *dichiarazioni di Bison*, delle quali *%left* favorisce la *reduce*, *%right* favorisce lo *shift* e *%nonassoc* produce un messaggio di errore.

Invece di usare la regola standard, é possibile forzare la precedenza facendo seguire alla produzione interessata il tag *%prec <terminale>*. Si può vedere un esempio di applicazione di questo tag nella figura 3.4.

```
%token INT PLUS MINUS TIMES UMINUS
%start EXP
%left PLUS MINUS
%left TIMES
%left UMINUS    /*UMINUS non esiste come token, é fittizio*/
%%
exp     : INT
        | exp PLUS exp
        | exp MINUS exp
        | exp TIMES exp
        | MINUS exp %prec UMINUS
```

Figura 3.4: Precedenza "forzata" in *Bison*

Considerando l'estratto di input della figura 3.4, la *reduce* della produzione $exp \rightarrow MINUS\ exp$ ha precedenza su qualunque operazione di *shift*. Questo é ciò che realmente accade per il "-" unario utilizzato nei linguaggi di programmazione.

3.4 Error Recovery

Le tecniche di parsing viste finora, *top-down* (sezione 3.2) e *bottom-up* (sezione 3.3), bloccano l'analisi sintattica quando si verifica il primo errore. A questo comportamento é preferibile quello che permette di ritornare una lista completa di tutti gli errori presenti nel programma passato al compilatore. Per rendere possibile ciò, si utilizzano due tecniche principali:

1. *recovery utilizzando un token di errore*. Per descriverne il funzionamento, si consideri la seguente grammatica:

   $exp \rightarrow ID \mid exp + exp \mid (exps)$
   $exps \rightarrow exp \mid exps; exp$

Supponiamo che il riconoscimento di un'espressione possa produrre un errore. In questo caso, é possibile indicare al parser, nella sintassi, di trascurare tutti i simboli successivi a quell'errore finquando non incontra un *;* o una *)*. Ciò può essere fatto aumentando le produzioni della grammatica con

$$exp \rightarrow (error) \mid error; exp$$

dove *error* é considerato un simbolo terminale al quale le azioni di *shift* vengono applicate allo stesso modo degli altri simboli. In questo modo, man a mano che si individuano degli errori si può:

- effettuare un'azione semantica opportuna, come, per esempio, stampare la tipologia di errore.
- continuare con il parsing normale.

Il problema di questa tecnica é che si deve intervenire sulle produzioni grammaticali aggiungendone, opportunamente, alcune, il che comporta una modifica nella struttura della grammatica, cosa che, spesso, può essere **pericolosa.**

2. *recovery globale.* La seguente tecnica consente non solo di trovare un errore ma tenta anche di **ripararlo.** E' in grado, cioè, di trovare l'insieme minimale di inserzioni e cancellazioni che trafsormano la stringa in input in una stringa corretta anche se queste operazioni non riguardano la parte di stringa che ha generato l'errore. In più, questa tecnica **non** modifica la sintassi sorgente.

   Un algoritmo che implementa in maniera limitata il *recovery globale* é l'*algoritmo di Burke-Fisher* (9).

Algorithm 9 Tecnica di Burke-Fisher

Input. Una stringa relativa ad una grammatica.
Output. La stringa presa in input, eventualmente corretta in caso di riconoscimento di errori.
Metodo. Tenta ogni possibile inserimento/cancellazione/rimpiazzamento di token in ogni punto della stringa che non vada oltre K token precedenti, dove K di solito é uguale a 3 o a 4. Vengono utilizzate due pile: una contenente la configurazione della pila K token prima del token che viene letto (**pila vecchia**); l'altra contenete la pila più attuale (**pila corrente**). Quando un errore viene rilevato, l'algoritmo modifica uno degli ultimi K caratteri letti e ricomincia il *parsing* a partire dalla *pila vecchia*, modificando l'ultimo carattere letto e procedendo, via via, all'indietro.

Il problema principale dell'algoritmo 9, é in relazione alle *azioni semantiche* relative ad ogni *reduce*. Il problema viene risolto ritardando le *azioni semantiche* a quando il simbolo ridotto é spostato sulla *pila vecchia*. In questo caso, la riduzione diventa **permanente.**

3.5 Classificazione delle grammatiche

Tutte le grammatiche viste fin qui stanno, tra loro, in un rapporto di inclusione, che ha come massimo esponente la grammatica LR(k). Questo rapporto é descritto in modo chiaro e sintetico nella figura 3.5.

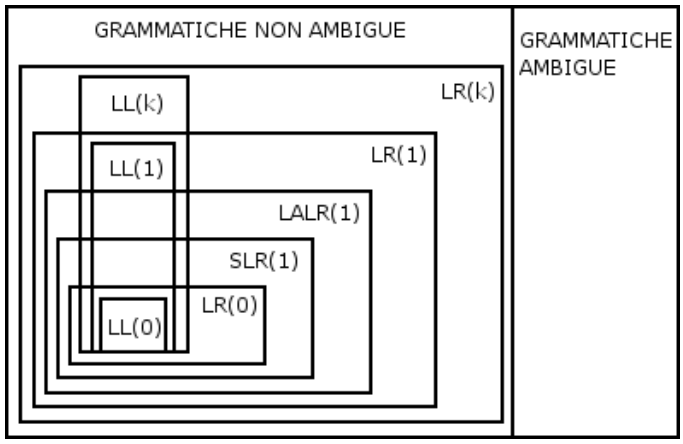


Figura 3.5: Inclusione tra grammatiche

Parte II

Dall'analisi semantica alla generazione del codice intermedio

Capitolo 4

Grammatiche con attributi

4.1 Attributi *sintetizzati* ed *ereditati*

Ci sono due notazioni principali per associare le regole semantiche alle produzioni di una grammatica:

1. la *definizione sintattica diretta*, che è una specifica di traduzione ad alto livello.

2. gli *schemi di traduzione*, che indicano l'ordine in cui le regole semantiche vanno valutate.

Una *definizione sintattica diretta* è una generalizzazione di una grammatica context-free in cui ogni simbolo grammaticale ha associato un insieme di attributi che è tipicamente diviso in due sottoinsiemi, quello degli *attributi sintetizzati* e quello degli *attributi ereditati* di un simbolo grammaticale. Il valore di un *attributo sintetizzato* di un nodo è calcolato a partire dal valore degli attributi dei nodi figli di quel nodo; il valore di un *attributo ereditato* di un nodo è calcolato dai valori degli attributi dei nodi fratelli e del nodo padre. Un attributo può rappresentare una qualsiasi cosa (una stringa, un intero, un tipo). Il valore di un attributo di un nodo dell'albero di parsing è definito attraverso la regola semantica associata con la produzione usata per questo nodo. La dipendenza tra gli attributi può essere rappresentata attraverso il *grafo delle dipendenze*, espresso in figura 4.1.

Nella *definizione sintattica diretta*, ogni produzione grammaticale $A \to \alpha$ è associata ad un insieme di regole semantiche della forma $b := f(c_1, c_2, \ldots, c_k)$ dove f è una funzione e rispettivamente

1. b è un attributo sintetizzato di A e $c_1, c_2, \ldots, c_k$ sono attributi appartenenti ai simboli grammaticali della produzione, oppure

2. b è un attributo ereditato da uno dei simboli grammaticali della parte destra della produzione e $c_1, c_2, \ldots, c_k$ sono attributi appartenenti ai simboli grammaticali della produzione.

In ogni caso si può dire che l'attributo b dipende dagli attributi $c_1, c_2, \ldots, c_k$.

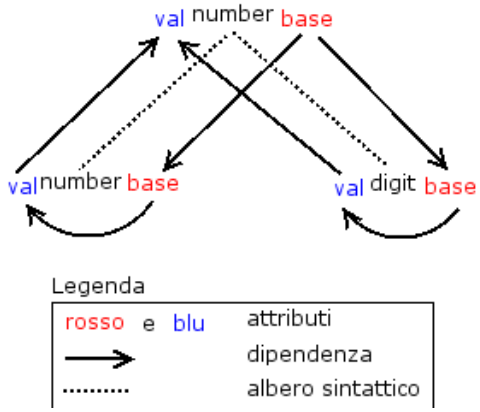


Figura 4.1: Dipendenza di attributi per la produzione $number \rightarrow number\ digit$

Una *grammatica con attributi* è una definizione sintattica diretta nella quale le funzioni nelle regole semantiche non possono avere effetti indesiderati (*side-effect*). Date le produzioni grammaticali seguenti

$L \rightarrow E\,\mathbf{n}$
$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow \mathbf{digit}$

la grammatica con attributi è definita come segue:

$L \rightarrow E\,\mathbf{n}$ {
  $print(E.val)$
}
$E \rightarrow E_1 + T$ {
  $E.val := E_1.val + T.val$
}
$E \rightarrow T$ {
  $E.val := T.val$
}
$T \rightarrow T_1 * F$ {
  $T.val := T_1.val \times F.val$
}
$T \rightarrow F$ {
  $T.val := F.val$
}
$F \rightarrow (E)$ {
  $F.val := E.val$
}
$F \rightarrow \mathbf{digit}$ {

$F.val :=$ **digit**.*lexval*
}

In questo esempio, tutto quello che è contenuto dentro le parentesi graffe rappresenta la regola semantica da seguire per quella produzione. In generale, se si considera la produzione della grammatica con attributi $A \rightarrow B\,C\,\{semanticaA\}$, tutte le azioni semantiche nei sottoalberi B e C sono eseguite **prima** di *semanticaA*. Questo non è completamente vero nel caso si consideri una grammatica con attributi sia sintetizzati sia ereditati. In questo caso, se un attributo b di un nodo in un albero di parsing dipende da un attributo c, allora la regola semantica associata a b per quel nodo deve essere valutata **dopo** la valutazione della regola semantica associata a c. Questa precedenza di valutazione sulla semantica può essere descritta visivamente creando il *grafo delle dipendenze*.

Inoltre, sempre in riferimento all'esempio appena affrentato, il token **digit** ha l'attributo sintetizzato *lexval* il cui valore viene assegnato in fase di analisi lessicale.

Attributi *sintetizzati*

Gli attributi sintetizzati sono usati molto di frequente in pratica. Una definizione sintattica diretta che usa attributi sintetizzati è detta *definizione con S-attributi*. Un albero di parsing per una definizione con S-attributi può essere compiuta attraverso una valutazione *bottom-up* delle regole semantiche per gli attributi di ogni nodo, risalendo dalle foglie alla radice.

Attributi *ereditati*

Un attributo è ereditato se il valore dello stesso, per un certo nodo di un albero di parsing, è definito nei termini degli attributi del nodo padre e/o dei nodi fratelli di quel nodo. Questo tipo di attributo è conveniente per esprimere determinate dipendenze di un costrutto di un linguaggio di programmazione relativamente al contesto di dove tale costrutto appare.

Grafo delle dipendenze

Le dipendenze tra gli attributi ereditati e gli attributi sintetizzati relativi ai nodi di un albero di parsing possono essere descritti attraverso un grafo diretto, detto *grafo delle dipendenze*. Prima di costruire un grafo delle dipendenze per un albero di parsing, è necessario porre ogni regola semantica nella forma $b := f(c_1, c_2, \ldots, c_k)$. Il grafo ha un nodo per ogni attributo ed un arco entrante nel nodo dell'attributo b ed uscente dal nodo dell'attributo c se b dipende dall'attributo c. L'alboritmo usato per creare il grafo delle dipendenze è il seguente:

```
for ogni nodo n nell'albero do
  for ogni attr a del simbolo nel nodo n do
    costruisci un nodo nel grafo delle dipendenze per a;
for ogni nodo n nell'albero do
  for ogni regola semantica b := f(c1, c2, ..., ck)
      associata alla produzione usata per n do
    for i:=1 to k do
```

```
costruisci nel grafo un arco dal nodo c_i
verso il nodo per b ;
```

Un esempio di albero delle dipendenze per una produzione grammaticale è già stato proposto nella figura 4.1.

Ordine di valutazione

Un *ordine topologico* di un grafo aciclico diretto è un ordinamento $m_1, m_2, \ldots, m_k$ di nodi del grafo tali che tutti gli archi $m_i \to m_j$ che collegano due nodi fanno si che m_i appaia prima di m_j nell'ordine. Ogni ordine topologico di un grafo delle dipendenze fornisce un corretto ordine di valutazione delle regole semantiche associate ai nodi nell'albero di parsing: da un ordine topologico di un grafo di dipendenze si desume l'ordine di valutazione delle regole semantiche.

Compreso questo, sono stati proposti vari metodi per la valutazione delle regole semantiche:

1. *metodo dell'albero di parsing.* Effettuato a tempo di compilazione, questo metodo ottiene un ordine di valutazione da un ordine topologico desunto dal grafo delle dipendenze costruito a partire dall'albero di parsing.

2. *metodo basato sulle regole.* Effettuato a tempo di compilazione, le regole semantiche associate alle produzioni sono analizzate passo dopo passo o tramite tool specializzati.

3. *metodo oblicuo.* Viene scelto un ordine di valutazione senza cosiderare le regole semantiche.

Gli ultimi due metodi non costruiscono esplicitamente il grafo delle dipendenze a tempo di compilazione, in modo da essere più efficienti come tempi e spazio di compilazione.

4.2 Albero di sintassi astratta

Un albero di sitassi astratta è una forma condensata dell'albero di parsing ed è utilizzato per rappresentare i construtti del linguaggio. Nell'albero di sintassi astratta, gli operatori e le lettere non appaiono nelle foglie ma sono associate ad un nodo interno che diventa padre delle stesse foglie specificate nell'albero di parsing. Se si considera la grammatica

$L \to E\,\mathbf{n}$
$E \to E + T$
$E \to T$
$T \to T * F$
$T \to F$
$F \to (E)$
$F \to \mathbf{digit}$

e la stringa, da essa derivabile, $3 + 3 * 2$ l'albero di sintassi astratta è quello espresso nella figura 4.2. La costruzione di un albero di sintassi astratta è così definita. Ogni nodo in un albero di sitassi astratta è implementato attraverso un

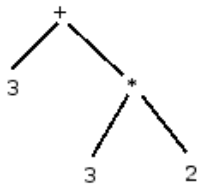


Figura 4.2: Albero di sintassi astratta per la stringa $digit + digit * digit$

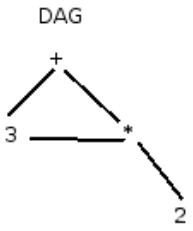
record con determinati campi. In un nodo per un'operatore, un campo identifica l'operatore stesso e gli altri campi contengono i puntatori agli operandi (nodi figli). Vengono comunemente usate le seguenti funzioni per creare i nodi di un albero di sintassi astratta da un espressione con operatori binari:

1. $mknode(op, left, right)$ crea un nodo etichettato *op* e fa puntare i due campi sucessivi agli operandi di sinistra e di destra.
2. $mkleaf(\mathbf{id}, entry)$ crea una foglia contenete l'etichetta *id* e inserisce in *entry* il puntatore all'entry nella tabella dei simboli per l'identificatore.
3. $mkleaf(\mathbf{num}, val)$ crea una foglia contenete l'etichetta *num* e inserisce in *val* il valore del numero.

Considerando che ognuna restituisce il puntatore alla struttura appena creata, il risultato dell'applicazione di queste funzioni alla grammatica sopra esposta definisce la grammatica con attributi seguente:

$E \rightarrow E_1 + T$ {
  $E.ptr := mknode('+', E_1.ptr, T.ptr)$
}
$E \rightarrow T$ {
  $E.ptr := T.ptr$
}
$T \rightarrow T_1 * F$ {
  $T.ptr := mknode('+', T_1.ptr, F.ptr)$
}
$T \rightarrow F$ {
  $T.ptr := F.ptr$
}
$F \rightarrow (E)$ {
  $F.ptr := E.ptr$
}
$F \rightarrow \mathbf{digit}$ {
  $F.ptr := mkleaf(\mathbf{digit}, \mathbf{digit}.lexval)$
}

Allo stesso modo dell'albero di sintassi astratta, si può definire un grafo aciclico (DAG) di sintassi astratta che non crea nodi se questi sono già stati inseriti precedentemente (un esempio si ha nella figura 4.3). I nodi sono implementati come record salvati in un array. Ogni record è associato ad un etichetta (l'indice



RAPPRESSENTAZIONE

1	digit	3	
2	digit	2	
3	*	1	2
4	+	1	3

Figura 4.3: DAG di sintassi astratta per la stringa $digit + digit * digit$

dell'array, detto *value number* per ragioni storiche) che determina la natura del nodo. Data una segnatura standard, definita dalla tripla $< op, l, r >$, che indica un nodo del DAG, algoritmo che permette l'inserimento di un nuovo nodo è molto semplice: presa in input una tripla come appena definita, si cerca nell'array un nodo m caratterizzato dalla stessa tripla. Si possono verificare due cose:

- **se** il nodo viene trovato, si ritorna il puntatore ad m.
- **altrimenti** si crea un nuovo nodo n con etichetta op, figlio sinistro l e figlio destro r e si ritorna il puntatore alla struttura appena creata.

Per aumentare l'efficienza dell'algoritmo di ricerca, di solito l'array viene implementato attraverso una tabella hash.

4.3 Grammatiche con S-attributi e L-attributi

Un attributo è *sintetizzato* se tutte le sue dipendenze puntano dai nodi figli al nodo padre dell'albero sintattico. Formalmente, un attributo è sintetizzato se, data la produzione grammaticale $A \rightarrow X_1 \dots X_n$, l'unica equazione con l'attributo a di A nella parte sinistra ha la forma

$$A.a := f(X_1.a_1, \dots X_1.a_k, \dots, X_n.a_1, \dots, X_n.a_k)$$

Un attributo è detto *left* se tutte le sue dipendenze sono ereditate dai soli fratelli che compaiono alla sua sinistra nell'albero sintattico e/o dal padre. Formalmente, un attributo di X_j, con $1 \leq j \leq n$, è left se, data la produzione grammaticale $A \rightarrow X_1 \dots X_n$, dipende soltanto:

1. dagli attributi dei simboli $X_1, \dots, X_{j-1}$ alla sinistra di X_j nella produzione.
2. dagli attributi ereditati da A.

Detto questo, si dice *grammatica con S-attributi* una grammatica che ha tutti attributi sintetizzati, e l'albero da essa generato viene valutato (bottom-up) in post-visita sinistra (nodo sinistro, nodo destro, radice). Una grammatica che ha attributi ereditati di tipo left è una *grammatica con L-attributi*, e l'albero da essa generato viene valutato (top-down) in pre-visita sinistra (radice, nodo sinistro, nodo destro).

Capitolo 5

Tabella dei simboli

La *tabella dei simboli* serve per associare ad ogni identificatore del linguaggio il suo tipo e le sue caratteristiche. E' possibile costruire la tabella dei simboli a partire dall'analisi lessicale e completarla nelle fasi sucessive. Nei linguaggi di programmazione moderni, la costruzione della tabella dei simboli viene rimandata in fase di analisi semantica.

Questa tabella definisce una specie di dizionario, a cui sono associate tre operazioni di base:

1. l'operazione di *lookup* (accesso), che deve essere veloce, cercando di evitare il più possibile ricerche lineari.
2. l'operazione di *insert* (inserimento), che deve richiedere tempi ragionevoli poichè avvengono una volta sola per ogni identificatore.
3. l'operazione di *delete* (cancellazione), che non è essenziale poichè lo spazio occupato dalla tabella e dagli elementi che la compongono può tranquillamente essere reclamato quando il compilatore termina.

La struttura che di solito si utilizza per l'implementazione è una tabella hash con buckets per risolvere le collisioni ma, se si vuole maggiore efficienza i bucket possono essere sostituiti con degli alberi binari di ricerca. La strutturazione degli elementi che compongono la tabella dei simboli dipende fortemente dal tipo di linguaggio sorgente.

Nei linguaggi di programmazione moderni, ovvero quelli che ammettono dichiarazioni annidate (tutti eccetto il Fortran) bisogna gestire il fatto che una dichiarazione ha una portata, un "tempo di vita", durante l'analisi del codice. Una possibile implementazione per risolvere questo problema utilizza una tabella dei simboli per ogni ambiente annidato. Il problema è che questa soluzione risulta costosa. Per gestire le portate di dichiarazioni, senza che questo comporti un costo esagerato, si utilizza una pila di ambienti, in cui ogni ambiente è la lista degli identificatori dichiarati in quell'ambiente. Di conseguenza, se si vuol ricercare un certo simbolo x nella tabella dei simboli, si prende il primo simbolo x che si incontra nella pila degli ambienti scorrendola dall'alto verso il basso.

Un ulteriore modo per implementare la tabella dei simboli è quello di abbinare una tabella hash ad un array di strutture: nella tabella hash sono contenuti gli indici di un array che implementa una pila di strutture, le cui entrate sono create man a mano che si trova una nuova dichiarazione; l'annidamento dei

blocchi viene lasciato di responsabilità di una seconda pila, in testa alla quale si trova l'indice della struttura più recente.

Capitolo 6

Type checking

6.1 Nozioni

Il *type checking* è quella sezione della compilazione che controlla che il tipo di un'espressione sia valido nel contesto in cui essa compare. Il *tipo* descrive una collezione di valori e vengono specificati da opportune *espressioni di tipo*. Queste espressioni di tipo possono essere un *tipo di base* (int, float, char, void, type_error) oppure *tipi complessi* (array, strutture, puntatori) formati applicando un *costruttore di tipo* ai tipi di base.

Le espressioni di tipo possono contenere variabili i cui valori sono espressioni di tipo. Una *dichiarazione di tipo* è un modo per definire nuovi tipi a partire da quelli già presenti. Un *sistema di tipi* è una collezione di regole per assegnare espressioni di tipo alle varie parti di un programma e vengono solitamente definiti per induzione sulla struttura. Il *type checker* è l'implementazione di un sistema di tipi. Per riuscire ad effettuare un checking corretto in un compilatore, si utilizza un ulteriore campo per gli elementi della grammatica con attributi detto *type* in cui viene salvato il tipo dell'elemento considerato.

Nelle sezioni seguenti vedremo come viene definito il type checking, a seconda dei costrutti incontrati, della grammatica qui di seguito.

$P \rightarrow D; S$
$D \rightarrow D; D | \mathbf{id} : T$
$T \rightarrow \mathbf{char} | \mathbf{integer} | \mathbf{array}[\mathbf{num}]\mathbf{of}\, T | * T | T' \rightarrow' T$
$E \rightarrow \mathbf{literal} | \mathbf{num} | \mathbf{id} | E\, \mathbf{mod}\, E | E[E] | E * | E(E)$
$S \rightarrow \mathbf{id} := E | \mathbf{if}\, E\, \mathbf{then}\, S | \mathbf{while}\, E\, \mathbf{do}\, S | S \rightarrow S; S$

6.1.1 Type checking delle Dichiarazioni e dei Tipi

$P \rightarrow D; S$
$D \rightarrow D; D$
$D \rightarrow \mathbf{id} : T$ {
    $addtype(\mathbf{id}.entry, T.type)$
}
$T \rightarrow \mathbf{char}$ {
    $T.type = char$
}

$T \rightarrow$ **integer** {
    $T.type = integer$
}
$T \rightarrow$ **array**[**num**]**of** T_1 {
    $T.type = array(1..\mathbf{num}.val, T_1.type)$
}
$T \rightarrow *T_1$ {
    $T.type = pointer(T_1.type)$
}
$T \rightarrow T_1' \rightarrow' T_2$ {
    $T.type := T_1.type \rightarrow T_2.type$
}

6.1.2 Type checking di Espressioni

$E \rightarrow$ **literal** {
    $E.type := char$
}
$E \rightarrow$ **num** {
    $E.type := integer$
}
$E \rightarrow$ **id** {
    $E.type := lookup(\mathbf{id}.entry)$
}
$E \rightarrow E_1$ **mod** E_2 {
    $E.type :=$ **if** $E_1.type = integer$ **and** $E_2.type = integer$
                then $integer$
                else $type_{error}$
}
$E \rightarrow E_1[E_2]$ {
    $E.type :=$ **if** $E_2.type = integer$ **and** $E_1.type = array(s,t)$
                then t
                else $type_{error}$
}
$E \rightarrow E_1*$ {
    $E.type :=$ **if** $E_1.type = pointer(t)$
                then t
                else $type_{error}$
}

6.1.3 Type checking di Statement

$S \rightarrow$ **id** $:= E$ {
    $S.type :=$ **if** **id**$.type = E.type$
                then $void$
                else $type_{error}$
}
$S \rightarrow$ **if** E **then** S_1 {
    $S.type :=$ **if** $E.type = boolean$
                then $S_1.type$

$$\textbf{else}\ type_{error}$$

}

$S \rightarrow \textbf{while}\ E\ \textbf{do}\ S_1$ {

$S.type := \textbf{if}\ E.type = boolean$

$$\textbf{then}\ S_1.type$$

$$\textbf{else}\ type_{error}$$

}

$S \rightarrow S_1; S_2$ {

$S.type := \textbf{if}\ S_1.type = void\ \textbf{and}\ S_2.type = void$

$$\textbf{then}\ void$$

$$\textbf{else}\ type_{error}$$

}

In ogni linguaggio di programmazione si possono costruire statement che, per come sono formati, non hanno un tipo definito. Ed esempio, qual è il tipo di un comando di assegnamento? Per risolvere questa mancata tipizzazione, a questi costrutti, se sono ben tipati, viene assegnato il tipo di base *void*.

6.1.4 Type Checking di Funzioni

$E \rightarrow E_1(E_2)$ {

$E.type := \textbf{if}\ E_2.type = s\ \textbf{and}\ E_1.type = s \rightarrow t$

$$\textbf{then}\ t$$

$$\textbf{else}\ type_{error}$$

}

In questo caso si lascia come sottintesa la produzione, con la relativa regola semantica, che permette di dichiarare funzioni. Si suppone di averla già in possesso.

6.2 Equivalenza

Nelle regole semantiche del type checker esposto nella sezione precedente comparivano dei comandi della forma "se due espressioni di tipo sono uguali allora fai qualcosa altrimenti fai qualcos'altro". La cosa non è poi così semplice: bisogna essere in grado di capire quando due espressioni di tipo sono equivalenti. Esistono due principali tipi di equivalenza:

- *equivalenza strutturale* (coincide con l'identità). Due espressioni sono strutturalmente uguali quando rappresentano lo stesso tipo o sono formate dall'applicazione dello stesso costruttore su tipi strutturalmente equivalenti.
- *equivalenza nominale*. Due espressioni sono equivalenti solo se sono sintatticamente uguali: l'equivalenza nominale vede ogni nome di tipo come un tipo differente.

Se si considerano due tipi *ricorsivi*, ovvero un tipo struttura S al cui interno uno dei suoi campi contiene un riferimento ad allo stesso tipo struttura S, questi sono considerati equivalenti se gli alberi sintattici a loro corrispondenti sono uguali.

6.3 Conversioni di tipo

La conversione da un tipo ad un altro è detta *implicita* se è portata a termine automaticamente dal compilatore. Questa conversione implicita, detta *coercizione*, è limitata nei linguaggi alle situazioni in cui non ci sia perdita di informazione. E' il type checker che solitamente si occupa di effettuare la coercizione. Per esempio, se si ha una somma tra due operandi A e B, il primo intero ed il secondo reale, prima di applicare questa somma, il type checker trasforma A in un reale in modo che l'operazione venga eseguita su operandi dello stesso tipo. Nel caso specifico appena considerato, l'unica coercizione possibile è quella che porta da interi a reali: il viceversa non è eseguibile perchè comporterebbe una perdita di informazioni e, di conseguenza, si violerebbe l'invariante prima esposto.

Una conversione viene detta *esplicita* quando viene eseguita direttamente dal programmatore. Un esempio di questa conversione può essere nell'operazione di *malloc* del C, in cui il programmatore, per avere un puntatore di un determinato, tipo fa precedere all'invocazione della funzione di allocazione l'operazione di casting.

6.4 Overloading di funzioni e operatori

Un simbolo viene detto *overloaded* se può assumere differenti significati a seconda del contesto in cui si trova. Nei linguaggi di programmazione, per esempio, tutte le operazioni aritmetiche sono overloaded poichè l'operazione somma applicata a due operandi è diversa a seconda del tipo degli operandi: se sono interi si usa la somma tra interi, se sono reali si usa la somma tra reali.

Un *overloading* viene risolto quando il significato dell'operatore/funzione è individuato in maniera unica. Nel caso delle operazioni aritmetiche, l'overloading viene fatto attraverso un'analisi per casi eseguita sugli operandi. L'overloading, però, non è assoluto: non è sempre possibile risolverlo guardando i soli argomenti della funzione. Nella sezione 6.1.4 è stato assunto che ogni espressione possa avere un unico tipo in modo che la regola semantica di type checking potesse essere

$$
\begin{array}{l}
E \rightarrow E_1(E_2)\ \{ \\
\quad E.type := \textbf{if}\ E_2.type = s\ \textbf{and}\ E_1.type = s \rightarrow t \\
\qquad\qquad\quad \textbf{then}\ t \\
\qquad\qquad\quad \textbf{else}\ type_{error} \\
\}
\end{array}
$$

Ciò non è vero! Bisogna considerare la possibilità che l'espressione sia associata non ad un solo tipo ma ad un insieme di tipi. In questo senso, si dice che la funzione è *polimorfa* nell'output. La semantica di type checking deve essere modificata conseguentemente come di seguito specificato:

$$
\begin{array}{l}
E \rightarrow E_1(E_2)\ \{ \\
\quad E.type := \{t | \exists s \in E_2.type\ t.c.\ s \rightarrow t \in E_1.type\} \\
\}
\end{array}
$$

6.5 Funzioni polimorfe

Una funzione viene detta *polimorfa* quando può essere applicata ad argomenti di tipo differente. Le funzioni polimorfe sono molto utili in pratica perchè consentono di definire algoritmi che operano su *strutture dati*, facendo riferimento a come queste strutture dati sono composte piuttosto che agli elementi che le compongono (polimorfismo parametrico).

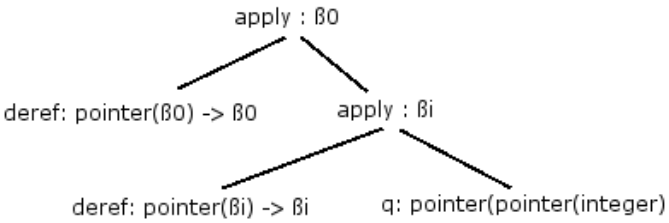


Figura 6.1: Albero sintattico etichettato per *defer(deref(q))*

Per parlare del polimorfismo nei linguaggi di programmazione bisogna introdurre qualche notazione: le lettere dell'alfabeto greco che si incontrereanno nei paragrafi successivi indicano il tipo *variabile*, che identifica un tipo sconosciuto. Lo statement preciso riferito ad un insieme sul quale la funzione polimorfa può essere applicata viene reso dal simbolo $\forall$ col significato di "per ogni tipo". Di conseguenza

$$\forall\beta.pointer(\beta) \rightarrow \beta$$

identifica la funzione polimorfa *deref* qui esposta:

```
function deref(p):
begin
   return p*
end;
```

Le regole di type cheking per le funzioni polimorfe differiscono di tre punti fondamentali con quelle delle normali funzioni:

1. le distinte occorrenze di una funzione polimorfa in una stessa espressione possono non essere dello stesso tipo.

2. la nozione di equivalenza tra tipi è completamente diversa a causa della presenza di tipi variabili. Supponiamo che E_1 di tipo $s \rightarrow s'$ sia applicata ad E_2 di tipo t. Invece di determinare l'equivalenza come spiegato in precedenza, E_1 ed E_2 devono essere unificate, ovvero bisogna determinare se s e t possono essere strutturalmente equivalenti attraverso un rimpiazzamento delle variabili di tipo in s e in t con un tipo espressione. Per esempio, nel nodo interno etichettato **apply** della figura 6.1, l'uguaglianza $pointer(\beta_i) = pointer(pointer(integer))$ è vera se β_i è sostituito con $pointer(integer)$.

3. si deve avere a disposizione un meccanismo per unificare due espressioni.

Sostituzioni, istanze, unificazione

L'informazione riguardante i tipi rappresentati da variabili è formalizzata definendo un "mapping" dai i tipi variabile ai tipi espressione. Questa operazione di mapping viene chiamata *sostituzione*. Per convenienza verrà in seguito scritto $S(t)$ per il tipo espressione che viene restituito dall'applicazione della funzione sostituzione con il tipo variabile *t*. Il risultato $S(t)$ viene detto *istanza di t*. Se la sostituzione S non specifica alcun tipo espressione per la variabile β a cui è applicata, si assume che $S(\beta)$ sia β. E' importante ricordare che ogni istanza di una funzione polimorfa è monomorfa.

Due tipi espressione t_1 e t_2 *unificano* se esite una sostituzione $S(t_1) = S(t_2)$. In pratica, si cerca l'*unficatore più generale* (in inglese, *most general unifier*, *mgu*). L'mgu di due espressioni t_1 e t_2 è una sostituzione S con le seguuenti prorpietà:

1. $S(t_1) = S(t_2)$
2. per ogni altra sostituzione S' tale che $S'(t_1) = S'(t_2)$, la sostituzione S' è un'istanza di S.

6.5.1 Type checking di funzioni polimorfe

Per riuscire a fare type checking delle funzioni polimorfe si utilizzano due principali funzioni:

$fresh(t)$ rimpiazza le variabili legate dal simbolo $\forall$ nell'espressione t con variabili nuove e ritorna un puntatore ad un nodo che rappresenta il nuovo tipo. Ogni simbolo $\forall$ in t viene eliminato.

$unify(m, n)$ unifica le espressioni di tipo che sono rappresentate dai nodi indirizzati da m e n. Inoltre:

- come effetto collaterale, tiene traccia di tutte le sostituzioni che sono state fatte.
- se le espressioni non sono unificabili, l'intero processo fallisce.

Data la seguente grammatica

$P \rightarrow D; E$
$D \rightarrow D; D | \mathbf{id} : Q$
$Q \rightarrow \forall \mathbf{type\_variable}.Q | T$
$T \rightarrow \mathbf{constructor}(T) | T \times T | (T) | T' \rightarrow'$
$T | \mathbf{basic\_type} | \mathbf{type\_variable}$
$E \rightarrow \mathbf{id} | E, E | E(E)$

le azioni semantiche per le espressioni sono

$E \rightarrow E_1(E_2)$ {
  $p := mkleaf(newtypevar)$;
  $unify(E_1.type, mknode('\rightarrow', E_2.type, p))$;
  $E.type := p$
}

$E \rightarrow E_1, E_2$ {
    $E.type := mknode('\times', E_1.type, E_2.type)$
}
$E \rightarrow$ **id** {
    $E.type := fresh(\mathbf{id}.type)$
}

Ci sono due modi per controllare la correttezza dei tipi: il *sistema di equazioni* e l'*algoritmo di unificazione*. Per fare un esempio del funzionamento di tutti e due riprendiamo l'esempio fornito nella figura 6.1.

Sistema di equazioni

Per ogni etichetta **apply** che si incontra sull'albero, partendo dal basso, si scrive un equazione. Per rendere logicamente possibile questa uguaglianza, è necessario trasformare il figlio di destra di ogni **apply** in una funzione (che prende in input il tipo del nodo del figlio destro e restituisce una nuova variabile) in modo da permettere il confronto con il figlio sinistro. Fatto questo, seguendo l'esempio di figura 6.1, risulta:

$$pointer(\beta_i)' \rightarrow' \beta_i = pointer(pointer(integer))' \rightarrow' \gamma_1$$
$$pointer(\beta_0)' \rightarrow' \beta_0 = \gamma_1' \rightarrow' \gamma_2$$

Sostituendo opportunamente nella prima equazione e applicando la sostituzione della variabile γ_1 nella seconda si ottiene

$$\beta_i = \gamma_1 = pointer(integer)$$
$$pointer(\beta_0)' \rightarrow' \beta_0 = pointer(integer)' \rightarrow' \gamma_2$$

e da qui, infine

$$\beta_i = \gamma_1 = pointer(integer)$$
$$\beta_0 = \gamma_2 = pointer(integer)$$

In questo senso, le equazioni individuano una *sostituzione* di variabili con termini.

Algoritmo di unificazione

Quest'altro metodo, molto più efficiente, controlla la correttezza dei tipi applicando l'operazione *unify*, descritta nell'algoritmo 10, sui due figli dei nodi **apply**. Anche in questo caso è necessario trasformare il figlio di destra di ogni **apply** in una funzione (che prende in input il tipo del nodo del figlio destro e restituisce una nuova variabile) in modo da permettere il confronto con il figlio sinistro. Alla fine dell'algoritmo, le due radici puntano entrambe alla lista unione.

Per comprendere questo algoritmo bisogna spiegare il significato delle due funzioni ausiliarie che vi compaiono all'interno.

$find(n)$ Ritorna il nodo rappresentativo della classe di equivalenza che contiene il nodo n.

Algorithm 10 Algoritmo di unificazione

```
boolean unify(node m, node n) {
  s=find(n);
  t=find(m);
  if(s=t)
    return true;
  elseif(s e t sono nodi che rappresentano lo stesso
         tipo di base)
    return true;
  elseif (s e' un nodo operatore con s1 e s2 figli and
          t e' un nodo operatore con t1 e t2 figli)
  {
    union(s,t);
    return unify(s1,t1) and unify(s2,t2);
  }
  elseif(s e' una variabile che non occorre in t o
         t e' una variabile che non occorre in s)
  {
    union(s,t);
    return true;
  }
  else
    return false;
```

$union(m, n)$ Fa il merge delle classi di equivalenza che contengono i due nodi m e n. Una volta uniti, la funzione sceglie un nuovo rappresentante per l'insieme unione seguendo le seguenti regole:

- se esite un qualche nodo non variabile, viene scelto uno di questi a fare da rappresentante.
- se sono presenti solo nodi variabili, viene scelto uno tra i due originali rappresentanti dei due insiemi uniti.

Oltre a ciò, è necessario indicare che all'inizio dell'algoritmo, ogni nodo è rappresentante di se stesso. L'algoritmo appena spiegato è descritto graficamente nella figura 6.2 (nella tabella il tipo in figura il tipo seguito da * è il rappresentante).

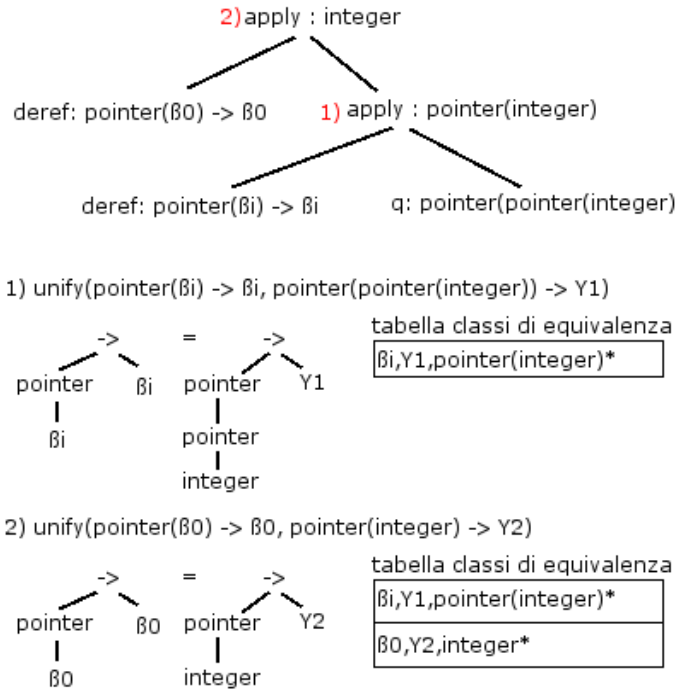


Figura 6.2: Algoritmo di unificazione per *deref(deref(integer))*

Capitolo 7

Ambienti di run-time

7.1 A cosa servono

Prima di affrontare la parte relativa alla generazione del codice intermedio, è necessario studiare le relazioni che intercorrono tra le istruzioni del programma e le azioni che saranno fatte al tempo di esecuzione, in modo da implementarle. A tempo di esecuzione, uno stesso nome nel codice sorgente può denotare dati differenti nel nella macchina di target. In questo capitolo verranno esaminate le relazioni tra i nomi e i dati a loro relativi.

L'allocazione e la deallocazione dei dati è gestita del package del supporto di run-time. L'organizzazione della memoria e la strutturazione dei registri della macchina, su cui verrà eseguito il programma, è quello che comunemente viene chiamato *ambiente di run-time*.

Per fare un esempio pratico, consideriamo il codice in figura 7.1, in cui sono ammesse dichiarazioni di funzioni annidate. Una *definizione di procedura* è una dichiarazione che, nella forma più semplice, associa un identificatore ad uno statement. L'identificatore è il *nome della procedura*, mentre lo statement è il *corpo della procedura*. Le procedure che ritornano un valore vengono chiamate *funzioni*.

Quando un nome di una procedura appare senza uno statement esecutivo specificato significa che la procedura viene *invocata* in quell'esatto punto del codice. Tutti gli identificatori che appaiono nella definizione della procedura sono chiamati *parametri formali* della procedura. Invece gli argomenti passati alla procedura al momento dell'invocazione vengono chiamati *parametri attuali* della procedura e vengono sostituiti al posto dei formali nel corpo della procedura invocata.

Albero di attivazione

Si possono fare le seguenti due assunzioni a proposito del flusso di controllo riferito alle procedure durante l'esecuzione del programma:

1. il flusso di controllo è sequenziale.
2. ogni esecuzione di procedura comincia all'inizio del corpo della procedura ed eventualmente ritorna il controllo nel punto immediatamente successivo al posto in cui era stata chiamata.

```
program sort(input,output);
  var a:array[0..10] of integer;
      x:integer;
  procedure readarray;
    var i integer;
    begin ... a ... end; {readarray}
  procedure swap(i,j:integer);
    begin x:=a[i]; a[i]:=a[j]; a[j]:=x end; {swap}
  procedure quicksort(m,n:integer);
    var k,v:integer;
    function partition(y,z:integer):integer;
      var i,j:integer;
      begin ... a ...
            ... v ...
            ... swap(i,j) ...
      end; {partition}
  begin ... end; {quicksort}
begin ... end. {sort}
```

Figura 7.1: Esempio di codice in Pascal

Ogni esecuzione del corpo di una procedura fa riferimento ad una "attivazione" della procedura. La *vita* di un'attivazione di una procedura p è data dalla sequanza di passi tra il primo e l'ultimo passo effettuato nell'esecuzione del corpo della procedura, incluso il tempo speso per eseguire le procedure invocate da p stessa. In generale, il tempo di vita si riferisce a consecutive sequenze di passi che compongono l'esecuzione del programma.

Una procedura viene detta *ricorsiva* se, durante la sua esecuzione, può re-invocare se stessa. Per definire visivamente tutte queste chiamate di procedura annidate, viene usato un albero, detto *albero di attivazione*, per tenere traccia dei vari record che vengono inseriti/eliminati durante l'esecuzione di un programma. In un albero di attivazione:

1. ogni nodo rappresenta un'attivazione di una procedura.
2. la radice rappresenta l'attivazione per il *main* del programma principale.
3. il nodo per la procedura a è genitore del nodo per la procedura b se e solo se il controllo passa dall'attivazione di a a quella di b.
4. il nodo per la procedura a è a sinistra del nodo per la procedura b se e solo se il tempo di vita di a occorre prima del tempo di vita per b.

Pila di controllo e ambiente

Il flusso di controllo in un programma corrisponde ad una visita posticipata sinistra partendo dalla radice dell'albero di attivazione. Per determinare quali procedure sono ancora "vive" e quali no, si usa una pila, detta *pila di controllo*, per tenere traccia delle attivazioni di procedura ancora in vita: ogni volta che una procedura p viene invocata si aggiunge la relativa attivazione sullo stack, e vi rimane finche p non termina la sua esecuzione.

Ogni linguaggio ha le proprie regole statiche che definiscono la portata delle dichiarazioni, siano esse di variabile o di funzione. Quelli che prima sono stati definiti informalmente come *dati* di una dichiarazione corrispondono alla *locazione di memoria* dove sono memorizzati i dati. Nei linguaggi di programmazione, il termine *ambiente* si riferisce ad una funzione che mappa un nome con un indirizzo di memoria, mentre il termine *stato* si riferisce ad una funzione che mappa un indirizzo di memoria con il valore in esso salvato. Ovvero, l'ambiente mappa un nome con un l-valore, lo *stato* mappa un l-valore con un r-valore.

7.2 Organizzazione della memoria

La suddivisione a tempo di esecuzione della memoria è fornita da:

- il codice di target generato
- i dati (locazioni di memoria)
- la pila che gestisce l'attivazione delle procedeure

La grandezza del codice di target è conosciuta a tempo di compilazione, così come la dimensione della maggior parte dei dati. Tutte questi oggetti, siccome sono conosciuti staticamente a tempo di compilazione, vengono salvati in un area *determinata staticamente*, indicata dalla figura 7.2. Quando interviene

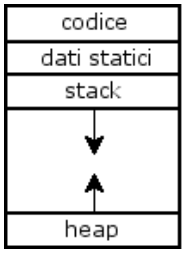


Figura 7.2: Tipica suddivisione della memoria a tempo di esecuzione.

un'invocazione, l'esecuzione di un'attivazione viene interrotta e le informazioni relative allo stato della macchina (valore del program counter e dei registri) vengono salvati sullo stack. Quando l'ultima procedura invocata restituisce il controllo, l'attivazione viene fatta ripartire dopo aver ripristinato i valori dei registri rilevanti e aver fatto puntare il program counter nel punto del codice immediatamente successivo alla chiamata. Un ultima area, chiamata *heap*, serve per memorizzare tutte le altre strutture dati. Le dimensioni dello stack e dell'heap sono soggette a cambiamenti durante l'esecuzione; per questo motivo essi occupano zone opposte della memoria e crescono il primo verso il basso, il secondo verso l'alto.

Record di attivazione

Le informazioni relative ad una singola esecuzione di una procedura sono gestite usando dei blocchi di memoria contigui chiamati *record di attivazione* che

consistono in una collezione di campi, come si può vedere dalla figura 7.3. I vari

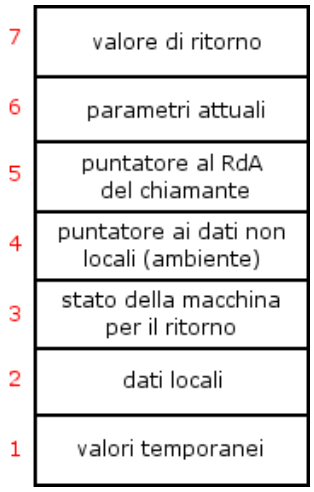


Figura 7.3: Record di attivazione

campi, a partire da quello dei valori temporanei, hanno il seguente significato:

1. sono i *valori temporanei* dove vengono memorizzati dei risultati parziali della valutazione di espressioni.

2. sono i *dati locali* alla procedura la cui dimensione è determinabile staticamente a partire dal tipo.

3. lo *stato della macchina per il ritorno* preserva i valori del program counter e dei registri **prima** dell'invocazione.

4. facendo riferimento ai *dati non locali* alla procedura, implementa l'ambiente statico nei linguaggi a scoping statico.

5. punta al *record di attivazione del chiamante*, in modo da far puntare il campo *top* della pila a questo puntatore quando la procedura termina.

6. il campo dei *parametri attuali* è riempito dal chiamante della procedura per segnalare i parametri passati al chiamato.

7. il campo dove il chiamato inserisce il *valore di ritorno* della procedura.

Il campo relativo ai dati locali e lo spazio di memoria necessario per memorizzarli può essere calcolato a tempo di compilazione. Di conseguenza, si tiene conto della quantità di memoria necessaria per memorizzare le dichiarazioni precedenti e, da questa quantità, si calcola un indirizzo relativo (detto *offset*) all'inizio del record di attivazione. Ovviamente, la memorizzazione dei dati locali dipende strettamente dall'indirizzamento della macchina utilizzata.

7.3 Tecniche di allocazione della memoria

Viene di solito usata una tecnica diversa si allocazione della memoria per ognuna delle tre aree, descritte in figura 7.2, a seconda di quella a cui si fa riferimento:

1. l'*allocazione statica* per tutti gli oggetti la cui dimensione è determinabile a tempo di compilazione.
2. l'*allocazione dinamica sullo stack* che gestisce i casi di allocazione a tempo di esecuzione.
3. l'*allocazione dinamica sull'heap* che gestisce l'allocazione e la deallocazione della memoria associata a determinate strutture di cui non si conosce a priori la dimesione.

7.3.1 Allocazione statica

Le condizione che devono avere gli oggetti allocati in questo modo sono fondamentalmente tre:

1. la dimensione dei dati e l'indirizzo della posizione in memoria deve essere conosciuta a tempo di compilazione.
2. le procedure ricorsive sono "ristrette" poichè tutte le attivazioni di una procedura usano gli stessi legami per i nomi locali.
3. le strutture dati non possono essere create dinamicamente se non è presente un meccanismo di allocazione a tempo di esecuzione.

7.3.2 Allocazione dinamica sullo stack

L'allocazione sullo stack è basata sull'idea di una *pila di controllo*: la memoria è organizzata a pila e i record di attivazione sono inseriti (*push*) ed estratti (*pop*) rispettivamente quando un'attivazione inizia e quano finisce. Inoltre, il valore delle variabili locali viene cancellato quando l'attivazione finisce. Si supponga che il registro *top* identifichi la testa dello stack. A tempo di esecuzione, un record di attivazione può essere allocato e deallocato incrementando e decrementando *top* a seconda della dimensione del record di attivazione. Se una procedura q ha un record di attivazione di dimensione a, allora *top* è incrementato di a prima che il codice di target di q sia eseguito. Una volta che q restituisce il controllo, *top* viene decrementato di a.

L'invocazione di procedure è implementata attraverso la generazione di quella che viene chiamata *sequenza di invocazione*. Questa sequanza di allocazione alloca un record di attivazione e inserisce le giuste informazioni nei campi. Una *sequenza di ritorno* ripristina lo stato della macchina in modo che la procedura chiamante possa continuare la sua esecuzione.

La *sequenza di invocazione* può essere descritta dai seguenti punti, svolti in modo strettamente sequenziale: *sequenza di invocazione*

1. il chiam*ante* valuta i paramentri attuali.
2. il chiam*ante* salva un indirizzo di ritorno è il vecchio valore del puntatore alla cima dello stack (*top_sp*) nel record di attivazione del chiam*ato*.

3. il chiam*ato* salva i valori contenuti nei registri e altre informazioni di stato.

4. il chim*ato* inizializza i dati locali ed inizia la sua esecuzione.

sequenza di ritorno

La *sequenza di ritorno* può essere descritta dai seguenti punti, svolti in modo strettamente sequanziale:

1. il chiam*ato* inserisce il valore di ritorno all'indirizzo successivo al record di attivazione del chiam*ante*.

2. usando le informazioni memorizzate nel campo *stato della macchina per il ritorno*, il chiam*ato* ripristina il puntatore alla cima dello stack (*top_sp*) e gli altri registri e ritorna all'indirizzo nel codice del chiam*ante*.

3. ua volta che il puntatore alla cima dello stack è stato decrementato, il chiam*ante* può copiare il valore ritornato nel suo record di attivazione ed usarlo per valutare un'espressione.

Variabili di lunghezza variabile

Una semplice strategia per gestire i dati di lunghezza variabile, come per esempio l'array, è quella di creare, nel record di attivazione, un campo contenente il puntatore all'array. L'array vero e proprio viene memorizzato in indirizzi di memoria esterni al record di attivazione per quella procedura. Il puntatore (il relativo indirizzo di memoria a cui punta) inserito nel record di attivazione della procedura si può conoscere già a tempo di compilazione in modo che il codice di target possa accedere agli elementi dell'array attraverso un puntatore.

Accesso ai dati non locali

Le regole di *scope* (portata) di un linguaggio determinano il trattamento dei dati globali di una procedura. Una regola comune, detta *scope statico*, determina la dichiarazione che viene applicata ad un nome esaminando solamente il testo del programma. Una regola alternativa, detta *scope dinamico*, determina la dichiarazione applicabile ad un nome a tempo di esecuzione, considerando i record di attivazione correnti.

il blocco

Quasi tutti i linguaggi moderni posseggono il concetto di *blocco* con la seguente sintassi:

$$\{dichiarazioni; statement\}$$

Lo *scope* di una dichiarazione in un linguaggio strutturato a blocchi (come può essere il C) è definito dalla seguente *regola di annidamento*:

1. lo scope di una dichiarazione in un blocco B include B.

2. se un nome x non è dichiarato nel blocco B allora un occorrenza di x in B è nello scope di una dichiarazione di x in un blocco contenitore B' tale che

   (a) B' ha una dichiarazione di x

   (b) di tutti quelli che contengono B ed hanno una dichiarazione di x, B' è un contenitore più prossimo a B.

Poichè lo scope di una dichiarazione non va oltre il blocco in cui si trova, lo spazio per la variabile può essere allocato quando si entra in un blocco e deallocato quando si esce. In questo senso, il blocco è una *procedura senza paramentri* che può essere invocata soltanto all'inizio del blocco e ritorna alla fine del blocco. Di conseguenza, l'accesso alle variabili globali di un blocco segue le stesse regole di quello sulle procedure, anche se nel caso dei blocchi la situazione risulta più semplice perchè il flusso di controllo verso/dal blocco segue il testo del programma. Si può alternativamente allocare preventivamente lo spazio necessario per i blocchi, quando si alloca il record di attivazione della procedura che li contiene, poichè la loro strutturazione è nota staticamente.

Ambiente con scoping statico senza annidamento di procedure (C, Java)

In C una definizione di procedura non può mia essere annidata in un'altra. Se in una procedura c'è un riferimento ad una variabile x non dichiarata allora x deve essere dichiarata nello spazio relativo alle variabili globali del programma. In questo caso si può tranquillamente usare l'allocazione dinamica sullo stack dei record di attivazione delle procedure perchè gli accessi alle variabili globali sono accessi assoluti alla porzione di dati statici della memoria a tempo di esecuzione.

Ambiente con scoping statico con annidamento di procedure (Pascal, JavaScript)

Nell'esempio descritto in figura 7.1, l'annidamento delle procedure è il seguente:

```
sort
   readarray
   swap
   quicksort
      partition
```

Applicando le regole di scope si vede che l'occorrenza della variabile a e l'invocazione della funzione *swap* nella funzione *partition* fanno riferimento alle dichiarazioni contenute nella funzione *sort*.

profondità di annidamento

La nozione di *profondità di annidamento* di una procedura è usata di conseguenza all'implementazione dello scope statico. La profondità di annidamento viene definita nel modo seguente: il programma principale è posto a profondità di annidamento 1; la profondità viene incrementata di un'unità se all'interno della procedura che stiamo considerando (in questo caso il *main*) compare una dichiarazione di una nuova procedura. Per esempio, sempre nella figura 7.1, la procedura *sort* è a profondità 1, *quicksort* (ma anche *readarray* e *swap*) è a profondità 2, mentre *partition* è a profondità 3.

access link

Un'implementazione dello scoping statico per le procedure annidate è ottenuto aggiungendo un puntatore, detto *access link*, ad ogni record di attivazione. Se una procedura p è annidata all'interno di una procedura q, l'access link nel record di attivazione di p punta all'access link nella più recente attivazione di q.

catena statica

Si supponga che la procedura p, annidata a profondità n_p, faccia riferimento ad una variabile non locale a dichiarata in una procedura al livello di annidamento $n_a \leq n_p$. La variabile a può essere recuperata a partire dalla procedura p come segue:

1. quando il controllo è di p, il record di attivazione di p è in cima allo stack. Vengono percorsi a ritroso $n_p - n_a$ access link dal record in cima allo stack. Il valore di $n_p - n_a$ è conosciuto a tempo di compilazione.

2. dopo aver percorso $n_p - n_a$ access link (che determinano la *catena statica*) si raggiunge il record di attivazione nel quale è contenuta la dichiarazione di a.

Gli access link nei record di attivazione vengono inizializzati all'atto di un'invocazione di procedura. Si supponga che una procedura p, annidata a profondità n_p, invochi la procedura q a profondita n_q. Il codice di inizializzazione dell'access link in questo caso dipende se la procedura invocata q è annidata o meno in p:

1. se $n_p < n_q$, ovvero la procedura q si trova a profondità maggiore di p, q deve essere dichiarata sicuramente in p altrimenti non sarebbe accessibile.

2. se $n_p \geq n_q$, ovvero la procedura q contiene p oppure si trovano allo stesso livello, per individuare l'ambiente statico di q bisogna risalire gli access link fino a trovare l'attivazione più recente della procedura che contiene q. Per fare ciò, bisogna risalire la *catena statica* di $n_p - n_q$ record di attivazione. Notare che il valore $n_p - n_q$ può essere determinato staticamente.

display

Una tecnica differente ma molto più efficiente a quella della risalita della catena statica è data da quella dei *display*. Il display, comunemente implementato come fosse un array, è tale che la variabile globale a profondita i è accessibile attraverso il puntatore in $d[i]$. Quando viene inizializzato un nuovo record di attivazione per una procedura sita a profondità di annidamento i:

1. il valore attuale di $d[i]$, che contiene un puntatore ad un record di attivazione, viene salvato in un campo "puntatore" nel nuovo record di attivazione.

2. $d[i]$ viene aggiornato facendolo puntare al nuovo record di attivazione.

Quando una procedura termina la sua esecuzione, prima di cancellare il suo record di attivazione dallo stack si preoccupa di ripristinare il valore di $d[i]$sul display al valore salvato.

Tutti questi passi sono giustificati come segue. Si supponga che una procedura p annidata a profondità j invochi una procedura q a profondita i. Allora si possono presentare due casi:

1. se $j < i$ significa che q è immediatamente annidata in p. In questo caso $i = j + 1$ e conseguentemente i primi j elementi del display sonogli stessi sia per p sia per q. Il puntatore $d[i]$ viene salvato nell'access link del nuovo record di attivazione di q e l'indirizzo di questo record viene memorizzato in $d[i]$.

2. se $j \geq i$ significa che le entrate nel display (da *1* a *i-1*) sono valide per q, quindi il puntatore $d[i]$ viene salvato nell'access link del nuovo record di attivazione di q e l'indirizzo di questo record viene memorizzato in $d[i]$. Tutti i puntatori contenuti in $d[i+1] \ldots d[j]$ restano memorizzati nel display poichè q non ha lcuna possibilità di accedervi.

Ambiente con scoping dinamico

Se si considera lo *scoping dinamico*, un record di attivazione eredita l'ambiente del record di attivazione del chiamante. Una variabile globale (non locale) a di una procedura p è la stessa variabile della procedura del chiamante q. Consideriamo il programma di seguito:

```
program dynamic(input, output);
  var x integer;
  procedure show;
    begin write(x) end;
  procedure small;
    begin var x integer; x:=2; show end;
  begin x:=1; show; small end.
```

Se si considera l'ambiente statico, questo programma stampa *1 1*. In ambiente dinamico, invece, la stampa risultante è *1 2*.

Le tecniche per permettere l'implementazione di questo ambiente sono simili a quelle viste per lo scoping statico:

1. il *deep access*, che utilizza la catena dinamica e la risale finchè non si trova la dichiarazione a cui si è interessati. Questa tecnica ha lo svantaggio di dover risalire la catena dinamica di un numero di record di attivazione non determinabile staticamente: basti pensare ad una funzione ricorsiva che ad ogni invocazione incrementa una variabile globale.

2. lo *shallow access*, che memorizza il valore corrente di ogni nome in un'area allocata staticamente. Il suo funzionamento è simile a quello del display: quando un record di attivazione A ridichiara un nome presente nell'ambiente, allora il vecchio nome viene salvato nel record di attivazione A e la struttura statica fa riferimento al nuovo nome appena dichiarato.

7.3.3 Allocazione dinamica sull'heap

L'allocazione dinamica sullo stack, descritta nella sezione 7.3.2, non viene usata se:

- il valore delle variabili locali deve essere conservato dopo la fine di un'attivazione.

- un'attivazione chiamata deve sopravvivere al chiamante. Questa possibilità non si incontra nei linguaggi dove l'albero di attivazione viene descritto dal flusso di controllo tra le procedure.

In entrambi i casi, la deallocazione dei record di attivazione non avviene in ordine LIFO e di conseguenza la memoria non può essere allocata a stack. L'heap è un area di memoria le cui porzioni possono essere occupate o libere, in qualunque ordine.

Le tecniche usate per l'allocazione dinamica della memoria sull'heap dipendono strettamente da come poi questa memoria viene deallocata.

Allocazione esplicita di blocchi di dimensione fissa

La più semplice forma di allocazione dinamica sull'heap coinvolge blocchi di dimensione fissata. La memoria libera viene gestita come fosse una lista concatenata: ogni blocco libero punta al successivo libero. La testa della lista (il primo blocco di memoria disponibile per un'allocazione) è segnalata dal puntatore *available*. In fase di allocazione, si prende il primo blocco, puntato da *available*, e si fa puntare *available* al successivo. In fase di deallocazione, il blocco liberato viene inserito in testa alla lista dei blocchi liberi.

Allocazione esplicita di blocchi di dimensione variabile

Quando dei blocchi vengono allocati e deallocati molte volte, la memoria inizia a diventare *frammentata*. La frammentazione non ha nessuna conseguenza

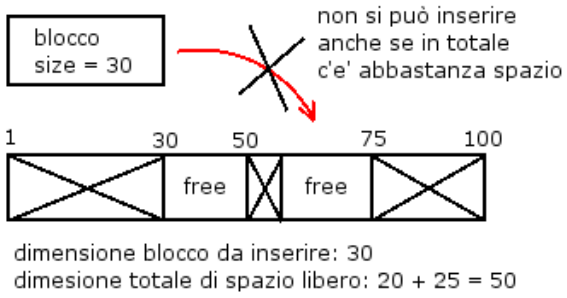


Figura 7.4: Frammentazione nell'allocazione esplicita con blocchi variabili

particolare per il funzionamento dell'allocazione con blocchi di dimensione fissata; invece, la situazione che si può presentare nel caso si utilizzino i blocchi di dimensione variabile, descritta sinteticamente in figura 7.4, diventerebbe un problema poichè non si potrebbe allocare un blocco B di dimensione superiore al più grande blocco libero presente anche se, considendo la totalità dei blocchi liberi, ci sarebbe spazio sufficiente per allocare B.

Un metodo usato per permettere l'allocazione di blocchi a dimensione variabile è chiamato *metodo first-fit*: se si vuol allocare un blocco, di dimensione s, si cerca, scorrendo la lista dei blocchi liberi, il primo blocco libero la cui dimensione t è tale che $t \geq s$; il blocco rimanente $t - s$ diventa libero. Quando un blocco viene deallocato, si controlla se è adiacente ad un blocco libero e, in tal caso, lo si unisce a questo per creare un blocco libero più grande, in modo da evitare inutili framentazioni.

Deallocazione implicita: garbage collection

La deallocazione implicita richiede la cooperazione tra il programma scritto dall'utente e il gestore dell'ambiente a tempo di esecuzione perchè quest'ultimo necessita di sapere quando un blocco di memoria non è più utilizzato. Considerando la strutturazione di un blocco di memoria presentata nella figura 7.5, il campo relativo all'*ampiezza del blocco* memorizza la dimensione in modo da poter determinare da dove inizia il blocco successivo. Il blocco viene definito *in uso*

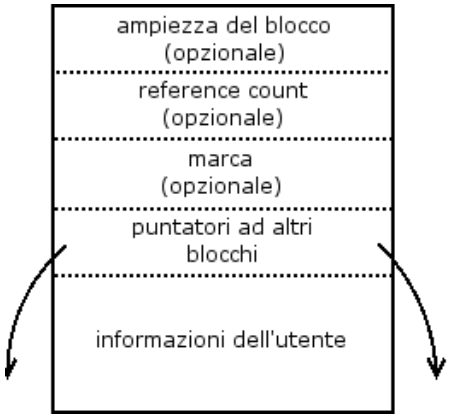


Figura 7.5: Struttura di un blocco di memoria

quando è possibile fare riferimento alle informazioni ivi contenute direttamente dal programma utente.

Detto questo si utilizzano due tecniche per permettere la deallocazione implicita.

- *Reference count.* Si tiene traccia, nel blocco b considerato, del numero di blocchi che puntano direttamente a b incrementando e decrementando la variabile *reference count* del blocco. Quando questa variabile va a 0 allora il blocco può essere deallocato, poichè non può più essere indirizzato dal programma utente. Questa soluzione non è troppo costosa e viene usata pesantemente se i puntatori tra blocchi non sono circolari. Se dovesse accadere ciò, il ciclo di blocchi, seppur il programma utente non lo puntasse più, non potrebbe essere deallocato poichè il *reference count* sarebbe ancora settato ad un valore maggiore di 0.

- *Tecnica di marking.* In questa tecnica viene sospesa momentaneamente l'esecuzione del programma utente e, attraverso dei puntatori, si determinano i blocchi che sono ancora in uso. Perchè questo approccio funzioni, bisogna conoscere **tutti** i puntatori presenti nello heap. L'algoritmo ha il seguente funzionamento:

  1. possedendo tutti i puntatori ai blocchi dello heap, si setta come *liberabile* il campo *marca* di tutti i blocchi.
  2. si scorrono tutti i blocchi che sono direttamente accessibili dall'ambiente, settando il campo *marca* di ognuno come *non liberabile*. Fatto questo si reitera la procedura di scorrimento partendo dai blocchi appena settati. La procedura termina quando non si settano nuovi blocchi.
  3. tutti i blocchi che hanno settato il campo *marca* a *liberabile* vengono deallocati dallo heap.

Capitolo 8

Generazione del codice intermedio

8.1 Codice a tre indirizzi

La generazione del *codice intermedio* è l'ultima parte del *front-end* per un compilatore. Questo è il codice che utilizza il back-end per generare il codice oggetto. Un tipo di codice intermedio, che qui studieremo, è quello definito da un susseguirsi di istruzioni sequenziali del tipo

$$x := y\ op\ z$$

dove x,y,z sono variabili, costanti o variabili temporanee generate dal compilatore, mentre *op* rappresenta un operazione.

Questo codice viene detto *codice a tre indirizzi*. Questo nome deriva dal fatto che un'istruzione contiene al più tre indirizzi: due per gli operandi ed uno per il risultato. Per esempio, la traduzione in codice a tre indirizzi dell'espressione $x + y * z$ sarebbe la sequenza:

```
t1 := y * z
t2 := x + t1
```

Qui di seguito sono esposti tutti i comandi utilizzabili nel codice a tre indirizzi:

```
x := y op z //comando di assegnamento a tre indirizzi
x := op y //comando di assegnamento a due indirizzi
x := y //comando di copia
goto L // salto incondizionato all'etichetta L
if x relop y goto L // salto condizionato a L
param x //definizione parametri per procedura
call p,n //invocazione della procedura p su n parametri
return y //ritorna il valore di ritorno della procedura
x := y[i] //assegnamento da array
x[i] := y // assegnamento di un array
x := &y // assegnamento dell'indirizzo di memoria di y
x := *y // assegnamento del contenuto puntato da y
*x := y // assegnamento di y nella memoria puntata da x
```

L'implementazione dei comandi a tre indirizzi è fornita con delle quadruple. Una *quadrupla* è una struttura con quattro campi: *op*, *arg1*, *arg2* e *result*. Vedendo una quadrupla matematicamente come $< op, arg_1, arg_2, result >$, un codice a tre indirizzi come *x:=y+z* genera la quadrupla $< +, y, z, x >$.

8.2 Tabella dei simboli e albero di sintassi astratta

I passi del processo di compilazione analizzati finora sono stati l'analisi lessicale, l'analisi sintattica e l'analisi semantica (con il type checking). Questi hanno portato alla produzione di un albero sintattico astratto ottimizzato (il dag astratto). La tabella dei simboli, creata durante la fase di analisi semantica e descritta al capitolo 5, è stata **logicamente cancellata** alla fine della fase stessa, ovvero la sua utilità è terminata con tale fase. In realtà le informazioni di questa tabella vengono utilizzate per creare un'ulteriore tabella dei simboli utilizzata in fase di generazione del codice intermedio, in corrispondenza delle valutazioni delle dichiarazioni. Per evitare inutili ambiguità, questa nuova tabella dei simboli verrà chiamata da ora in avanti *ICT* (*intermediate code table*).

Nel capitolo dedicato all'organizzazione della memoria (7), quando si faceva riferimento ad una tabella dei simboli, si intendeva usare la ICT e non quella discussa per l'analisi semantica. Il quel capitolo, però, non era ancora importante questa distinzione, o meglio non si poteva comprendere completamente.

Sebbene per la generazione del codice si intervenga direttamente sull'albero sintattico astratto restituito dalla fase di analisi semantica, albero i cui nodi sono ben tipati, per definire in maniera formale le tecniche di gerazione di codice verranno utilizzate grammatiche con attributi. Risulta comunque facile riportare le azioni semantiche sull'albero sintattico in corrispondenza di non terminali assumendo una visita posticipata sinistra.

8.3 Dichiarazioni

Quando si esaminano le sequenze di dichiarazione di una procedura o di un blocco si è soliti allocare memoria al fine di memorizzare le variabili locali alla procedura/blocco che si sta considerando. Per ogni variabile locale viene creata un'entry nella ICT, relativa alla procedura in cui è dichiarata, in cui vengono specificate diverse informazioni come il *tipo* o l'*indirizzo relativo*. L'indirizzo relativo consiste in un offset calcolato a partire dalla base dell'area utilizzata per i dati statici (o dal campo per i dati locali) fino al record di attivazione.

Dichiarazioni di variabile in una procedura

La sintassi di un linguaggio come il C raggruppa tutte le dichiarazioni in una singola procedura per essere processate come fossero un gruppo. In questo caso viene utilizzata una varabile globale, detta *offset*, per tenere traccia del prossimo indirizzo di memoria libero. consideriamo la seguente grammatica:

$P \rightarrow D$
$D \rightarrow D; D | \mathbf{id} : T$
$T \rightarrow \mathbf{integer} | \mathbf{real} | \mathbf{array}[\mathbf{num}]\,\mathbf{of}\,T_1 | * T_1$

Come logico, la variabile globale *offset* deve essere inizializzata prima di qualsiasi altra operazione presente nel codice, in modo che le parti semantiche delle produzioni della grammatica agiscano su un qualcosa di concreto. Per fare ciò, è necessario modificare la produzione $P \rightarrow D$ con $P \rightarrow M\,D$, dove M è una produzione del tipo $M \rightarrow \varepsilon$ il cui solo compito è quello di far eseguire la regola semantica a lei associata di inizializzazione dell'offset prima di qualunque altra cosa.

E' importante ricordare che, siccome tutti gli attributi, che ora andremo a definire, della grammatica sopra descritta sono *sintetizzati*, la regola di valutazione dell'azione semantica di una produzione viene eseguita **dopo** quelle relative ai non terminali espressi della parte destra della stessa produzione, ovvero: se consideriamo la grammatica con attributi relativa alla produzione $A \rightarrow B\,C\ \{semanticaA\}$, *semanticaA* viene valutata **dopo** aver valutato in ordine *semanticaB* e *semanticaC*.

Detto ciò, per comprendere appieno la grammatica con attributi relativa alle dichiarazioni, è bene sapere che:

- la procedura $enter(name, type, offset)$ crea una nuova entrata nella ICT per *name*, associandogli il tipo *type* a partire dall'indirizzo in *offset*.
- il non terminale T possiede due attributi:
  1. *type*, che ne identifica il tipo.
  2. *width*, che specifica la quantità di memoria necessaria per allocare i valori di quel tipo.

Vediamo ora la definizione della grammatica con attributi:

$P \rightarrow M\,D$
$M \rightarrow \varepsilon$ {
    $offset := 0$
}
$D \rightarrow D_1; D_2$
$D \rightarrow id : T$ {
    $enter(id.name, T.type, offset)$;
    $offset := offset + T.width$
}
$T \rightarrow$ **integer** {
    $T.type = integer$;
    $T.width = 4$
}
$T \rightarrow$ **real** {
    $T.type = real$;
    $T.width = 8$
}
$T \rightarrow$ **array**[**num**] **of** T_1 {
    $T.type = array(\mathbf{num}.val, T_1.type)$;
    $T.width = \mathbf{num}.val \times T_1.width$
}
$T \rightarrow *T_1$ {
    $T.type = pointer(T_1.type)$;

$T.width = 4$
}

E' bene ricordare che, nei linguaggi come il C, è possibile dichiarare un puntatore prima della dichiarazione al tipo a cui punta. In questi casi è bene che tutti i puntatori abbiano la stessa dimensione.

Determinare l'informazione di scope

In un linguaggio che permette l'annidamento di procedure le variabili locali di ogni procedura possono essere assegnate con il metodo visto precedentemente. Quando, invece, si incontra una dichiarazione di procedura occorre definire una nuova ICT relativa a questa dichiarazione. Per permettere la dichiarazione di procedura, la grammatica precedente deve essere modificata come segue:

$P \rightarrow D$
$D \rightarrow D; D | \mathbf{id} : T | \mathbf{proc\ id}; D; S$

Per comprendere pienamente la grammatica con attributi, modificata per permettere anche le dichiarazioni di procedura, che verrà specificata tra poco, è bene sapere che:

- il non terminale S identifica uno statement (comando). Siccome non è necessario per l'argomento d'interesse attuale (dichiarazione di procedure), non ne verranno specificate le produzioni in questa sezione.
- la funzione $mktable(previous)$ crea una nuova ICT ritornando il puntatore alla nuova tabella creata. Quando è il *main* a creare una tabella, passa *null* come paramentro in input alla procedura.
- la procedura $enter(table, name, type, offset)$ crea una nuova entry per la variabile *name* nella ICT puntata da *table*, associandogli il tipo *type* a partire dall'indirizzo in *offset*.
- la procedura $addwidth(table, width)$ registra nell'ICT passata in input l'ampiezza dello spazio relativo ai simboli utilizzati nella ICT.
- la procedura $enterproc(table, name, newtable)$ crea una nuova entry, nell'ICT puntata da *table*, per la procedura *name*. L'argomento *newtable* viene fatto puntare all'ITC creata per *name*.
- si utilizzano due pile, salvate come variabili globali, *tblptr* e *offset* che puntano rispettivamente a quella che memorizza i puntatori alle tabelle dei simboli e a quella che specifica l'offset attuale. Le funzioni per interagire con le seguenti pile sono:
  - $top(stack)$ che ritorna l'elemento in cima alla pila.
  - $pop(stack)$ che ritorna l'elemento in cima alla pila, eliminandolo da quest'ultima.
  - $push(el, stack)$ che inserisce l'elemento *el* in cima alla pila *stack*.
- i non terminali M e N sono stati aggiunti alla grammatica per permettere una corretta inizializzazione delle tabelle e dell'offset.

La grammatica con attributi che gestisce le dichiarazioni di procedura è la seguente:

$P \rightarrow M\ D$ {
    $addwidth(top(tblptr), top(offset))$;
    $pop(tblptr)$;
    $pop(offset)$
}
$M \rightarrow \varepsilon$ {
    $t := mktable(null)$;
    $push(t, tblptr)$;
    $push(0, offset)$
}
$D \rightarrow D_1; D_2$
$D \rightarrow \mathbf{proc}\ \mathbf{id}; N\ D_1; S$ {
    $t := top(tblptr)$;
    $addwidth(t, top(offset))$;
    $pop(tblptr); pop(offset)$;
    $enterproc(top(tblptr), \mathbf{id}.name, t)$
}
$D \rightarrow id : T$ {
    $enter(top(tblptr), \mathbf{id}.name, T.type, top(offset))$;
    $top(offset) := top(offset) + T.width$
}
$N \rightarrow \varepsilon$ {
    $t := mktable(top(tblptr))$;
    $push(t, tblptr)$;
    $push(0, offset)$
}

Nomi dei campi nei record

La produzione

$$T \rightarrow \mathbf{record}\ D\ \mathbf{end}$$

se inclusa nella grammatica precedente, permette di definire un record contenente i campi specificati dal non terminale D. Prima di vedere la grammatica con attributi relativa è bene sapere che:

- il costruttore *record* viene utilizzato per definire il tipo di dato relativo.
- il non terminale L stato aggiunto alla grammatica per permettere una corretta inizializzazione delle tabelle e dell'offset.

La grammatica con attributi relativa è:

$T \rightarrow \mathbf{record}\ L\ D\ \mathbf{end}$ {
    $T.type := record(top(tblptr))$;
    $T.width := top(offset)$;
    $pop(tblptr)$;
    $pop(offset)$

}
$L \rightarrow \varepsilon$ {
$t := mktable(null)$;
$push(t, tblptr)$;
$push(0, offset)$
}

8.4 Comandi di assegnamento

In questa sezione si vuole definire una grammatica con attributi che permette di definire comandi (statement) di assegnamento. La grammatica finora generata con l'aggiunta degli statement di assegnamento su cui si lavorerà è la seguente:

$P \rightarrow D$
$D \rightarrow D; D | \mathbf{id} : T | \mathbf{proc}\ \mathbf{id}; D; S$
$T \rightarrow \mathbf{integer} | \mathbf{real} | \mathbf{array}[\mathbf{num}]\ \mathbf{of}\ T_1 | * T_1 | \mathbf{record}\ D\ \mathbf{end}$
$S \rightarrow \mathbf{id} := E$
$E \rightarrow E + E | E * E | - E | (E) | \mathbf{id}$

Prima di dare la sintassi della grammatica con attributi, è bene sapere che:

- $\mathbf{id}.name$ restituisce il nome dell'identificatore.
- *place* è un attributo relativo al non terminale E dove viene memorizzata la variabile temporanea che contiene il valore dell'espressione.
- *newtemp* è una funzione che restituisce ogni volta un **indirizzo** di una nuova variabile temporanea.
- $lookup(\mathbf{id}.name)$ verifica se c'è una entry per $\mathbf{id}.name$ nell'ICT t che è in cima alla pila delle tabelle. Se è presente in t ritorna il puntatore all'identificatore (più precisamente il suo indirizzo di memoria) altrimenti richiama ricorsivamente l'operazione di *lookup* con lo stesso $\mathbf{id}.name$ sulla ICT "genitore" puntata da *t*. Se al termine di questa ricerca ricorsiva non viene trovata nessuna entry per $\mathbf{id}.name$, ritorna *null*.
- $emit(''code'')$ è una procedura che scrive il codice a tre indirizzi. L'operatore "||" rappresenta la concatenazione di codice.
- *code* è un attributo, utilizzato da E e S, in cui viene inserito il codice generato.

La grammatica con attributi relativa è:

$S \rightarrow \mathbf{id} := E$ {
$p := lookup(\mathbf{id}.name)$;
if $p \neq null$ **then**
$S.code := E.code || emit(''p' :=' E.place'')$
else $error$
}
$E \rightarrow E_1 + E_2$ {
$E.place := newtemp$;

$E.code := E_1.code||E_2.code||$
$emit(''E.place' :=' E_1.place' +' E_2.place'')$
}
$E \rightarrow E * E$ {
$E.place := newtemp;$
$E.code := E_1.code||E_2.code||$
$emit(''E.place' :=' E_1.place' *' E_2.place'')$
}
$E \rightarrow -E_1$ {
$E.place := newtemp;$
$E.code := E_1.code||$
$emit(''E.place' :='' uminus' E_2.place'')$
}
$E \rightarrow (E_1)$ {
$E.place = E_1.place$
}
$E \rightarrow$ **id** {
$p := lookup(\textbf{id}.name);$
if $p \neq null$ **then**
$E.place := p;$
$E.code := \emptyset$
else *error*
}

Un importante precisazione va fatta in relazione alla funzione *newtemp*, che si occupa di creare una nuova variabile temporanea ogni qual volta che viene invocata. Il problema è che la presenza di troppi simboli temporanei rischia di riempire la tabella dei simboli con le loro dichiarazioni, richiedendo ulteriore spazio in memoria in corrispondenza di essi.

L'alternativa a questo problema è quella di utilizzare una variante della funzione *newtemp* che consenta il riuso di variabili temporanee. Questo è possibile grazie alla creazione di un array A in cui, in ogni indice, viene salvata una variabile temporanea. Associato a questo array A compare una variabile contatore c inizializzata a *zero*, che viene usata per sapere l'indice dell'array della prossima variabile demporanea utilizzabile. Come si vede dalla tabella 8.1, l'al-

comandi	**valore di c (dopo l'esecuzione del comando)**
	0
\$0:=a*b	1
\$1:=c+d	2
\$0:=\$0*\$1	1
\$1:=e*f	2
\$0:=\$0-\$1	1
x:=\$0	0

Tabella 8.1: Codice a tre indirizzi con variabili temporanee

goritmo utilizzato è dato dai seguenti punti da eseguire in maniera strettamente sequenziale per ogni linea l di codice a tre indirizzi:

1. per ogni variabile temporanea utilizzata come operando del codice a tre indirizzi di l,si decrementa c di uno.

2. se si ha bisogno di una nuova variabile temporanea per il codice di l si utilizza quella contenuta in $A[c]$. Fatto ciò, si incrementa c di uno.

Indirizzamento degli elementi di un array

Solitamente, un *array* è memorizzato in un blocco di locazioni consecutive. Se ogni elemento di un array occupa w byte, allora l'elemento di indice i è indirizzato da

$$base + (i - low) * w$$

dove *base* e *low* sono rispettivamente l'indirizzo e l'indice del primo elemento. Una variante di interesse di questa espressione può essere

$$(base - low * w) + i * w$$

ottenuta eseguendo opportune sostituzioni. La differenza tra le due è che il valore $base - low * w$, che rappresenta l'indirizzo del primo elemento dell'array, può essere calcolato a tempo di compilazione e viene, di conseguenza, salvato in un opportuno campo del record di attivazione della procedura che contiene la dichiarazione dell'array in corrispondenza del nome di quest'ultimo. Per permettere l'assegnamento su/da array la grammatica precedente viene modificata, per quel che riguarda le operazioni di assegnamento, come segue:

$S \rightarrow L := E$
$E \rightarrow E + E | E * E | -E | (E) | L$
$L \rightarrow \mathbf{id} | \mathbf{id}[E]$

Prima di dare la sintassi della grammatica con attributi, è bene sapere che:

- *place* è un attributo relativo al non terminale L dove viene memorizzata la variabile temporanea che contiene il valore dell'espressione.

- *offset* è un attributo relativo al non terminale L dove viene memorizzato l'indirizzo dell'elemento considerato nell'array ed è definito dall'equazione $i * w$.

- la funzione $count(name)$ dato in input il nome *name* dell'array, restituisce l'indirizzo del primo elemento dell'array $(base - low * w)$.

- la funzione $width(value)$ ritorna la dimensione in byte dell'elemento passato in input.

Data questa nuova grammatica, la relativa grammatica con attributi, relativa alle sole produzioni modificate, è definita come segue:

$S \rightarrow L := E$ {
  if $L.offset = null$ **then**
    $S.code := L.code || E.code || emit(''L.place' :=' E.place'')$
  else
    $S.code := E.code ||$

$emit(''L.place'['L.offset']'' :=' E.place'')$
}
$E \rightarrow L$ {
if $L.offset = null$ **then**
$E.place := L.place$
else
$E.place := newtemp;$
$E.code := L.code||$
$emit(''E.place' :=' L.place'['L.offset']' '')$
}
$L \rightarrow$ **id** {
$L.place := lookup(\textbf{id}.name);$
$L.offset := null$
}
$L \rightarrow \textbf{id}[E]$ {
$L.place := newtemp;$
$L.offset := newtemp;$
$L.code := emit(''L.place' :=' count(lookup(\textbf{id}.name))'')||$
$emit(''L.offset' := E.place * width(lookup(\textbf{id}.name))'')$
}

Conversioni di tipo nell'assegnamento

In pratica, gli operandi di una data espressione possono essere di tipo differente. In questo caso risulta importante applicare l'operazione giusta per permettere l'esecuzione corretta dell'espressione, convertendo opportunamente uno degli operandi al tipo dell'altro. Questa operazione di *coercizione* è permessa grazie ad un'apposita funzione di trasformazione. La coercizione è necessaria per permettere l'opportuno *overload* dell'operatore che intereagisce con i due operandi.

Queste due operazioni possono essere introdotte direttamente nella semantica del comando di assegnamento. Supponiamo che gli unici tipi di dato della grammatica siano *int* e *real* e sopponiamo di aggiungere alle operazioni del codice a tre indirizzi quella di coercizione *inttoreal*. In questo caso, l'azione semantica relativa alla produzione $E \rightarrow E + E$ diventa:

$E \rightarrow E_1 + E_2$ {
$E.place := newtemp;$
if $E_1.type = int$ **and** $E_2.type = int$ **then**
$E.code := E_1.code||E_2.code||$
$emit(''E.place' :=' E_1.place'int +' E_2.place'');$
$E.type := int$
elseif $E_1.type = real$ **and** $E_2.type = real$ **then**
$E.code := E_1.code||E_2.code||$
$emit(''E.place' :=' E_1.place'real +' E_2.place'');$
$E.type := real$
elseif $E_1.type = int$ **and** $E_2.type = real$ **then**
$u := newtemp;$
$E.code := E_1.code||E_2.code||$
$emit(''u' :=' inttoreal\ E_1.place'')||$

$emit(''E.place' :=' u'real +' E_2.place'');$
$E.type := real$
elseif $E_1.type = real$ **and** $E_2.type = int$ **then**
$u := newtemp;$
$E.code := E_1.code||E_2.code||$
$emit(''u' :=' inttoreal\, E_2.place'')||$
$emit(''E.place' :=' E_1.place'real +' u'');$
$E.type := real$
else $error$
}

Lo stesso identico procedimento si dovrebbe attuare anche per tutte le altre operazioni aritmetiche se la conversione di tipo si vuol implementare all'interno degli assegnamenti.

8.5 Espressioni booleane

Nei linguaggi di programmazione, le *espressioni booleane* servono al duplice scopo di:

- calcolare un *valore logico* (detto anche *valore di verità*).
- controllare il flusso di esecuzione all'interno dei comandi *ifthen*, *ifthenelse* e *while*.

Qui di seguito è riportata la grammatica finora utilizzata con l'aggiunta della sintassi relativa alle espressioni booleane:

$P \rightarrow D$
$D \rightarrow D; D | \mathbf{id} : T | \mathbf{proc\ id}; D; S$
$T \rightarrow \mathbf{integer} | \mathbf{real} | \mathbf{array}[\mathbf{num}]\ \mathbf{of}\ T_1 | * T_1 | \mathbf{record}\ D\ \mathbf{end}$
$S \rightarrow L := E$
$E \rightarrow E + E | E * E | -$
$E | (E) | L | \mathbf{true} | \mathbf{false} | E\ \mathbf{or}\ E | E\ \mathbf{and}\ E | \mathbf{not}\ E | \mathbf{id\ relop\ id}$
$L \rightarrow \mathbf{id} | \mathbf{id}[E]$

In relazione a questa nuova grammatica, il terminale **relop** definisce tutte le operazioni di confronto tra operandi numerici ($<, >, =, \leq, \geq, \neq$), mentre gli operatori **and** e **or** sono associativi a sinistra. Inoltre la precedenza fra gli operatori booleani è descritta come segue (dalla maggiore priorità alla minore): **not**, **and**, **or**.

Ci sono due metodi principali per rappresentare il valore di un'espressione booleana: il primo codifica il *true* e il *false* attraverso interi (rispettivamente 1 e 0); il secondo rappresenta il valore di un'espressione booleana mediante la posizione nel programma, ed è solitamente utilizzata quando si vuole valutare parzialmente[1] un'espressione.

[1] L'ottimizzazione che si può compiere valutando un'espressione parzialmente può essere pericolosa, soprattutto in relazione a dei possibili effetti collaterali non valutati. Per fare un esempio in linguaggio C:

```
#include <stdio.h>
```

Rappresentazione numerica

Supponendo che *1* rappresenti *true* e *0* rappresenti *false*, sotto l'ipotesi di valutazione completa dell'espressione booleana, un'espressione come a **or** b **and not** c viene rappresentata nel codice a tre indirizzi come segue:

```
t1 := not c
t2 := b and t1
t3 := a or t2
```

Un'espressione relazionale come $a < b$ è equivalente al comando condizionale *if* $a<b$ *then 1 else 0*, che può essere tradotta nel codice a tre indirizzi come:

```
100:  if a<b goto 103
101:  t := 0
102:  goto 104
103:  t := 1
104:
```

Facendo le seguenti considerazioni

- sia *nextstat* l'indirizzo del prossimo comando del codice a tre indirizzi nella sequenza di output.
- sia *place* un attributo relativo al non terminale E dove viene memorizzata la variabile temporanea che contiene il valore dell'espressione.
- sia *code* un attributo utilizzato da E in cui viene inserito il codice generato.
- siano le funzioni *emit* e *newtemp* definite come in precedenza: la prima scrive il codice a tre indirizzi (l'operatore "||" rappresenta la concatenazione di codice) mentre la seconda restituisce ogni volta un **indirizzo** di una nuova variabile temporanea.

la grammatica con attributi relativa alle operazioni booleane codificate con una rappresentazione numerica è la seguente:

```
#define TRUE 1
#define FALSE 0
typedef int bool;
bool pippo() {
  while(TRUE)
    ;
  return(TRUE);
}
int main() {
  bool a = TRUE;
  if(a || pippo()) {
    printf("Dentro l'if\n");
  }
  exit(0);
}
```

Come si può vedere, la funzione *pippo* una volta invocata cicla infinitamente senza mai ritornare. Se provate a compilare ed eseguire su shell il programma qui descritto esso conclude stampando a video la stringa "Dentro l'if". Questo perchè in C l'operazione *or* (||) in questo caso viene valutata parzialmente, ovvero: siccome il primo operando (a) è *true*, per determinare il risultato dell'operazione non serve valutare anche il secondo operando, seppur questo, se valutato, poteva far "bloccare" tutto il programma.

$E \rightarrow$ **true** {
    $E.place := newtemp$;
    $E.code := emit(''E.place' :='\ 1'')$
}
$E \rightarrow$ **false** {
    $E.place := newtemp$;
    $E.code := emit(''E.place' :='\ 0'')$
}
$E \rightarrow E_1$ **or** E_2 {
    $E.place := newtemp$;
    $E.code := E_1.code||E_2.code||$
      $emit(''E.place' :='\ E_1.place' or' E_2.place'')$
}
$E \rightarrow E_1$ **and** E_2 {
    $E.place := newtemp$;
    $E.code := E_1.code||E_2.code||$
      $emit(''E.place' :='\ E_1.place' and' E_2.place'')$
}
$E \rightarrow$ **not** E_1 {
    $E.place := newtemp$;
    $E.code := E_1.code||$
      $emit(''E.place' :='\ not' E_1.place'')$
}
$E \rightarrow$ **id$_1$ relop id$_2$** {
    $E.place := newtemp$;
    $E.code :=$
$emit(''\ 'if' \mathbf{id_1}.place\ \mathbf{relop}.op\ \mathbf{id_2}.place' goto'\ nextstat + 3'')||$
      $emit(''E.place' :='\ 0'')||$
      $emit(''\ 'goto' nextstat + 2'')||$
      $emit(''E.place' :='\ 1'')$
}
$E \rightarrow (E_1)$ {
    $E.place := E_1.place$;
    $E.code := E_1.code$
}

Per fare un esempio, la traduzione dell'espressione booleana $a < b\ or\ c < d\ and\ e < f$ è la seguente:

```
100:  if a<b goto 103
101:  t1 := 0
102:  goto 104
103:  t1 := 1
104:  if c<d goto 107
105:  t2 := 0
106:  goto 108
107:  t2 := 1
108:  if e<f goto 111
109:  t3 := 0
110:  goto 112
```

```
111:  t3 := 1
112:  t4 := t2 and t3
113:  t5 := t1 or t4
```

Nelle espressioni booleane implementate con la rappresentazione numerica, il problema della produzione di un codice a tre indirizzi che non valuta completamente un'espressione booleana se non è strettamente necessario (o col metodo della *valutazione parziale* o con il *short-circuit code*) non è risolvibile "al volo" perchè non si conosce l'indirizzo di codice a cui saltare poichè non è stato ancora generato.

Comandi di controllo di flusso

Ora verra considerata la traduzione in codice a tre indirizzi di espressioni booleane all'interno del contesto offerto dai comandi *ifthen*, *ifthenelse* e *while*. Qui sotto compare riproposta la solita grammatica con l'aggiunta dei comandi per il controllo di flusso:

$P \rightarrow D$
$D \rightarrow D; D | \mathbf{id} : T | \mathbf{proc\ id}; D; S$
$T \rightarrow \mathbf{integer} | \mathbf{real} | \mathbf{array}[\mathbf{num}]\ \mathbf{of}\ T_1 | * T_1 | \mathbf{record}\ D\ \mathbf{end}$
$S \rightarrow L := E | \mathbf{if}\ E\ \mathbf{then}\ S | \mathbf{if}\ E\ \mathbf{then}\ S\ \mathbf{else}\ S | \mathbf{while}\ E\ \mathbf{do}\ S | S; S$
$E \rightarrow E + E | E * E | -$
$E | (E) | L | \mathbf{true} | \mathbf{false} | E\ \mathbf{or}\ E | E\ \mathbf{and}\ E | \mathbf{not}\ E | \mathbf{id\ relop\ id}$
$L \rightarrow \mathbf{id} | \mathbf{id}[E]$

In tutti i casi relativi ai comandi di controllo di flusso, il non terminale E fa riferimento ad un'espressione booleana. Nella traduzione, si può assumere che il comando a tre indirizzi prodotto possa essere simbolicamente "etichettato" con un'apposita funzione.

Prima di definire la grammatica con attributi relativa ai comandi per in controllo di flusso, è bene sapere che:

- la funzione *newlabel* ritorna una nuova etichetta simbolica ogni volta che viene invocata.
- all'espressione booleana E vengono associati due attributi contenenti ambedue delle etichette: $E.true$ è l'etichetta a cui saltare quando il controllo di flusso è vero, mentre $E.false$ è quella a cui saltare si il controllo diflusso è falso.
- l'attributo *code* di E e S è utilizzato per inserivi il codice prodotto finora.
- l'attributo (ereditato) *next* di S contiene l'etichetta dell'istruzione successiva ad S.

Una definizione grafica della semantica dei comandi per il controllo di flusso può essere offerta dalla figura 8.1 mentre le azioni semantiche della grammatica con attributi relativa alle espressioni booleane ed ai comandi condizionali gestita con la tecnica delle etichette è la seguente:

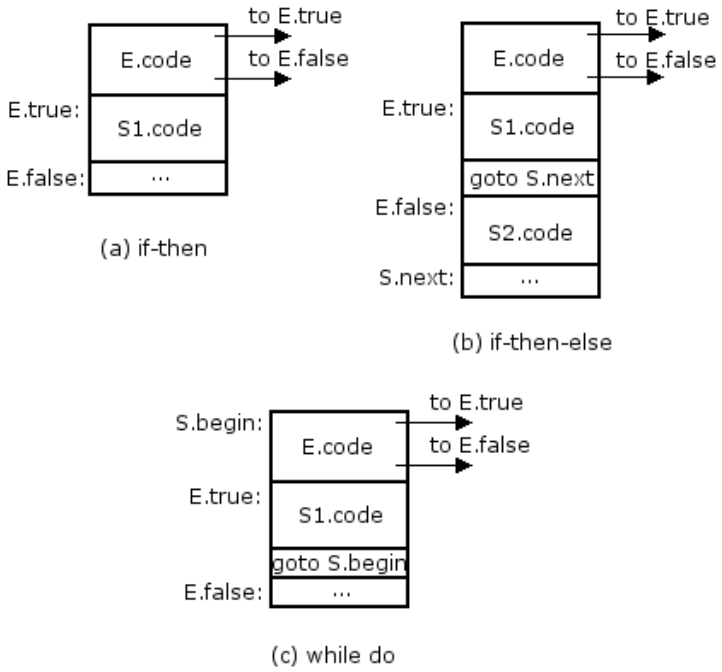


Figura 8.1: Codice per i comandi *ifthen*, *ifthenelse*, *whiledo*

$E \rightarrow \mathbf{true}$ {
  $E.code := emit(''\,'goto'\ E.true'')$
}
$E \rightarrow \mathbf{false}$ {
  $E.code := emit(''\,'goto'\ E.false'')$
}
$E \rightarrow E_1\ \mathbf{or}\ E_2$ {
  $E_1.true := E.true;$
  $E_1.false := newlabel;$
  $E_2;.true := E.true$
  $E_2.false := E.false;$
  $E.code := E_1.code||$
    $emit(''E_1.false' :\ '')||E_2.code$
}
$E \rightarrow E_1\ \mathbf{and}\ E_2$ {
  $E_1.true := newlabel;$
  $E_1.false := E.false;$
  $E_2.true := E.true;$
  $E_2.false := E.false;$
  $E.code := E_1.code||$
    $emit(''E_1.true' :\ '')||E_2.code$
}
$E \rightarrow \mathbf{not}\ E_1$ {
  $E_1.true := E.false;$

$E_2.false := E.true;$
$E.code := E_1.code$
}
$E \rightarrow \mathbf{id_1}\ \mathbf{relop}\ \mathbf{id_2}$ {
$E.code :=$
$emit(''if'\mathbf{id_1}.place\ \mathbf{relop}.op\ \mathbf{id_2}.place'goto'\ E.true'')||$
$emit(''goto'\ E.false'')$
}
$E \rightarrow (E_1)$ {
$E_1.true := E.true;$
$E_1.false := E.false;$
$E.code := E_1.code$
}
$S \rightarrow \mathbf{if}\ E\ \mathbf{then}\ S_1$ {
$E.true := newlabel;$
$E.false := S.next;$
$S_1.next := S.next;$
$S.code := E.code||$
$emit(''E.true' :\ '')||S_1.code$
}
$S \rightarrow \mathbf{if}\ E\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2$ {
$E.true := newlabel;$
$E.false := newlabel;$
$S_1.next := S.next;$
$S_2.next := S.next;$
$S.code := E.code||$
$emit(''E.true' :\ '')||S_1.code||$
$emit(''goto'\ S.next'');||$
$emit(''E.false' :\ '')||S_2.code$
}
$S \rightarrow \mathbf{while}\ E\ \mathbf{do}\ S_1$ {
$S.begin := newlabel;$
$E.true := newlabel;$
$E.false := S.next;$
$S_1.next := S.begin;$
$S.code := emit(''S.begin' :\ '')||E.code||$
$emit(''E.true' :\ '')||S_1.code||$
$emit(''goto'\ S.begin)$
}
$S \rightarrow S_1; S_2$ {
$S_1.next := newlabel;$
$S_2.next := S.next;$
$S.code := S_1.code||$
$emit(''S_1.next' :)||S_2.code$
}

Espressioni booleane e aritmetiche

E' importante rendersi conto che la grammatica per le espressioni booleane utilizzata finora è semplificata. In particolare risulta praticamente impossibile

permettere un'espressione booleana del tipo $(a + b) < c$. Nei linguaggi dove i valore *true* e *false* sono rappresentati numericamente con i valori *1* e *0*, $(a < b) + (b < a)$ può essere considerata un'espressione aritmentica con risultato *0* se a e b hanno lo stesso valore, *1* altrimenti.

Implementare il codice intermedio di queste espressioni è semplice quando le espressioni booleane sono rappresentate numericamente ma diventa un problema quando sono rappresentate attraverso etichette. Per riuscire a generare codice per quest'ultimo caso si suppone di avere nel non terminale E un attributo *type* che può assumere i valori di *arith* e *bool*, di modo che la semantica della produzione $E \rightarrow E + E$ diventa:

$E \rightarrow E_1 + E_2$ {
  $E.type := arith$;
  if $E_1.type = arith$ **and** $E_2.type = arith$ **then**
    $E.place := newtemp$;
    $E.code := E_1.code || E_2.code ||$
      $emit(''E.place' :=' E_1.place' int +' E_2.place'')$
  elseif $E_1.type = arith$ **and** $E_2.type = bool$ **then**
    $E.place := newtemp$;
    $E.true := newlabel$;
    $E.false := newlabel$;
    $E.code := E_1.code || E_2.code ||$
      $emit(''E_2.true' :' E_1.place' :=' E_2.place + 1'') ||$
      $emit(''' goto' nextstat + 1) ||$
      $emit(''E_2.false' :' E_1.place' :=' E_1.place'')$
  elseif ...

8.6 Backpatching

La strada più facile per implementare una *definizione sintattica diretta* è quella di usare due "passate": la prima costruisce l'albero sintattico per l'input, la seconda scorre l'albero con una visita posticipata sinistra e calcola le transizioni date dalle definizioni. Il problema maggiore per generare in una sola passata il codice per le espressioni booleane e per i comandi di controllo di flusso è dovuto al fatto che non si conoscono le etichette alle quali il controllo deve passare al momento dell'esecuzione di un comando di salto. Questo problema si può aggirare generando una serie di comandi di salto senza associare loro, almeno momentaneamente, alcuna etichetta a cui saltare. Le etichette verranno inserite quando si saprà esattamente dove saltare.

Questa operazione appena descritta prende il nome di *backpatching*. In sintesi, viene tenuta traccia di una lista di istruzioni di salto (*goto*) i cui argomenti vanno riempiti con lo stesso indirizzo. Gli argomenti verranno riempiti quando l'indirizzo viene generato. Al fine di manipolare la lista di indirizzi si utilizzano tre funzioni:

1. $makelist(i)$ crea una nuova lista che contiene l'indirizzo i e ritorna il puntatore a tale lista.

2. $merge(p1, p2)$ concatena le liste puntate da *p1* e *p2* ritornando il puntatore alla lista concatenata.

3. $backpatch(p, i)$ inserisce i come etichetta nelle istruzioni puntate dalla lista p.

Oltre a ciò, per comprendere la grammatica generativa relativa alle espressioni booleane nel caso si consideri la tecnica del backpatching, è bene sapere che:

- i non terminali M e N vengono introdotti nella grammatica con attributi. Il primo serve a memorizzare l'inidirizzo della prossima istruzione , il secondo è utile per creare liste ed emettere *goto*.
- il non terminale E ha due attributi, *truelist* e *falselist*, facenti riferimento rispettivamente alla lista dei salti da effettuare nel caso la condizione si verifichi o non si verifichi. Vengono usate per generare il codice di salto per le espressioni booleane.
- il non terminale M ha un attributo *quad* dove viene memorizzato l'indirizzo della prossima istruzione.
- la variabile globale *nextstat* memorizza l'indirizzo della prossima istruzione e viene aggiornata dall'invocazione della funzione *femit*.
- la funzione $femit("code")$ scrive su un file il contenuto di *code*.
- l'attributo *nextlist*, posseduto dai non terminali S e N, fa riferimento alla lista di salti condizionali e non che non sono stati "riempiti" dopo il completamento delle azioni semantiche realtive al terminale a cui fa riferimento.

Qui di seguito viene mostrata la grammatica con attributi relativa alle espressioni booleane implementata con il backpatching:

$M \rightarrow \varepsilon$ {
  $M.quad := nextstat$
}
$E \rightarrow$ **true** {
  $E.truelist := makelist(nextstat)$;
  $femit("'goto'\_")$
}
$E \rightarrow$ **false** {
  $E.falselist := makelist(nextstat)$;
  $femit("'goto'\_")$
}
$E \rightarrow E_1$ **or** E_2 {
  $backpatch(E_1.falselist, M.quad)$;
  $E.truelist := merge(E_1.truelist, E_2.truelist)$;
  $E.falselist := E_2.falselist$
}
$E \rightarrow E_1$ **and** E_2 {
  $backpatch(E_1.truelist, M.quad)$;
  $E.truelist := E_2.truelist$;
  $E.falselist := merge(E_1.falselist, E_2.falselist)$
}

$E \rightarrow \mathbf{not}\, E_1$ {
$E.truelist := E_1.falselist$;
$E.falselist := E_1.truelist$
}
$E \rightarrow \mathbf{id_1}\, \mathbf{relop}\, \mathbf{id_2}$ {
$E.truelist := makelist(nextstat)$;
$E.falselist := makelist(nextstat + 1)$;
$femit(''if'' \mathbf{id_1}.place\, \mathbf{relop}.op\, \mathbf{id_2}.place\, 'goto'\_'')$;
$femit(''goto'\_'')$
}
$E \rightarrow (E_1)$ {
$E.truelist := E_1.truelist$;
$E.falselist := E_1.falselist$
}
$N \rightarrow \varepsilon$ {
$N.nextlist := makelist(nextstat)$;
$femit(''goto'\_)$
}
$S \rightarrow \mathbf{if}\, E\, \mathbf{then}\, M\, S_1$ {
$backpatch(E.truelist, M.quad)$;
$S.nextlist := merge(E.falselist, S_1.nextlist)$
}
$S \rightarrow \mathbf{if}\, E\, \mathbf{then}\, M_1\, S_1\, N\, \mathbf{else}\, M_2\, S_2$ {
$backpatch(E.truelist, M_1.quad)$;
$backpatch(E.falselist, M_2.quad)$;
$S.nextlist :=$
$merge(S_1.nextlist, merge(N.nextlist, S_2.nextlist)$
}
$S \rightarrow \mathbf{while}\, M_1\, E\, \mathbf{do}\, M_2\, S_1$ {
$backpatch(S_1.nextlist, M_1.quad)$;
$backpatch(E.truelist, M_2.quad)$;
$S.nextlist := E.falselist$;
$femit(''goto'\, M_1.quad'')$
}
$S \rightarrow S_1; M\, S_2$ {
$backpatch(S_1.nextlist, M.quad)$;
$S.nextlist := S_2.nextlist$
}

8.7 Invocazioni di procedure

Consideriamo la seguente semplice grammatica per invocare procedure:

$S \rightarrow \mathbf{call}\, \mathbf{id}(Elist)$
$Elist \rightarrow Elist, E | E$

Un modo semplice per implementare un'invocazione di procedura è quello di utilizzare una coda *queue* (definita globalmente) in cui vengono inseriti i valori delle espressioni *Elist*. La routine che emette l'invocazione emette un'istruzione

param per ogni elemento della coda descrivendo la seguente grammatica con attributi:

$S \rightarrow$ **call id**$(Elist)$**{**
    $t := 0$
    for each *item p on queue* **do**
        $femit(''\,'param'\;p)$
        $t := t + 1$
    $femit(''\,'call'\;\mathbf{id}.place, t'')$
}
$Elist \rightarrow Elist, E$**{**
    append $E.place$ *to the end of queue*
}
$Elist \rightarrow E$**{**
    initialize *queue to contain* $E.place$
}

Elenco delle figure

1.1 Un compilatore 11
1.2 Sequenza di compilazione 12
1.3 Albero di parsing per "posizione := iniziale + percorso * 60" . . . 13
1.4 Albero sintattico per "posizione := iniziale + percorso * 60" . . . 13
1.5 Fasi di un compilatore 15

2.1 Interazione tra l'analizzatore lessicale e il parser 17
2.2 NFA per il simbolo ϵ 22
2.3 NFA per un qualsiasi simbolo dell'alfabeto a 22
2.4 NFA per l'espressione regolare $s|t$ 22
2.5 NFA per l'espressione regolare st 22
2.6 NFA per l'espressione regolare s^* 23
2.7 Creazione di un analizzatore lessicale con Flex 24

3.1 Modello per i parser LR 33
3.2 Costruzione del DFA data la grammatica G_{prova} 38
3.3 File di input per Bison 45
3.4 Precedenza "forzata" in *Bison* 46
3.5 Inclusione tra grammatiche 48

4.1 Dipendenza di attributi per la produzione $number \rightarrow number\ digit$ 52
4.2 Albero di sintassi astratta per la stringa $digit + digit * digit$ . . . 55
4.3 DAG di sintassi astratta per la stringa $digit + digit * digit$ 56

6.1 Albero sintattico etichettato per *defer(deref(q))* 63
6.2 Algoritmo di unificazione per *deref(deref(integer))* 67

7.1 Esempio di codice in Pascal 70
7.2 Tipica suddivisione della memoria a tempo di esecuzione. 71
7.3 Record di attivazione 72
7.4 Frammentazione nell'allocazione esplicita con blocchi variabili . . 78
7.5 Struttura di un blocco di memoria 79

8.1 Codice per i comandi *ifthen*, *ifthenelse*, *whiledo* 94

List of Algorithms

1	(Costruzione di Thompson) Da una espressione regolare ad un NFA	21
2	Costruzione di un DFA dato un NFA	23
3	Eliminazione della ricorsione sinistra	30
4	Algoritmo di parsing LR	35
5	Costruzione diretta di un DFA per una grammatica G	36
6	Chiusura per il parser LR(1)	40
7	Costruzione diretta di un DFA di un parser LR(1) per una grammatica G	41
8	Costruzione dell'automa LALR(1)	43
9	Tecnica di Burke-Fisher	47
10	Algoritmo di unificazione	66

Elenco delle tabelle

2.2 Parti di una stringa 19
2.3 Proprietà algebriche delle espressioni regolari 20

3.1 Tabella relativa alla costruzione del DFA data la grammatica G_{prova} 39

8.1 Codice a tre indirizzi con variabili temporanee 87