UNIVERSITÁ DEGLI STUDI DI PISA
Facoltá di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Scienze dell'Informazione

Tesi di Laurea

LLParser 2: UN ESECUTORE DI GRAMMATICHE AD ATTRIBUTI TOP-DOWN

Candidato:
Roberto Bomprezzi

Relatore:
Prof. Marco Bellia

Controrelatore:
Prof. Andrea Maggiolo

Correlatore:
Dott. Samuele Manfrin

A.A. 2002/2003

Contents

A mio padre Riccardo,
A mia madre Adele ed a
mia sorella Daniela,
che hanno fatto l'impossibile per me.
Alle persone care, che mi hanno sostenuto.
A Medjugorje.

Introduzione

LLParser é un sistema per la valutazione Top-Down di grammatiche attributate LL(1) ed e' il risultato della tesi in Scienze dell'Informazione del Dott. Samuele Manfrin, laureatosi a Pisa nell'Ottobre del 2001, seguito dal relatore Prof. M. Bellia.

Obiettivo del presente lavoro di tesi é stato la realizzazione di una nuova versione di LLParser, denominata LLParser 2. L'introduzione di nuove funzionalitá e la eliminazione di vincoli rendono LLParser 2 uno strumento ancor piú potente per l'analisi, la traduzione e la generazione di codice.

Le motivazioni che spinsero alla ideazione di LLParser e successivamente al passaggio a LLParser 2 sono di due tipi. Originariamente si pensó ad uno strumento didattico da poter offrire agli studenti del corso di Compilatori, ma giá durante la realizzazione di LLParser scaturí la possibilitá di ottenere un ambiente di lavoro adeguato anche alle piú esigenti attivitá di progetto per la sperimentazione di linguaggi.

LLParser 2 si colloca in quella categoria di strumenti che affrontano esclusivamente problematiche di tipo Top-Down, consentendo la definizione, l'analisi e l'uso di Grammatiche LL(1), di Grammatiche L-Attributate e di Schemi di Traduzione Discendente.

Questo lavoro di tesi é diviso nei seguenti capitoli:

Capitolo 1: parla inizialmente della storia di LLParser e poi evidenzia sommariamente le sue caratteristiche principali. Si conclude elencando i limiti di LLParser che hanno indotto alla realizzazione della nuova versione LLParser 2.

Capitolo 2: descrive i concetti teorici che sono alla base delle problematiche che si sono dovute affrontare. Ricorderemo quindi le strutture che ci serviranno, come le grammatiche, le derivazioni, le attributate, i linguaggi intermedi, e tutto ció che é stato utilizzato durante il progetto.

Capitolo 3: spiega quali sono state le scelte progettuali effettuate. Si inizierá spiegando cosa é stato mantenuto e perché, per poi passare alle novitá introdotte con le vari estensioni.

Capitolo 4: entra nei dettagli implementativi delle estensioni introdotte per l'analisi statica, spiegando le variazioni subite dalle strutture dati e dal codice.

Capitolo 5: spiega i dettagli implementativi relativi alle estensioni introdotte per la Generazione di codice per effetti laterali ad una passata, evidenziando le modifiche che si sono dovute apportare al sistema.

Capitolo 6: chiude il lavoro ricordando cosa é stato fatto, valutando le modalitá seguite e proponendo eventuali nuovi sviluppi futuri del progetto, nell'ottica di far diventare LLParser 2 uno strumento sempre piú completo e funzionale.

Appendice A: contiene tutto il quarto capitolo della tesi del Dott. Samuele Manfrin. Puó essere utile consultarlo durante la lettura del capitolo *Estensioni per l'analisi statica*.

Appendice B: contiene l'intero codice scritto per realizzare LLParser 2, che consiste in un client ed un server. Il client é codice HTML che fa uso anche di JavaScript e CCS per le sue componenti dinamiche. Il server é puro codice Perl che implementa il cuore del sistema LLParser 2.

Al termine di questa introduzione si vuole fare un breve cenno relativamente alla decisione presa di aderire al Progetto Sperimentale ETD. Si tratta di un progetto dell'Universitá di Pisa che ha come obiettivo la presentazione, conservazione e disponibilitá in forma elettronica delle tesi discusse

nell'Ateneo pisano. Conseguenza di questa scelta é la possibilitá, offerta ai laureandi della classe di Scienze e di Ingegneria, di poter disporre del presente lavoro collegandosi all'indirizzo www.unipi.it/etd e seguendo la procedura di consultazione descritta nell'help del sito.

Inoltre, trattandosi LLParser 2 di un sistema che prevede il suo utilizzo tramite un comune browser, é possibile usarlo collegandosi agli indirizzi http://projects.cli.di.unipi.it/bomprezz e http://www.llparser.org/.

Chapter 1

Storia di LLParser

In questo capitolo presentiamo la prima versione di LLParser, rilasciata nel 2001, ed in particolar modo spieghiamo i motivi che hanno spinto alla sua realizzazione e quali sono le sue principali caratteristiche.

1.1 Come nasce

LLPARSER consiste in un sistema per la valutazione top-down di grammatiche LL(1) con attributi sintetizzati ed ereditati, ed e' il risultato del lavoro di tesi in Scienze dell'Informazione del Dott. Samuele Manfrin [2], laureatosi a Pisa nell'Ottobre del 2001.

LLParser é scaturito dall'esigenza di disporre di uno strumento, versatile e facile da usare, che permettesse agli utenti di concentrarsi esclusivamente sulle problematiche relative al mondo dei compilatori, senza richiedere ulteriori sforzi nell'apprendimento di un nuovo strumento di lavoro. Infatti, per sfruttare completamente le potenzialitá di LLParser, é sufficiente conoscere due metalinguaggi, quello per descrivere le produzioni di una grammatica e quello per scrivere le azioni semantiche da associare alle produzioni, che sono ampiamente trattati ed utilizzati durante il corso di Compilatori.

1.2 Caratteristiche principali

A questo punto presentiamo le caratteristiche tecniche e funzionali principali del sistema.

1.2.1 Modalitá d'uso

Uno dei punti di forza di LLParser consiste nella facilitá d'uso. Infatti, all'utente non é richiesta l'installazione di alcun software, dato che per utilizzare LLParser é sufficiente avere un browser e la possibilitá, ormai diffusissima, di accedere ad internet.

L'interfaccia grafica é essenziale e molto intuitiva, in modo da non demotivare l'utente con richieste di settaggi od altro. La GUI, infatti, consiste in un form che permette l'inserimento, in due campi distinti, dei dati da fare analizzare. In un primo campo, quello piú grande, si devono inserire le produzioni della grammatica con le eventuali azioni semantiche. Nel secondo campo, che consiste in un sola linea di testo, si deve inserire la frase del linguaggio che si vuole analizzare.

Sono presenti anche quattro check box che consentono di personalizzare l'esecuzione di LLParser, a seconda delle necessitá dell'utente. Infatti, selezionandoli opportunamente, é possibile richiedere o meno:

- l'esecuzione di azioni semantiche
- la visualizzazione dei valori degli attributi
- la visualizzazione dei valori delle variabili d'ambiente
- l'output delle emit.

Per facilitare il primo impatto con il sistema, sono presenti alcuni link che consentono il caricamento delle diverse componenti dell'interfaccia grafica con degli esempi giá testati.

Terminato l'input, é sufficiente premere il tasto di richiesta dell'analisi. Il lato client invia al server le informazioni tramite CGI e resta in attesa

dell'output. I risultati vengono ricevuti e visualizzati in una nuova pagina, che, a seconda dei check box precedentemente settati, puó assumere diversi formati.

In ogni caso, la pagina di output conterrá sempre il riepilogo dei dati inseriti, il risultato del calcolo dei first, dei follow e della tabella del parser. Se é stata richiesta l'analisi di una frase, sará possibile visualizzare anche tutti i passi seguiti dal parser, ovvero, vedere l'evoluzione degli stack di sistema e della coda di input.

1.2.2 Cosa permette di fare

LLParser puó essere usato dagli studenti del corso di Compilatori sia come strumento per facilitare la comprensione della teoria [1] sia come correttore automatico di esercizi svolti manualmente.

Piú precisamente é possibile utilizzarlo per svolgere le seguenti attivitá:

- Definire grammatiche LL(1)
- Calcolare i first e i follow di grammatiche
- Verificare le condizioni di LL(1)-soddisfacibilitá di grammatiche context free
- Calcolare la tabella di analisi di grammatiche LL(1)
- Verificare la tecnica di derivazione leftmost
- Esercitarsi sull'uso di un automa a pila
- Definire grammatiche attributate, con attributi sia ereditati che sintetizzati
- Calcolare ed analizzare la propagazione degli attributi
- Effettuare dell'analisi statica.

Le suddette attivitá sono possibili grazie alla grande quantitá di dati che LLParser restituisce come output alla fine della sua elaborazione, peraltro con un minimo sforzo da parte dell'utente.

1.2.3 Limiti

Giá nel 2001, sebbene fossero tanti i pregi di LLParser, si erano prospettati possibili ulteriori sviluppi, nel tentativo di renderlo piú amichevole nei confronti degli studenti. Si pensó inoltre di completarlo riguardo a certi aspetti non trattati, al fine di ottenere uno strumento utile anche agli esperti del settore che volessero utilizzarlo nella sperimentazione di linguaggi.

Il primo limite che saltó subito all'occhio fu il vincolo sull'uso dei simboli grammaticali che erano cosí definiti:

- Non Terminali: [A-Z]
- Terminali: [a-z]

Si avevano contemporaneamente un vincolo di quantitá, non particolarmente grave per l'uso didattico, ed uno di espressivitá un pó piú fastidioso.

Altro limite derivava dalla fase di generazione di codice intermedio che era implementata mediante la libera visualizzazione di testo tramite una emit. Ció non consentiva quindi una effettiva generazione di codice per effetti laterali e codice ad una passata con uso della tecnica del backpatch.

Un altro aspetto, a suo tempo giustamente non considerato per il tipo di prodotto che si voleva realizzare, fu l'assenza di un preprocessore di analisi lessicale. Questo peró obbligava gli utenti, al momento dell'inserimento di una frase del linguaggio da analizzare, a non poter ragionare in termini di lessemi e di doversi abituare a ragionare in termini di token.

Chapter 2

Preliminari

In questo capitolo enunceremo concetti e definizioni che sono alla base di LLParser 2. Non si ha l'intenzione di spiegare dettagliatamente la teoria, ma solo di richiamare termini che saranno utilizzati durante la trattazione del lavoro. Per una descrizione completa dell'argomento si puó fare riferimento a [1, 3, 5].

Quindi in questo capitolo tratteremo gli aspetti che riguardano la metodologia Top-Down utilizzata. Durante l'implementazione del nuovo sistema abbiamo usato particolari tecniche di programmazione Perl, che saranno segnalate e descritte ogni volta che si presenteranno nei prossimi capitoli [6, 7].

2.1 Grammatiche Context-Free

La sintassi dei costrutti dei linguaggi di programmazione puó essere descritta tramite delle Grammatiche Context-free.

Una Grammatica Context-free é una quadrupla $G =< V, \Sigma, S\epsilon V, P >$ cosí definita [1]:

- V é l'insieme dei Non Terminali

- Σ é l'insieme dei Terminali

- S é un simbolo Non Terminale scelto come Simbolo Distinto

- P é un insieme di Produzioni.

Le produzioni di una grammatica specificano il modo con il quale i Terminali ed i Non Terminali possono essere combinati per formare stringhe. Ogni produzione consiste di un Non Terminale, seguito da "::=", seguito da una stringa di Terminali e Non Terminali.

Ci sono diversi modi per descrivere il processo mediante il quale una grammatica definisce un linguaggio:

- Con la generazione di Parse-Tree
- Con la Derivazione di stringhe.

Formalmente un Parse-Tree é cosí definito [1]:

- La radice é il Simbolo Distinto S
- Ogni nodo interno é etichettato con un Non Terminale
- Ogni nodo foglia é etichettato con un Terminale
- Se Y é un Non Terminale che etichetta un nodo, ed $X_1, X_2, \ldots, X_n$ sono le etichette dei suoi figli, allora $Y ::= X_1 X_2 \ldots X_n$ é una produzione della grammatica.

Le foglie di un Parse-Tree, lette da sinistra verso destra, formano la stringa generata dal Simbolo Distinto presente nella radice. Il procedimento per costruire il Parse-Tree, partendo dalla stringa del linguaggio in esame, é detto parsing della stringa.

LLParser 2 é basato principalmente sul concetto di Derivazione. L'idea di fondo é che una produzione é trattata come una regola di riscrittura nella quale il Non Terminale alla sinistra é rimpiazzato con la stringa alla destra della produzione. Si dice che $\alpha A \beta$ deriva $\alpha \gamma \beta$ se $A ::= \gamma$ é una produzione della grammatica, e questa relazione si denota con $\alpha A \beta \Rightarrow \alpha \gamma \beta$.

Sono utilizzate altre due relazioni, che scaturiscono direttamente dal concetto di Derivazione. Definiamo [1]:

- $\Rightarrow^*$ come la chiusura riflessiva e transitiva di $\Rightarrow$
- $\Rightarrow^+$ come la chiusura transitiva di $\Rightarrow$

I Parse-Tree risultano molto utili per la visualizzazione degli oggetti derivabili da una grammatica, ma non mostrano l'ordine delle derivazioni. Invece $\Rightarrow^*$ e $\Rightarrow^+$ hanno il pregio di mostrare l'ordine di derivazione delle stringhe.

Comunque, non siamo ancora giunti al tipo di Derivazione da noi usata. Infatti, la vera caratteristica del Riconoscimento Top-Down é quella di eseguire "Derivazioni left-most", che consistono in Derivazioni nelle quali ad ogni passo si deriva il Non Terminale piú a sinistra.

Questo concetto é molto importante perché consente di diminuire il lavoro di un parser. Per spiegare questa affermazione abbiamo bisogno di una definizione e di una proprietá [1]:

- Definizione: Sia data una grammatica $G =< V, \Sigma, S\epsilon V, P >$. Definiamo *Forma Sentenziale Sinistra* della grammatica G l'insieme LSF_G di "tutte le stringhe di Terminali e Non Terminali derivabili da G usando solo derivazioni left-most"
- Proprietá: Se $w\epsilon L(G)$ allora esiste una derivazione left-most tale che $S \Rightarrow^+ w$.

Questa proprietá ci assicura che per vedere se $w\epsilon L(G)$ non importa fare tutte le derivazioni possibili, basta limitarci alle derivazioni left-most, infatti $L(G) = \{w\epsilon\Sigma^* | S \Rightarrow^+ w\} = \{w\epsilon\Sigma^* | w\epsilon LSF_G\}$, ed é abbastanza intuitivo vedere che il secondo insieme é molto piú piccolo del primo. In questo modo il carico di lavoro di un parser é notevolmente diminuito.

2.2 Grammatiche LL(1)

Per consentire un corretto funzionamento di un parser Top-Down, é importante che le grammatiche da analizzare verifichino delle proprietá [1]. Innanzitutto sarebbe importante utilizzare una grammatica non ambigua. Una

grammatica é ambigua quando puó generare piú parse-tree per una qualche stringa. Trasformando le produzioni incriminate in altre equivalenti, ma meglio strutturate, é possibile risolvere questo tipo di problema.

Esistono inoltre altri tre tipi di trasformazioni, che si devono utilizzare per risolvere altrettanti problemi:

- Si deve eliminare la Ricorsione Sinistra, per evitare la non terminazione di un procedimento di parsing top-down. Nel caso di ricorsione semplice, la produzione $A ::= A\alpha|\beta$ deve essere sostituita dalle produzioni $A ::= \beta B$ e $B ::= \alpha B|\epsilon$. Nel caso di mutua ricorsione, una possibilitá é di sostituire le produzioni $A ::= B\alpha|\gamma$ e $B ::= A\beta|\delta$ con la coppia $A ::= B\alpha|\gamma$ e $B ::= (B\alpha|\gamma)\beta|\delta$ facendo attenzione ad eliminare la ricorsione semplice che é stata introdotta

- Si deve Fattorizzare a Sinistra per consentire un funzionamento deterministico del parser. In questo caso, la produzione $A ::= \alpha\beta|\alpha\gamma$ deve essere sostituita con le due produzioni $A ::= \alpha B$ e $B ::= \beta|\gamma$

- Si deve eliminare la Stella di Kleene, sostituendo la produzione $A ::= \gamma^*$ con $A ::= \gamma A|\epsilon$.

L'uso di queste tecniche non é sempre immediato. In particolar modo, le tecniche per l'eliminazione della mutua ricorsione possono richiedere parecchie trasformazioni, prima di riuscire nell'intento. Tutto dipende, ovviamente, dalla struttura della grammatica in esame. Comunque, con questi pochi accorgimenti, é possibile scrivere grammatiche perfettamente adatte ad una analisi di tipo Top-Down Predittivo.

Queste grammatiche vengono chiamate LL(1): la prima L indica che la scansione dell'input avviene da sinistra verso destra; la seconda indica l'utilizzo di derivazioni left-most; l'1 sta ad indicare che un parser top-down, che prende in input la grammatica, é in grado di determinare il suo prossimo passo osservando solamente il proprio stato interno ed in piú "un" solo simbolo in input.

2.3 Parser Predittivo Top-Down

Il parser da noi realizzato é di tipo Generativo/Adattivo. Con Generativo si intende il fatto che, per cambiare la grammatica da analizzare, basta sostituire la tabella del parser con quella relativa alla nuova grammatica. Con Adattivo si intende dire che la sostituzione di una sola produzione della grammatica comporta solo il cambiamento di una riga della tabella. Questo implica, quindi, il non dover assolutamente toccare le routine che implementano il funzionamento del parser, a differenza di quello che accadrebbe con un Parser Predittivo a Discesa Ricorsiva.

Il parser Predittivo Top-Down Generativo/Adattivo si basa su di un Automa a Pila, dalle seguenti componenti [1]:

- L'Input, che é un buffer che contiene la stringa da analizzare, seguita dal terminatore $
- Lo Stack, che é una pila che contiene una sequenza di simboli della grammatica, con in fondo il terminatore $
- La Tabella del Parser, che é un vettore bidimensionale M[A,a], dove A é un Non Terminale, mentre a é un Terminale o $. La Tabella é cosí definita: $\forall A ::= \alpha$
  - $\forall$ Terminale $a \in (first(\alpha) - \{\epsilon\})$ si pone $M(A, a) := (A ::= \alpha)$
  - Se $\epsilon \in first(\alpha)$ allora $\forall$ Terminale $a \in follow(A)$ si pone $M(A, a) := (A ::= \alpha)$
  - Tutte le rimanenti celle sono *error*
- Il Programma del Parser, che si comporta come segue:
  - Se top(Stack)=lookahead=$ allora il Parser passa in uno stato finale di riconoscimento della stringa
  - Se top(Stack)=lookahead≠$ allora si eseguono le istruzioni "pop(stack);lookahead:=nextToken"

- Se top(Stack) é un Non Terminale A, allora si eseguono le istruzioni "pop(Stack);push(α,Stack)", sempre se la tabella contiene M(A,lookahead)=(A::=α).

Il comportamento del Parser é descritto in termini delle sue configurazioni, che sono rappresentate da coppie (Stack,Input rimanente). La stringa da riconoscere é inserita nel buffer Input, seguita da $. Lo Stack servirá per far andare avanti il riconoscimento attraverso le varie sentenziali sinistre che si devono derivare per arrivare alla stringa da riconoscere.

Per poter realizzare un parser di questo tipo, come prima cosa si deve trasformare la grammatica, se necessario, per ottenerne una di tipo LL(1), come si é giá detto precedentemente. A questo punto si puó costruire la Tabella M del Parser, facendo uso delle funzioni first e follow.

Il calcolo di first(X) é cosí definito:

- $first(a) = \{a\}\ \forall a \in Terminali$
- $first(\epsilon) = \{\epsilon\}$
- $first(X) = \cup_{\{X::=\alpha \in P\}} first(\alpha)\ \forall X \in Non\ Terminali$
- $first(X_1 \dots X_n) = [\cup_{\{i<k\}}(first(X_i) - \{\epsilon\})] \cup first(X_k)$ per k massimo intero tale che: $X_1 \dots X_{k-1} \Rightarrow^* \epsilon$ oppure k=n

Invece il follow(A) é definito come segue:

- $follow(A) = \{\cup_{\{B::=\alpha A\beta\}}(first(\beta) - \{\epsilon\})\} \cup \{\cup_{\{B::=\alpha A\beta | \beta \Rightarrow^* \epsilon\}} follow(B)\} \cup \{\cup_{\{B::=\alpha A\}} follow(B)\}\ \forall A \in Non\ Terminali$
- \$ $\in follow(S)$ se S é il *Simbolo Distinto*.

Le due funzioni appena definite, oltre a permettere la costruzione della tabella del parser, consentono di dare un'ulteriore definizione di Grammatica LL(1). Infatti, una grammatica é LL(1) se valgono le seguenti due proprietá [1]:

- $\forall A ::= \alpha|\beta\ first(\alpha) \cap first(\beta) = \{\}$
- $\forall A ::= \alpha|\beta$ se $\alpha \Rightarrow^* \epsilon$ allora $first(A) \cap first(\beta) = \{\}$.

2.4 Grammatiche attributate

Le Grammatiche Attributate permettono contemporaneamente il riconoscimento di stringhe e l'esecuzione di azioni semantiche.

Una Grammatica Attributata é una coppia (G,A) dove G é una Grammatica Context-free, mentre A é una collezione di azioni associate alle varie produzioni di G. Le azioni sono espressioni scritte in un qualche metalinguaggio, che possono coinvolgere:

- Operazioni del metalinguaggio
- Funzioni scritte nel metalinguaggio
- Attributi della G, associati alle istanze dei simboli grammaticali
- Token del linguaggio che corrispondono ad indirizzi nella Symbol Table, dove si saranno inseriti i rispettivi valori.

Le Grammatiche Attributate sono potenti, infatti, dato che le azioni sono esprimibili usando un qualunque metalinguaggio, che potrebbe essere anche un tradizionale Linguaggio di Programmazione, potremmo essere in grado di calcolare tutte le funzioni calcolabili. Questo vuol dire che, con questo tipo di grammatiche, potremmo fare qualsiasi tipo di analisi necessaria, sempre facendoci guidare dal nostro riconoscitore.

Bisogna fare attenzione a come si definiscono gli attributi. Infatti, definire male un attributo in una Grammatica Attributata é altrettanto grave, come scrivere un programma che si blocca. Le proprietá che le azioni ben definite devono soddisfare sono:

- Le parti sinistre delle azioni devono essere attributi di simboli presenti nella produzione associata
- Le parti destre delle azioni devono essere espressioni che usano attributi di simboli presenti nella produzione associata

- Le parti destre delle azioni devono essere espressioni che usano operatori del metalinguaggio o funzioni scritte nel metalinguaggio. Nel caso in cui queste operazioni abbiano degli effetti laterali, le Grammatiche ad Attributi verrebbero dette SDD

- Le parti destre delle azioni devono essere espressioni effettivamente calcolabili al momento della derivazione della produzione. Questo vuol dire che gli attributi che entrano in gioco devono giá essere disponibili.

Per garantire il corretto uso degli attributi, introduciamo due nozioni [1]:

- Grafo delle Dipendenze
- Topological Sorting.

Il Grafo delle Dipendenze é un grafo che sovrapponiamo al Parse-tree, in modo tale che ad ogni nodo del Parse-tree corrisponda un nodo attributo, se il nodo del Parse-tree ha un attributo. Esisterá un arco dal Nodo Attributo 2 al Nodo Attributo 1 se il Nodo Attributo 1 é definito in funzione del Nodo Attributo 2. I Nodi Attributo costanti sono essenziali per la corretta definizione delle Grammatiche Attributate, perché corrispondono alla inizializzazione dei valori degli attributi in gioco. Il Grafo delle Dipendenze consente di evidenziare gli attributi mal definiti e, in particolare modo, permette di verificare se a partire dagli attributi costanti si riescono a calcolare tutti gli altri attributi presenti.

Il Topological Sorting é un ordinamento parziale dei nodi che rispetta le dipendenze tra i nodi. Quando si riesce a trovare un Topological Sorting su un Grafo delle Dipendenze, siamo sicuri che esiste un ordine di visita dei nodi che consenta di calcolare tutti gli attributi. Su uno stesso Grafo delle Dipendenze possono esistere piú Topological Sorting.

Ricapitolando: un Grafo delle Dipendenze e un Topological Sorting, insieme, danno un controllo di esecuzione, permettendo di stabilire se una Grammatica Attributata é ben scritta oppure no. La cosa interessante é che, a seconda di come é fatto il Grafo delle Dipendenze, possiamo dire se la sequenza di esecuzione é compatibile con quella di un riconoscitore Top-Down.

Gli attributi possono essere di due tipi [1]:

- Attributo Sintetizzato, se é definito in funzione di attributi di nodi figlio
- Attributo Ereditato, se é definito in funzione di attributi del nodo padre e/o nodi fratello.

Gli Attributi Sintetizzati si usano per descrivere proprietá di semantica composizionale che riguardano la struttura dell'albero. Gli Attributi Ereditati si usano per indagare su proprietá contestuali. Abbiamo diversi metodi che consentono la valutazione degli Attributi di una Grammatica Attributata; qui presentiamo solo quello da noi usato, ovvero il Metodo Oblivious.

Con il Metodo Oblivious si valutano le azioni nel momento in cui si deriva una produzione, quindi, derivazione della stringa ed esecuzione di tutte le azioni associate, avvengono in un unico passo. La cosa interessante di questo metodo é che l'ordine di esecuzione delle azioni scaturisce direttamente dal tipo di analisi che si sta eseguendo, nel nostro caso, un'analisi di tipo Top-down.

Perché si possa applicare il Metodo Oblivious nella Grammatica Attributata, le definizioni degli attributi devono essere tali da essere compatibili con una visita Depth-First del Grafo delle Dipendenze. Questo perché l'analizzatore Top-Down effettua una visita Depth-First del Parse-Tree.

Le Grammatiche Attributate che possono essere valutate con il Metodo Oblivious sono dette [1]:

- L-Attributate se gli attributi di un nodo sono sintetizzati, oppure ereditati che dipendono dagli attributi ereditati del nodo padre o dai sintetizzati dei nodi fratelli che lo precedono
- S-Attributate se gli attributi di un nodo possono essere solo sintetizzati. Le S-Attributate sono un caso particolare di L-Attributate.

É possibile ottenere un Esecutore Oblivious a partire da un Parser, effettuando alcune modifiche a quest'ultimo. Si devono associare allo Stack di un Analizzatore Top-Down altri due stack, uno per gli attributi ereditati e l'altro

per gli attributi sintetizzati. Perché le cose funzionino, bisognerá assicurare la seguente proprietá: gli attributi presenti sui due Stack degli Attributi, si riferiscono sempre al simbolo presente sul top dello Stack di Controllo del Parser. Inoltre, sullo Stack del Parser, oltre ad esserci i simboli della grammatica, ci potranno essere anche le azioni semantiche associate alle produzioni; questo significa che il Parser dovrá incorporare delle nuove routine, che siano in grado di valutare le azioni semantiche.

Per permettere un corretto funzionamento di un Esecutore Oblivious, peró, non é sufficiente utilizzare una Grammatica L-Attributata. Infatti, é necessario un metodo che consenta di evidenziare la differenza tra attibuti ereditati ed attributi sintetizzati, ed in particolar modo un metodo che consenta di esplicitare il momento in cui un attributo deve essere calcolato dall'esecutore. Per permettere questo, é necessario usare gli Schemi di Traduzione, che consistono in grammatiche attributate che consentono di immergere azioni semantiche all'interno delle produzioni stesse e non solo in fondo. In questo modo sará possibile indicare, all'esecutore, quali sono gli attributi ereditati e in quale punto della produzione dovranno essere calcolati. Gli attributi sintetizzati continueranno, invece, ad essere definiti esclusivamente in fondo a ciascuna produzione della grammatica.

2.5 Generazione di Codice Intermedio

Supponiamo di trovarci nella seguente situazione:

- abbiamo superato l'analisi lessicale, quindi il programma é stato diviso in token

- abbiamo eseguito il parsing del programma, quindi conosciamo la struttura del programma ed ogni suo costrutto.

Se abbiamo superato le due fasi, vuol dire che il programma é libero da errori grammaticali, quindi possiamo iniziare ad eseguire la traduzione, che potrá

essere eseguita in diversi modi. Infatti, alcune possibili scelte sono le seguenti [3]:

- Si potrebbe generare del codice intermedio direttamente durante il parsing del programma

- Si potrebbe generare una rappresentazione intermedia e solo successivamente tradurla in un codice intermedio, o addirittura nel codice scritto nel linguaggio target.

La scelta dipende, in parte, da quante ottimizzazioni si vogliono eseguire. Infatti, una rappresentazione intermedia, tipo quella dei Syntax-Tree, potrebbe offrire una certa quantitá di informazioni utili per l'ottimizzazione, mentre il codice oggetto non offre grandi possibilitá di ottimizzazioni. Comunque, tutte le possibili alternative hanno in comune una cosa: si basano sulle informazioni scoperte durante il parsing.

É importante notare che esiste sempre una corrispondenza tra "Produzioni della Grammatica" e "Computazioni elementari del linguaggio" [3]. Per esempio, se viene derivata una produzione che contiene un'operazione di somma, allora il programma sorgente, in qualche parte del codice, conterrá un'addizione.

Questo vuol dire che possiamo associare una semantica ad ogni produzione. Facendo uso di questa associazione tra produzioni e loro significato, abbiamo trovato il primo passo per eseguire la traduzione di un programma. Poiché la sequenza delle produzioni guida la Generazione del Codice Intermedio, chiameremo questa tecnica Syntax-Directed Translation.

La computazione delle azioni associate alle produzioni, assegna un significato ad ogni produzione, cosí queste operazioni sono dette Azioni Semantiche. Le informazioni ottenute tramite le Azioni Semantiche sono mantenute mediante attributi della grammatica.

Le piú comuni rappresentazioni intermedie sono [1, 3]:

- Syntax Tree

- 3-Address Code.

Un Syntax Tree ha la stessa forma di un Parse Tree, ma gli operatori prendono il posto dei simboli Non Terminali nei nodi interni dell'albero. Quindi, mentre nel Parse Tree si enfatizza la struttura grammaticale del costrutto, nel Syntax Tree si enfatizza solo il calcolo che deve essere eseguito.

Invece, il 3-Address Code, é una rappresentazione che spezza il programma in comandi elementari che hanno non piú di 3 operandi, e non piú di un operatore. Praticamente, la forma generale di un comando 3-Address Code é x:=y op z, dove [3]:

- x, y e z sono nomi, costanti o variabili temporanee generate dal compilatore, e dobbiamo immaginarli come puntatori, a posizioni di una Symbol Table che contengono i rispettivi valori

- op é un generico operatore.

Il 3-Address Code é un compromesso: ha la forma generale di un Linguaggio ad alto livello, ma i suoi singoli comandi sono abbastanza semplici da essere vicini a quelli di un Linguaggio Assembly. Non é permesso alcun accumulo di espressioni aritmetiche, quindi, se un programma sorgente contiene un'istruzione della forma x+y*z, dovrá essere tradotta nella sequenza: $t_1 := y * z; t_2 := x + t_1$, dove t_1 e t_2 sono variabili temporanee generate dal compilatore.

Questa operazione di smembramento di complicate espressioni o di istruzioni, in 3-Address Code, é molto utile per le fasi successive di un compilatore che deve effettuare delle ottimizzazioni e generare del codice oggetto.

Definiamo ora i costrutti tipo di un 3-Address Code [1]. Abbiamo detto che le istruzioni di un 3-Address Code sono simili ad un Linguaggio Assembly. Le istruzioni possono avere etichette simboliche ed esistono istruzioni per il controllo del flusso. Se il codice intermedio generato viene inserito all'interno di un array, allora un'etichetta rappresenta la posizione, all'interno di questo array, in cui si trova l'istruzione etichettata. Elenchiamo i possibili costrutti del 3-Address Code:

- $x := y \quad op \quad z \qquad con \quad op = \{+, *, -, /, AND, OR\}$
- $x := \quad op \quad y \qquad con \quad op = \{uminus, NOT\}$
- $x := y$
- $goto \quad L$
- $if \quad x \quad relop \quad y \quad goto \quad L \qquad con \quad relop = \{<, >, <>, =, <=, >=\}$
- $paramx$, $call\ p, n$ e $returny$ per il meccanismo delle procedure. L'intero n indica il numero di parametri attuali della procedura invocata, che precedono l'invocazione della funzione.
- $x := y[i]$ e $x[i] := y$ per la realizzazione del meccanismo degli array
- $x := \&y$, $x := *y$, $*x := y$ per le operazioni con i puntatori
- *nop* che é un comando che non modifica lo stato; si usa a volte per lasciare in memoria spazi vuoti
- *halt* che é un comando che termina l'esecuzione.

La scelta delle istruzioni permesse é una importante caratteristica nella progettazione di una forma intermedia di codice. L'insieme delle istruzioni deve essere abbastanza ricco da permettere la traduzione delle istruzioni del codice sorgente, peró bisogna ricordarsi che un insieme piccolo di istruzioni renderebbe piú facile la successiva traduzione in codice oggetto.

Un'istruzione 3-Address Code é una forma astratta di codice intermedio. In un compilatore, questi comandi, possono essere implementati con dei record contenenti gli operandi e l'operatore. Tra le diverse possibili implementazioni, ricordiamo quella a Quadruple.

Una Quadrupla é un record con quattro campi, che chiamiamo op, arg1, arg2, result. Il campo op contiene un codice interno per l'operatore. A seconda dell'istruzione considerata, i restanti campi saranno riempiti come segue [1]:

- per $x := y \quad op \quad z$ si pone y in arg1, z in arg2, x in result

- per $x :=\ op\ y$ non si usa il campo arg2
- per $x := y$ non si usa arg2
- per *goto L* si inserisce l'etichetta L in result
- per $if\ x\ relop\ y\ goto\ L$ si inserisce l'etichetta in result, x in arg1 e y in arg2
- per *param x*, non si usa né arg2, né result.

Il contenuto dei campi arg1, arg2 e result sono normalmente dei puntatori a posizioni di una Symbol Table, che contengono i rispettivi valori.

2.6 Generazione per effetti laterali

La Generazione di Codice per effetti laterali prevede la memorizzazione del codice intermedio, per esempio, in un file globale. In questo tipo di traduzione bisogna sempre ricordarsi che attraversare una struttura vuol dire ritrovarsi con il file globale modificato con il rispettivo codice.

Le espressioni ed i comandi vengono tradotti in modo diverso [1]:

- La traduzione di un'espressione genera il codice, che viene memorizzato nel file globale, ed un attributo .loc che a run-time conterrá il valore dell'espressione

- La traduzione di un comando genera il codice, anch'esso memorizzato nello stesso file, ed un attributo .next detto attributo degli incompleti. L'attributo .next é una lista di posizioni contenenti istruzioni con lo stesso incompleto.

Una funzione caratteristica della Generazione di Codice per effetti laterali, é la emit [1]. La emit é una funzione del metalinguaggio delle azioni semantiche, che prende una stringa rappresentante un'istruzione del 3-Address Code e va ad inserirla nella prima posizione libera del file globale.

La variabile globale *quad* contiene l'indice della prima posizione libera del file globale. All'inizio quad vale zero e dopo ogni emit viene incrementata.

Memorizzare il codice intermedio in un file globale consente la traduzione di comandi che permettono il controllo del flusso, ed in particolare, mediante la tecnica del backpatching, é anche possibile generare del codice incompleto, da completare appena possibile, riuscendo comunque ad effettuare traduzioni ad una passata. Un caso tipico in cui é richiesta la generazione di codice incompleto, é quello di un salto ad una posizione di codice che deve essere ancora generata. In questo caso viene memorizzata nel file globale un'istruzione 3-Address Code di salto, che nella componente result non contiene l'etichetta. L'attributo sintetizzato .next che deriva da questo passo di traduzione, dovrá successivamente essere usato per eseguire il "backpatch" appena si sará arrivati nella posizione in cui si doveva saltare.

Il backpatch é eseguito mediante un'operazione che deve essere messa a disposizione dal metalinguaggio delle azioni semantiche. Questo stesso metalinguaggio deve offrire l'operazione newtemp, che restituisce una locazione temporanea del linguaggio intermedio. La newtemp viene usata di frequente, perché lo smembramento di espressioni complesse in sequenze di operazioni elementari richiede l'uso di variabili temporanee del linguaggio intermedio.

Chapter 3

Scelte di progetto

3.1 Cosa é stato mantenuto e perché

LLParser é stabile, facilmente utilizzabile e con buone prestazioni; queste qualitá sono il frutto di decisioni progettuali ben indovinate [2] che durante la realizzazione di LLParser 2 si sono lasciate ovviamente inalterate.

Durante la progettazione di LLParser fu deciso che il sistema, per essere piú facilmente utilizzabile dagli studenti, doveva essere accessibile via web. Questo tipo di scelta implicó in modo naturale l'uso di una architetura client-server che anche per LLParser 2 risulta essere la piú conveniente.

Utilizzare il Perl come linguaggio di implementazione lato server é stata sicuramente una scelta perfetta sia per le caratteristiche del Perl nella gestione delle stringhe, sia per le buone prestazioni ottenibili utilizzando Perl in collaborazione con il server web Apache che contiene al suo interno un proprio interprete Perl; questa caratteristica consente al server web di sostenere numerosi accessi CGI contemporanei mantenendo prestazioni accettabili [2, 17, 14].

Anche le funzionalitá dell'interfaccia utente sono rimaste sostanzialmente identiche, a parte una nuova veste grafica [12, 11] che é stata resa necessaria dalle nuove esigenze di input. Infatti, dato che con LLParser 2 é possibile utilizzare simboli grammaticali piú espressivi, é stato necessario ampliare sia l'area di input della frase da analizzare, sia l'area di inserimento delle

produzioni. Lo spazio utilizzato é stato recuperato sostituendo i link degli esempi con una lista a scomparsa ed una nuova area di testo che spiega in cosa consiste l'esercizio selezionato. Per terminare si é pensato di inserire un pulsante di help che offra un minimo supporto ai nuovi utenti.

3.2 Come si procede nelle estensioni

Introduciamo le scelte progettuali relative alle diverse estensioni che hanno permesso la realizzazione di LLParser 2.

3.2.1 Rimozione dei vincoli sull'uso dei simboli grammaticali

LLParser permetteva l'uso di Non Terminali rappresentati da una singola lettera maiuscola e di Terminali rappresentati da una singola lettera minuscola. Di conseguenza, le produzioni della grammatica e le frasi riconoscibili erano sequenze continue di token di lunghezza uno[2], come di fatto capitava quasi sempre negli esercizi del corso di Compilatori.

Per ottenere una maggiore espressivitá dello strumento, si é pensato di consentire l'uso, in LLParser 2, di Terminali e Non Terminali piú significativi, separati da uno o piú spazi.

Le Espressioni Regolari che definiscono i nuovi oggetti grammaticali sono:

Non Terminali: $[A-Z][A-Za-z]*$

Terminali: $[a-z][A-Za-z0-9]*|[+-*/():;\&?!\ ,[]''=^{@}><]$

e sono state opportunamente sostituite a quelle vecchie.

Questa modifica, peró, ha imposto la conseguente revisione dell'intero codice di LLParser che, pur rimasto inalterato nella specifica degli algoritmi principali, ha dovuto subire modifiche su strutture dati e implementazione di molti algoritmi.

Per dare un'idea del tipo di interventi effettuati, presentiamo uno schema che consenta un confronto tra LLParser e LLParser 2:

LLParser	LLParser 2
- Stringa contenente una sequenza continua di caratteri	- Stringa di token separati da uno spazio
- Split sui caratteri	- Split sugli spazi che separono i token
- Match con la vecchia espressione regolare dei Terminali	- Match con la nuova espressione regolare dei Terminali
- Match con la vecchia espressione regolare dei Non Terminali	- Match con la nuova espressione regolare dei Non Terminali
- Riconoscimento token	- Uso di ancore nelle espressioni regolari dei match per consentire un corretto riconoscimento dei token
- La tabella del parser é un hash	- La tabella del parser é un hash di hash
- Lo stack é una stringa	- Lo stack é un array Perl ad allocazione dinamica gestito come stack
- L'input é una stringa	- L'input é un array Perl ad allocazione dinamica gestito come coda
- Su stack e input il controllo del vuoto si fa con la stringa nulla	- Controllo del vuoto con undef
- La chiave della tabella del parser é una stringa di due caratteri	- La chiave é data da due token in due stringhe
- Uso di llparser.dat per l'output	- Uso di llparser1.dat per una nuova formattazione dell'output

Inoltre, per comoditá, sono state introdotte alcune nuove subroutine e in-

seriti nuovi messaggi di errore.

3.2.2 Meccanismo di generazione del codice

Diverse nuove funzionalitá offerte da LLParser 2 scaturiscono dalle estensioni che permettono l'uso di meccanismi di Generazione di codice per effetti laterali ad una passata. Per consentirne l'uso sono state messe a disposizione dell'utente le seguenti nuove funzioni:

newtemp: é una funzione che restituisce una sequenza di nomi distinti, $£1$, $£2$, ..., in risposta a successive chiamate, rappresentanti locazioni di memoria del 3-Address Code. Si puó solo assegnare ad una variabile d'ambiente all'interno di un'azione semantica.
Un esempio d'uso: $\{\#prova = newtemp\}$;

quad: restituisce l'attuale valore di quad ovvero la prima posizione libera dell'array contenente il 3-Address Code. É liberamente utilizzabile nelle espressioni delle azioni semantiche.
Esempi d'uso:
$\{\#temp = 1 + quad; \#prova = quad; if A.attr = quad \ldots\}$;

[]: permette l'inizializzazione di una lista vuota, utilizzabile all'interno del meccanismo del backpatch;

[quad]: si usa negli assegnamenti delle azioni semantiche per inizializzare una lista contenente l'attuale valore di quad. É importante per un corretto uso del meccanismo di backpatch.
Un esempio d'uso: $\{\#temp = [quad]\}$;

push(#lista,posizione): inserisce *quad* o un numero intero, rappresentante una posizione dell'array contenente il codice intermedio, in *#lista*. Anche in questo caso si é trattata l'eventuale immissione errata di argomenti.
Esempi d'uso: $\{push(\#lista, quad); push(\#lista, 12)\}$;

BK(#lista,posizione): inserisce il parametro *posizione* nella quarta componente del 3-Address Code presente nell'array del codice intermedio nelle posizioni contenute in *#lista*. Come scelta di progetto si é cercato di trattare l'eventuale immissione di argomenti scorretti, restituendo messaggi di errore che aiutino nella comprensione della situazione. In particolare, il primo argomento deve essere una variabile d'ambiente mentre il secondo deve essere *quad* o un valore intero. Si usa nelle azioni semantiche in uno dei seguenti modi:
$\{BK(\#lista, quad); BK(\#lista, 12)\}$;

emit(istruzione): consente l'inserimento di istruzioni nell'array contenente il codice intermedio. L'*istruzione* viene scelta tra quelle offerte dal 3-Address Code messo a disposizione dell'utente. Mostriamo alcuni esempi di come puó essere usata nelle azioni semantiche:
$\{emit(\#prova\ '\!:=\!'\ 1\ '+'\ A.in); emit('goto'\#primo); emit('goto'\_);$
$emit('goto'7); emit('if'A.in\ '<'\ \#true\ 'goto'\_)\}$.
Il riconoscimento della versione sintattica della emit manda in esecuzione diverse funzioni che insieme realizzano la versione semantica. Queste funzioni si possono dividere in due insiemi. Il primo é composto da:

- PrepareEmit
- TransformQuadrupla
- VarAssegn
- Argomento
- splitvar
- GestErroriVar

che hanno il compito di riconoscere l'istruzione e di preparare la quadrupla verificando la correttezza sintattica di istruzione e argomenti.
Il secondo insieme é composto da:

- RunEmit
- EsisteArg
- EsisteRes
- Emit

che hanno il compito, al momento della derivazione delle produzioni, di eseguire la emit inserendo la quadrupla nell'array del codice intermedio, ma solo dopo aver superato il controllo di esistenza degli argomenti dell'istruzione sui vari stack di sistema. É anche stato inserito alla fine un controllo che avvisa quando a qualche salto non é stato eseguito il backpatch.

Parlando della emit abbiamo accennato al 3-Address Code. Il Linguaggio Intermedio a 3 operandi, attualmente utilizzato, mette a disposizione i seguenti costrutti:

- $x := y \quad op \quad z \qquad con \quad op = \{+, *, -, /, AND, OR\}$
- $x := \quad op \quad y \qquad con \quad op = \{uminus, NOT\}$
- $x := y$
- $goto \quad L$
- $if \quad x \quad relop \quad y \quad goto \quad L \qquad con \quad relop = \{<, >, <>, =, <=, >=\}$

La notazione per consentirne l'uso dentro la emit prevede l'inserimento di apici che racchiudono le keyword e gli operatori del 3-Address Code, a differenza di variabili d'ambiente ed attributi che non li richiedono.
La rappresentazione interna scelta é quella a quadruple giá discussa nel Capitolo 2.

3.2.3 Gestione dei lessemi dei token ide e num

Sia LLParser che LLParser 2 non hanno un analizzatore lessicale perché scopo dello strumento era quello di concentrare l'attenzione degli utenti sulla

fase di parsing. Comunque con LLParser 2 si é pensato di permettere, agli utenti interessati, di inserire frasi del linguaggio che al posto dei token *ide* e *num* contengano dei lessemi adeguati. Questa nostra scelta impone all'utente, quando necessario, di utilizzare il token *ide* per la categoria lessicale degli identificatori e di usare il token *num* come categoria lessicale dei numeri interi.

Abbiamo trattato il problema in due modi diversi, secondo che l'utente desideri usare o meno azioni semantiche.

I caso: Grammatiche non attributate

É il caso piú semplice ed é del tutto trasparente all'utente. Senza obbligare l'utente a dover inserire alcun tipo di dichiarazione di categorie lessicali o di routine che implementino il riconoscimento lessicale, LLParser 2 é in grado di capire se in input é presente o meno un lessema appartenente ad *ide* o *num*, e di comportarsi di conseguenza.

Non essendo presente in LLParser 2 alcun meccanismo capace di trattare informazioni relative a lessemi e non essendo richiesta l'esecuzione di azioni semantiche, si é pensato di poter fare a meno, in questo particolare caso, di una Symbol Table. É scaturito cosí un meccanismo elementare capace di trattare il problema senza assolutamente appesantire il funzionamento del sistema.

Prima di tutto é stato sufficiente inizializzare, durante la fase di analisi delle produzioni, una struttura con tutte le keyword incontrate. Successivamente, durante la fase di derivazione, quando in input LLParser 2 troverá un lessema che non é keyword, dovrá solo controllare se appartiene alla categoria lessicale *ide* o *num*; se la risposta é positiva, sostituirá il lessema con il rispettivo token e continuerá con la derivazione, altrimenti interromperá l'esecuzione con un messaggio di errore.

II caso: Grammatiche attributate

É il caso piú interessante e sicuramente piú utile. In questo caso é necessario tenere traccia dei lessemi in input, associandoli ai rispettivi token, e per fare ció si deve utilizzare una Symbol Table.

Invece di implementare un meccanismo automatico interno a LLParser 2, che avrebbe influito negativamente sulla generalitá delle grammatiche definibili, si é pensato di mettere a disposizione dell'utente le seguenti quattro nuove funzioni, da poter usare all'interno delle azioni semantiche, che consentano la gestione della Symbol Table [4]:

insert(nome,token,tipo): inserisce le informazioni del lessema nella Symbol Table del blocco attivo

lookup(nome): cerca nella Symbol Table il lessema passato per parametro

setblock: da invocare quando si vuole entrare in un blocco

resetblock: da invocare quando si vuole uscire da un blocco.

Questo insieme di funzioni permette la definizione di grammatiche per linguaggi molto eterogenei; infatti, a seconda di come si usano, é possibile definire grammatiche per linguaggi

- con dichiarazioni implicite o esplicite
- con o senza strutture a blocchi
- con o senza funzioni (per il momento non é possibile dichiarare funzioni ricorsive).

A questo punto, all'interno delle azioni semantiche, é possibile fare riferimento al lessema del token *ide* con l'attributo *ide.name*, mentre a quello di *num* con *num.value*. Come per i Non Terminali, c'é la convenzione che, se in una produzione ci sono piú *ide* o piú *num*, i loro attributi si differenziano con *ide1.name, ide2.name, ...* o *num1.value, num2.value, ...*

La Symbol Table creata e gestita dall'utente sfrutta in parte i meccanismi interni offerti dal Perl stesso, che, in automatico, associa ad ogni package una separata Symbol Table, raggiungibile partendo dalla Symbol Table principale del linguaggio Perl [7].

Per tenere traccia, durante l'analisi, della sequenza di incapsulamento dei blocchi, si usa uno stack di identificatori di blocchi che in ogni istante ha sul top il blocco attivo.

Ovviamente anche in questo secondo caso, durante la fase di analisi delle produzioni é necessario inizializzare una struttura con tutte le keyword incontrate. Invece durante la fase di derivazione, quando LLParser 2 troverá in input un lessema che non é keyword, dovrá controllare se appartiene o meno ad una delle due categorie lessicali ammesse e comportarsi di conseguenza; in particolar modo quando eseguirá la trasformazione di un lessema nel suo token avviserá con il nuovo messaggio di output:[transforming lexema in token].

Chapter 4

Estensioni per l'analisi statica

In questo capitolo descriviamo le modifiche, apportate a LLParser, che consentiranno l'uso in LLParser 2 di Terminali e Non Terminali piú espressivi.

Dato che questo lavoro di tesi consiste nel potenziamento di LLParser, e considerando che sono previsti ulteriori nuovi sviluppi, si é pensato di strutturare questo capitolo come il quarto capitolo della tesi del Dott. Samuele Manfrin [2], che per comoditá alleghiamo nell'*appendice A*.

Questa scelta ha i seguenti due pregi: come prima cosa, permetterá di soffermarci esclusivamente sulle nostre scelte implementative, rimandando ai corrispondenti paragrafi per aspetti giá discussi nella precedente tesi. Come seconda cosa, consentirá a successivi tesisti di avere un quadro della situazione piú chiaro, sia sull'architettura del sistema originario, sia sulle modifiche che si sono apportate per consentire questa particolare estensione.

Per una visione d'insieme dei tipi di interventi effettuati, é possibile consultare la tabella presentata nel *paragrafo 3.2.1 "Rimozione dei vincoli sull'uso dei simboli grammaticali"*.

Per evitare di perdere il controllo della situazione, durante questa fase di modifiche si é pensato di utilizzare un sistema di controllo delle versioni. In particolare si é utilizzato l'*RCS (Revision Control System)* di Linux, che si é rivelato utilissimo in molti casi [13].

4.1 Struttura generale del programma

Il codice di LLParser 2 si puó considerare come composto da tre grandi parti:

- Acquisizione dell'input ed inizializzazione delle strutture dati
- Generazione della tabella di parsing
- Riconoscimento della stringa in input, con eventuale esecuzione delle azioni semantiche e generazione di codice intermedio.

Gli interventi che si sono dovuti effettuare non hanno risparmiato quasi nessuna subroutine, ma abbiamo cercato di mantenere inalterate le specifiche degli algoritmi esistenti.

4.2 Strutture dati

Facciamo un confronto tra le strutture dati di LLParser e LLParser 2, soffermando l'attenzione sulle estensioni trattate in questo capitolo. Per riutilizzare il piú possibile il codice preesistente, si é cercato di non modificare troppo le strutture esistenti. Le differenze che si riscontrano possono essere di due tipi diversi:

- Differenze sui valori contenuti
- Differenze sulla struttura.

Le strutture dati necessarie al funzionamento del parser sono:

- La tabella di parsing
- Lo stack
- La coda di input
- Un insieme di stack per gli attributi e le variabili d'ambiente
- Una struttura per la memorizzazione della grammatica

- Una struttura per i first
- Una struttura per i follow.

In LLParser la grammatica senza azioni veniva memorizzata nell'hash *%g*. Le chiavi corrispondevano ai *Non Terminali*, mentre il valore contenuto era un reference ad un array contenente le parti destre di tutte le produzioni di quel *Non Terminale*. Per chiarire la situazione, consideriamo il seguente esempio:

```
S::=aS|B          $g{‘‘S’’} = [‘‘aS’’,‘‘B’’]
B::=c|@           $g{‘‘B’’} = [‘‘c’’,‘‘@’’]
```

Inoltre, per comoditá, ogni produzione veniva memorizzata nell'array @prod.

In LLParser 2 si sono mantenute le stesse strutture, che peró contengono valori diversi che dipendono dalle nuove possibilitá di input. Infatti, una possibile situazione potrebbe essere:

```
S ::= ide S | Next       $g{‘‘S’’}    = [‘‘ide S’’,‘‘Next’’]
Next ::= num | @         $g{‘‘Next’’} = [‘‘num’’,‘‘@’’]

@prod = [‘‘S::=ide S | Next’’,‘‘Next::=num | @’’]
```

In LLParser la grammatica attributata era memorizzata negli array referenziati dall'hash %ga. Questo significa che @$ga{“A”} referenziava un array che conteneva i reference all'array di tutte le parti destre delle produzioni di “A” divise per token. Consideriamo il seguente esempio:

```
A::=aA{A.tot=1+A1.tot}|@{A.tot=0}

${$ga{‘‘A’’}}[0] = [‘‘a’’,‘‘A’’,‘‘{A.tot=1+A1.tot}’’]
${$ga{‘‘A’’}}[1] = [‘‘@’’,‘‘{A.tot=0}’’]
```

In LLParser 2 ció che cambia é ancora il contenuto che puó assumere *%ga*. Diamo un esempio confrontabile con quello precedente:

```
Next::= ide Next {Next.tot=1+Next1.tot} | @ {Next.tot=0}


${$ga{‘‘Next’’}}[0] = [‘‘ide’’,‘‘Next’’,
                                ‘‘{Next.tot=1+Next1.tot}’’]
${$ga{‘‘Next’’}}[1] = [‘‘@’’,‘‘{A.tot=0}’’]
```

In LLParser, i first venivano memorizzati come stringhe nell'array associativi %first, con chiave tutte le parti destre delle produzioni. Invece i follow venivano memorizzati come stringhe nell'array associativo %fw, con chiave tutti i Non Terminali. Vediamo un esempio:

```
S::=aS|B  %first = (‘‘aS’’=>‘‘a’’,‘‘B’’=>‘‘c@’’,
                    ‘‘c’’=>‘‘c’’,‘‘@’’=>‘‘@’’)
B::=c|@   %fw = (’S’=>’$’,’B’=>’$’)
```

In LLParser 2 cambia ancora una volta il contenuto di queste strutture; dovremo lavorare con stringhe contenenti token separati da spazi, come si puó vedere nell'esempio:

```
S::=ide S | Next  %first = (‘‘ide S’’=>‘‘ide’’,
                            ‘‘Next’’=>‘‘num @’’,
                            ‘‘num’’=>‘‘num’’,‘‘@’’=>‘‘@’’)
Next::=num | @   %fw = (’S’=>’$’,’Next’=>’$’)
```

In LLParser, la tabella del parser veniva memorizzata in due diversi modi. Nell'hash %table c'era la tabella senza azioni, mentre nell'hash %tablea c'era la tabella con le azioni. In entrambi gli hash la chiave era rappresentata dalla stringa ottenuta giustapponendo il Non Terminale sullo stack ed il terminale in input. Nel primo hash il contenuto consisteva nella produzione associata alla chiave; nel secondo era un reference all'elemento corrispondente dell'array %ga.

In LLParser 2, in questo caso, si é pensato di cambiare le strutture. Si é voluto evitare la costruzione di chiavi composte dalla giustapposizione di due stringhe di diversa lunghezza. Si é preferito trasformare sia %table che

%tablea in hash di hash. Per poter accedere a queste nuove strutture, sono ora necessarie due chiavi distinte; il Non Terminale sullo stack ed il Terminale in input. Nell'hash %table il contenuto consiste nella produzione associata a questa coppia di chiavi, nella sua nuova forma. Invece in %tablea si trova un reference all'elemento corrispondente dell'array %ga, anche questo nella sua nuova forma vista poco fa.

Infine, LLParser manteneva le informazioni relative ad attributi e variabili d'ambiente negli stack @ere, @sin, @corr ed @amb. Lo stesso accade in LLParser 2, peró, sempre considerando il fatto che le nuove regole di input consentono di memorizzare, su questi stack, attributi dai nomi piú significativi.

4.3 Dettaglio implementativo

Diamo uno schema riassuntivo dei diversi passi che LLParser 2 esegue, elencando le principali subroutine che entrano in gioco, specificando volta per volta se si sono dovute apportare delle modifiche.

- Acquisizione dei dati
  - &read_parse (nessuna modifica)
  - &TakeData e le sue subroutine (sono state modificate)
- Memorizzazione dell'input nel log di sistema
  - &QueryLog (nessuna modifica)
- Valutazione dei First
  - &ValutaFirst e le sue subroutine (sono state modificate)
- Valutazione dei Follow
  - &ValutaFW e le sue subroutine (sono state modificate)

- Verifica del tipo di grammatica
  - &CheckLL1 ed una sua subroutine (sono state modificate)
- Generazione della tabella di parsing
  - &MKTable (é stata modificata)
- Analisi LL(1) della stringa in ingresso, con eventuale esecuzione di azioni semantiche e generazione di codice
  - &AnalisiLL1 e le sue subroutine (sono state modificate)
  - &AnalisiLL1a e le sue subroutine (sono state modificate)
- Output finale
  - &OutPage (é stata modificata)

Adesso descriveremo piú dettagliatamente il lavoro svolto.

4.3.1 Acquisizione dei dati e preparazione della grammatica

Come giá anticipato precedentemente, la *&read_parse* non ha dovuto subire modifiche.

La *&TakeData* é stata cambiata per gestire la maggiore libertá di input concessa agli utenti, ed usa le seguenti subroutine:

- *&CleanString*
- *&Decomponi*
- *&Analizza*
- *&OutErrPage.*

La *&TakeData*, per riconoscere la struttura delle produzioni, usa la seguente espressione regolare [10, 6] che tiene conto sia della nuova espressivitá dei simboli grammaticali, sia della possibilitá di formattare l'input con un qualsiasi numero di spazi:

```
/([A-Z][A-Za-z]*[\s]*)::=(.+?)
                    (?=([A-Z][A-Za-z]*[\s]*::=|\$))/sg
```

In LLParser era possibile inserire grammatiche della forma:

```
S::=TE
E::=+TE|@
T::=FB
B::=*FB|@
F::=(S)|i
```

Invece, grazie alla nuova espressione regolare, la grammatica appena vista puó essere inserita in LLParser 2 in svariati modi, tutti ugualmente riconosciuti. Questo primo esempio ha una formattazione ordinata:

```
S::=Term Expr
Expr::=+ Term Expr |@
Term::=Fatt Termine
Termine::=* Fatt Termine | @
Fatt::=( S ) | id
```

Con questo secondo esempio, che ha una formattazione volutamente disordinata, si desidera mostrare la grande libertá di input che LLParser 2 offre:

```
 S  ::=  Term


   Expr
Expr ::=+ Term    Expr
 |   @
 Term::=    Fatt    Termine
```

```
 Termine   ::= *   Fatt   Termine

 | @
Fatt::=(
 S
) |
 id
```

&TakeData, prima di memorizzare le singole produzioni, le ripulisce dagli spazi di troppo utilizzando *&CleanString*, una nuova funzione impiegata in molte parti del codice. La *&CleanString* prende in input una stringa e, mediante l'uso di pattern matching [6], elimina gli spazi iniziali, accorpa gli spazi intermedi in un unico spazio ed elimina gli spazi finali. Alla fine restituisce la stringa cosí trattata.

Alla *&Decomponi* sono state cambiate le espressioni regolari dei suoi due pattern match. Con la prima modifica si é ottenuto l'espressione regolare

```
m/(([a-zA-Z0-9]+|[\+\-\*\/\(\)\:\;\&\?\!\~\,\[\]
                          \'\"\=\^\@\<\>])|(\{.+?\}))/g
```

che permette il riconoscimento di azioni semantiche o di *token* anche piú lunghi di un carattere. Invece, la seconda espressione regolare é divenuta

```
/^([a-z][A-Za-z0-9]*|[\+\-\*\/\(\)\:\;\&\?\!\~\,\[\]
                                       \'\"\=\^\@\<\>])$/
```

e permette il riconoscimento di tutti i *Terminali* presenti nelle produzioni. Anche la costruzione della stringa *$clean* é cambiata; i token riconosciuti, che non siano un'azione semantica, si accodano a *$clean*, facendo attenzione a separarli con uno spazio.

La *&Decomponi* invoca la *&Espansione*, che é una funzione che ha solo il compito di invocare la *&modificatore* quando in input riceve un'azione semantica. In *&Espansione* abbiamo cambiato, per comoditá, l'espressione regolare che riconosce l'azione semantica.

In *&modificatore*, durante questo primo tipo di estensione, é cambiata solo l'espressione regolare per il riconoscimento dei nuovi Non Terminali. Le altre sostanziali modifiche che sono state necessarie per realizzare le altre estensioni, saranno discusse nei prossimi capitoli. La *&modificatore* usa le seguenti routine:

- *&strsplit*
- *&CleanString*
- *&splitvar*
- *&GestErroriVar*
- *&splitassign*
- *&OutErrPage*
- *&evalguardia*
- *&PrepareEmit*.

In *&strsplit* abbiamo dovuto apportare dei cambiamenti per evitare dei comportamenti indesiderati di LLParser 2, che si sarebbero potuti verificare nel caso in cui l'utente avesse scelto dei nomi infelici per attributi o variabili d'ambiente. Se, per esempio, durante l'uso del backpatch, l'utente avesse dichiarato una variabile d'ambiente *#endif*, per indicare l'intenzione di volere effettuare un salto alla fine di un *if then else*, il sistema avrebbe trasformato la variabile in *# end*, con un conseguente errore a run-time. Per ovviare a queste, sebbene difficili, eventualitá, abbiamo utilizzato le "ancore" $\backslash b$ [6] all'interno dei pattern match presenti in *&strsplit*. In questo modo le prime sostituzioni, necessarie per le "macroespansioni" delle azioni semantiche, risultano corrette. Piú precisamente, con le nuove sostituzioni:

```
....
$s =~ s/[\s;]then\b/;{;/g;
```

```
$s =~ s/[\s;]endif\b/ end/g;
....
$s =~ s/[\s;]endw\b/end/g;
$s =~ s/[\s;]begin\b/;{;/g;
$s =~ s/[\s;]end\b/;};/g;
....
$s =~ s/[\s;]else\b/;}; else; {;/g;
$s =~ s/#true\b/1/g;
$s =~ s/#false\b/0/g;
```

si riesce ad evitare la trasformazione indesiderata di, per esempio, *tenda* in *t;};a*, di *#endif* in *# end* o di qualunque altra strana coincidenza.

In *&splitassign* abbiamo dovuto modificare, nel modo seguente, il pattern match che riconosce tutte le variabili attributate presenti nell'espressione da analizzare

```
$d =~ /([A-Z][A-Za-z]*[0-9]*\.[a-zA-Z0-9]+)/g
```

cosí, dalle vecchie A1.in, B.next, B2.prova, si puó passare alle ben piú espressive Expr1.in, Statement.next, Statement2.prova, ...

Inoltre, sempre in *splitassign*, é stato specializzato un messaggio di errore e, all'occorrenza, si é fatto uso della nuova funzione *&CleanString* per preparare al meglio i dati da trattare. Le altre sostanziali modifiche che sono state necessarie per realizzare le altre estensioni, saranno discusse nei prossimi capitoli.

La *&splitvar* é invocata da diverse altre funzioni. Anche qui abbiamo dovuto modificare l'espressione regolare che definisce i Non Terminali. Infatti, per permettere un corretto riconoscimento delle nuove variabili attributate, si deve utilizzare il pattern match

```
$err=(($var =~ m/^([A-Z][a-zA-Z]*)[0-9]*$/) &&
                  ($attrib =~ m/^[a-zA-Z0-9]+$/))?0:1;
```

Ecco alcuni esempi di output della *&splitvar*:

```
con ‘‘Stm2.next’’ si ha (‘‘Stm2’’,‘‘next’’,‘‘Stm’’,0)
con ‘‘#temp’’ si ha (‘‘#’’,‘‘’’,‘‘temp’’,0)
con ‘‘#TEMP si ha (‘‘#’’,‘‘’’,‘‘TEMP’’,2)
                    che evidenzia il codice di errore 2
con ‘‘stm2.next’’ si ha (‘‘Stm2’’,‘‘next’’,‘‘Stm’’,1)
                    che evidenzia il codice di errore 1
con ‘‘dfH’’ si ha (‘‘dfh’’,‘‘’’,‘‘’’,4)
                    che evidenzia il codice di errore 4
```

Come si vede nell'esempio precedente, nella *&splitvar*, per consentire una gestione un po' piú accurata degli errori, é stato introdotto un ulteriore quarto caso di riconoscimento. In questo caso viene restituito il codice di errore *4*, che evidenzia l'uso di un identificatore scorretto. Altre modifiche apportate alla funzione saranno descritte nei capitoli successivi.

La funzione *&evalguardia* non ha subito modifiche. Fa uso della *&sostvar*, che, avendo subito modifiche per un altro tipo di estensione, sará trattata nel capitolo 6.

La chiamata alla vecchia funzione *&evalemit*, che era presente in *&modificatore*, é stata sostituita da un insieme di nuove funzioni che saranno oggetto del capitolo 6.

4.3.2 La preparazione della tabella di parsing

Dovendo lavorare con oggetti della grammatica profondamente cambiati, la parte di codice che prepara la tabella del parser ha richiesto delle modifiche sia al codice che alle strutture dati.

Il calcolo di First

Questo calcolo viene effettuato dalla subroutine *&ValutaFirst*, con l'ausilio delle funzioni:

- *&ValutaFirstToken*

- *&PulisciInsieme.*

In *&ValutaFirst* é cambiato l'aggiornamento dell'insieme dei Terminali che appartengono a qualche First. Infatti, ora, vengono accodati nella stringa *$xt* tutti i Terminali trattati, separandoli con uno spazio. Questa scelta ha permesso di non modificare eccessivamente gli algoritmi di questa parte. Quindi, mentre in LLParser un possibile valore della *$xt* poteva essere

```
‘‘a)dlu-’’
```

ora, in LLParser 2, un possibile valore di questa variabile potrebbe essere

```
‘‘while + ide ) if num’’
```

Si é scelto di mantenere l'uso di una stringa, pur avendo dei dati un po' piú sostanziosi, perché Perl é un linguaggio che ha ottime prestazioni nella gestione di stringhe, anche nel caso di stringhe di notevoli dimensioni. A proposito di questa caratteristica, citiamo una frase estratta da [6]:

> The shortest possible string has no characters. The longest string fills all of your available memory. This is in accordance with the principle of "no built-in limits" that Perl follows at every opportunity.

In *&ValutaFirstToken* si é portata ovviamente avanti la scelta di lavorare, quando possibile, con le stringhe. La prima modifica necessaria é stata quella di cambiare il modo di utilizzo della split di Perl. Prima, infatti, era necessario eseguire lo split [6] su stringhe del tipo

```
‘‘aBcDdd’’
```

mentre ora si deve effettuare su stringhe del tipo

```
‘‘if Expr then Stm else Stm’’
```

Per questo motivo, l'istruzione che usa la split é diventata

```
foreach my $c (split(/\s+/,$t))
```

la quale usa, come caratteri separatori, gli spazi presenti tra i vari token della stringa passata per parametro, ed é quindi in grado di restituire come risultato un array contenente, ad esempio, i token

```
‘‘if’’, ‘‘Expr’’, ‘‘then’’, ‘‘Stm’’, ‘‘else’’, ‘‘Stm’’
```

invece dei semplici caratteri

```
‘‘a’’, ‘‘B’’, ‘‘c’’, ‘‘D’’, ‘‘d’’, ‘‘d’’
```

In *&ValutaFirstToken*, tra l'altro, é cambiata l'istruzione che contiene il pattern match per il riconoscimento dei simboli *Terminali*

```
 if ($c=~/^([a-z][A-Za-z0-9]*|
                 [\+\-\*\/\(\):;&?!~,\[\]\’\"=\^@><])$/)
               ,  { return($tot." ".$c) }
```

Come si puó notare, questa istruzione, oltre ad avere la nuova espressione regolare, restituisce la variabile *$tot* che non é piú una stringa di caratteri. Infatti, le tre variabili *$tot, $l, $insieme*, da stringhe di caratteri, sono diventate stringhe di token separati da spazi, e devono quindi essere gestite come la *$xt* vista prima.

La *&PulisciInsieme* é stata adeguata alle modifiche che si sono apportate alle funzioni chiamanti. Per questo motivo, anche qui, lo split viene eseguito considerando gli spazi presenti tra i token. Inoltre anche il primo operando della join [6], utilizzata nella costruzione dell'output della funzione, é uno spazio

```
....
foreach my $c (split(/\s+/,$s)) { $h{$c}++ }
   my $result = join(" ", sort keys %h);
   return $result;
....
```

in modo da ottenere una stringa di token distinti.

Il calcolo di Follow

Questo calcolo viene eseguito dalla funzione *&ValutaFW*, che fa uso delle routine:

- *&ValutaFirstToken*
- *&PulisciInsieme*

giá discusse poco sopra.

Anche nella *&ValutaFW* abbiamo dovuto modificare le due split, che ora usano gli spazi come separatori di token. La prima split si trova nell'istruzione

```
my @c=split(/\s+/,$i);
```

che inizializza un vettore estraendo tutti i token che compongono la parte destra della produzione in esame. Invece la seconda split, che si trova nella seconda fase della *&ValutaFW*, é usata nell'istruzione

```
foreach my $m ( split(/\s+/,$mai) ) {....}
```

per dividere la stringa presente in *$mai*, una variabile contenente tutti i Non Terminali presenti in un follow parziale. In *&ValutaFW* sono state anche aggiornate le espressioni regolari che hanno il compito di riconoscere i Terminali ed i Non Terminali. Per assicurare il corretto funzionamento del pattern match, si sono dovute usare anche le ancore $\backslash b$ nel seguente modo:

```
....
if ( $ch =~ /\b([A-Z][A-Za-z]*)\b/) {...
....
(my $mai=$fw{$k})=~s/(\b[a-z][A-Za-z0-9]*\b)|
      [\$+\-\*\/\(\):;&?!~,\[\]\'\"=\^@><]//g;
....
$fw{$k}=~s/\b([A-Z][A-Za-z]*)\b//g;
....
```

Sempre in *&ValutaFW*, le variabili *$partefinale*,*$f*, *$fw*, *$mai*, *$xt*, che prima erano stringhe di caratteri, ora, sono diventate stringhe di token separati da spazi. Questo ha richiesto la modifica della loro gestione e, all'occorrenza, si sono dovute ripulire dagli spazi in eccesso tramite la *&CleanString*. Gli spazi in eccesso si possono presentare in alcuni casi, con operazioni di eliminazione di token dalle stringhe.

Con queste modifiche, che hanno mantenuto l'essenza delle strutture dati usate, siamo riusciti a riutilizzare l'algoritmo giá usato in LLParser.

La verifica del tipo di grammatica

Questa operazione viene svolta dalla funzione *&CheckLL1*, con l'ausilio della &Controlla. Queste due funzioni implementano il controllo delle due proprietá che devono essere soddisfatte dalle grammatiche LL(1) [1]:

- $\forall A ::= \alpha|\beta \ first(\alpha) \cap first(\beta) = \{\}$
- $\forall A ::= \alpha|\beta$ se $\alpha \Rightarrow^* \epsilon$ allora $first(A) \cap first(\beta) = \{\}$.

La *&CheckLL1* e la &Controlla usano i First e i Follow, quindi le modifiche apportate sono della stessa natura di quelle sopra descritte. Infatti, in *&CheckLL1* la variabile *$unione* é diventata una stringa di token separati da uno spazio, mentre in *&Controlla* si esegue uno split che usa come carattere separatore gli spazi tra i token.

4.3.3 L'output: il parser in azione

In LLParser si scelse di eseguire il parsing vero e proprio in concomitanza con l'output della pagina web. Questa scelta fu fatta per evitare problemi in caso di piú accessi concorrenti al server [2, 17]. Con LLParser 2, condividendo le motivazioni, si é mantenuta la stessa scelta.

La funzione che si occupa dell'output é la *&OutPage* che si avvale delle seguenti funzioni:

- *&MKTable*

- *&NuovaAnalisiLL1a* che sostituisce la AnalisiLL1a
- *&NuovaAnalisiLL1* che sostituisce la AnalisiLL1
- *&Emit* che sostituisce la vecchia versione.

In *&OutPage* sono state aggiornate le chiamate alle nuove funzioni. Inoltre fa uso di un nuovo formato generico di stampa, contenuto nel file *llparser1_1.dat*. Questo nuovo formato di stampa é stato reso necessario dalle nuove esigenze di output, derivanti dall'uso di simboli grammaticali piú lunghi.

In *&MKTable* c'é la prima modifica veramente importante alle strutture dati: i due vecchi hash *%table e %tablea* sono diventati due hash di hash. Questo si ripercuote ovviamente sul funzionamento di questa parte di codice, che dovrá accedere a queste tabelle mediante due chiavi, opportunamente calcolate ed usate. Continuano, inoltre, ad esserci le modifiche derivanti dalla scelta fatta a monte, di lavorare con stringhe di token separati da spazi. Infatti gli split di *$ff e $xt* si effettuano sugli spazi che separano i token.

In *&NuovaAnalisiLL1a* abbiamo ulteriori nuove modifiche sulle strutture dati. La stringa che veniva utilizzata in LLParser come input é stata sostituita con un array dinamico di Perl gestito a coda.

Questo significa che le azioni che venivano compiute su questa stringa, sono state sostituite con azioni, semanticamente equivalenti, sulla nuova coda. Quindi, il controllo dello svuotamento degli stack di sistema, importante per la terminazione del lavoro di riconoscimento, é stato eseguito utilizzando il valore speciale undef [6]

```
while (($stack[0] ne undef) || ($in[0] ne undef)) {....
```

Inoltre, le due variabili *$tos* e *$tosin*, che venivano usate per indicare l'inizio delle due strutture, sono state sostituite con *$stack[-1] e $in[0]* [6] . Per evidenziare il fatto che l'input é una coda, si sono usate le istruzioni Perl di pop e shift nella gestione di *@in*.

Inoltre, siccome le tabelle usate ora sono implementate con hash di hash, si é dovuto adeguare il meccanismo di accesso, introducendo l'uso di due chiavi separate: *$keyNT e $keyT*.

In *&NuovaAnalisiLL1a* continuano a presentarsi le solite modifiche viste giá per le altre funzioni. Sono state modificate le espressioni regolari per i Terminali e per i Non Terminali e, in certe variabili, ora sono contenute stringhe di token separati da spazi.

Nella *&NuovaAnalisiLL1*, sia lo stack che l'input erano stati implementati con due stringhe. Si é preferito sostituire la stringa che veniva utilizzata in LLParser come stack, con un array dinamico di Perl. Stessa sorte per la stringa che veniva usata come input, che é stata sostituita da un array dinamico gestito a coda. Di conseguenza, sono state apportate modifiche simili a quelle appena discusse per la *&NuovaAnalisiLL1a*.

Chapter 5

Estensioni per la generazione di codice

In questo capitolo parleremo delle modifiche apportate al sistema per consentire l'implementazione delle estensioni relative alla generazione del codice. Parleremo prima delle strutture dati utilizzate, per poi passare alla descrizione dettagliata delle subroutine. Le nuove funzioni introdotte formano un sottosistema quasi autonomo, nel senso che l'integrazione con il sistema LLParser preesistente é stata realizzata modificando le sole tre subroutine *splitassign*, *sostvar* e *modificatore*.

5.1 Le strutture dati

Abbiamo avuto bisogno di una struttura dati dinamica per l'output delle emit [1]. Dato che gli array Perl sono dinamici [6] e sono gestiti in modo molto efficiente, si é pensato fosse conveniente per le prestazioni del sistema, memorizzare il tutto in un array *@emit*. Considerando inoltre l'ottima gestione Perl delle stringhe, ogni elemento di questo array é una stringa che rappresenta una quadrupla del 3-Address Code.

Per uniformarci alla teoria del corso di Compilatori, l'indice della prima posizione libera di questo array é contenuta nella variabile intera *$quad* e

potrá essere usata da un utente tramite la denotazione *quad* [1].

Un'ulteriore variabile intera che é stata introdotta é *$globalloc*, utilizzata nell'implementazione della newtemp. É un contatore globale che servirá per generare gli interi 0,1,2, ..., che saranno usati nella costruzione di una sequenza di nomi distinti, £1, £2, ... Questi saranno restituiti in risposta a successive chiamate di *newtemp* e rappresenteranno locazioni di memoria del 3-Address Code.

5.2 Il codice

Descriveremo ora la parte di LLParser 2 che tratta della Generazione del Codice. Questa nuova funzionalitá puó essere suddivisa in due fasi:

- Preparazione delle azioni che devono generare codice
- Esecuzione a run-time di queste azioni.

La preparazione delle azioni avviene durante la fase di "analisi delle produzioni della grammatica in esame", mentre l'esecuzione delle azioni avviene durante la fase di "analisi della frase in input".

5.2.1 Preparazione delle azioni

In LLParser gran parte della preparazione delle azioni semantiche avveniva nella funzione *&modificatore*; é sembrato quindi logico intervenire su questa subroutine per permettere la gestione delle nuove funzionalitá richieste per la generazione del codice intermedio.

La funzione *&modificatore* esegue una "macroespansione" dell'azione semantica associata alla produzione in esame. Le azioni, scritte con la sintassi accettata da LLParser 2, vengono riconosciute e subito convertite in istruzioni Perl, in modo da poterle interpretare a run-time, mediante la funzione di valutazione semantica *eval* del linguaggio Perl.

Ci siamo adeguati all'uso della *eval* perché ció ha permesso di evitare l'introduzione di un ulteriore livello di interpretazione, portando sicuramente dei benefici alle prestazioni del sistema.

Le nuove istruzioni che vengono macroespanse da *&modificatore* sono:

newtemp: L'utente puó usare la versione sintattica di *newtemp* solo in una azione semantica di assegnamento, per esempio in {*#temp=newtemp*}. Questo significa che la funzione *&modificatore* riconoscerá l'operazione di assegnamento e richiederá l'esecuzione della *&splitassign*.

La funzione *&splitassign* riconosce questo tipo di assegnamento e genera una stringa contenente "...=\&newtemp()", ovvero contenente una chiamata alla funzione *&newtemp* da non eseguire subito, ma da eseguire a run-time.

Cosí, in fase di esecuzione dell'azione semantica, verrá assegnato alla variabile d'ambiente, presente alla sinistra dell'assegnamento, il valore di ritorno della *&newtemp*.

Con questa implementazione é perfettamente lecito scrivere azioni semantiche della forma {*#t=newtemp;emit(#t':='S.in+3);S.out=#t*}.

quad: Il valore *quad* denota la prima posizione libera dell'array contenente il 3-Address Code e si puó usare nelle espressioni delle azioni semantiche, come per esempio in $\{\#temp = 1 + quad; \#prova = quad; if A.attr = quad \ldots\}$. A seconda di dove *quad* viene usato, *&modificatore* richiederá l'esecuzione di una diversa subroutine.

Se *quad* viene usato nelle guardie dell'if o del while, *&modificatore* passa la stringa da "macroespandere" a *&sostvar*. Quest'ultima provvederá a riconoscere *quad* e a sostituirlo con la stringa "...\$quad..." che contiene la variabile "non interpolata" *$quad*.

Quando invece *quad* viene usato nelle espressioni degli assegnamenti, sará la *&splitassign* ad avere il compito di sostituirlo con la stessa tecnica.

"L'interpolazione di variabili scalari all'interno di stringhe" é una particolare tecnica di programmazione Perl che consiste nel seguente fatto: ogni variabile scalare *$a* nominata all'interno di una stringa "...$a..." é rimpiazzata con il suo valore corrente. La "non interpolazione" si realizza facendo precedere la variabile da un \ e consente di ottenere stringhe contenenti variabili non rimpiazzate dal loro valore corrente; se, successivamente, si valuta questa stringa con la funzione di interpretazione *eval*, allora la variabile sará finalmente interpolata, consentendo di rimpiazzarla con il suo valore corrente [6].

La non interpolazione della variabile *$quad* é fondamentale per il corretto funzionamento di LLParser 2. Potremo cosí fare riferimento, a run-time, all'effettiva prima posizione libera di *@emit*.

[]: Nella tecnica del backpatch é solito usare liste contenenti puntatori a istruzioni incomplete. Per tale motivo introduciamo l'operatore *[]* che consente di inizializzare una lista vuota. Si puó utilizzare *[]* in una istruzione di assegnamento, come ad esempio in {*#emptylist=[]*}.

Dato che, il meccanismo introdotto é usato nell'istruzione di assegnamento, ancora una volta la *&modificatore* invoca la *&splitassign* che ha il compito di riconoscere il nuovo operatore e di trattarlo.

La *&splitassign* restituisce una stringa contenente "...[]", che é una lista Perl vuota [6].

Preparare la stringa in questo modo, durante la fase di analisi delle produzioni, é fondamentale. Cosí, a run-time, potremo assegnare, alla variabile d'ambiente a sinistra dell'assegnamento, il puntatore ad una lista Perl vuota.

[quad]: Abbiamo appena detto che nella tecnica del backpatch é solito usare liste contenenti puntatori a istruzioni incomplete. Per tale motivo introduciamo anche l'operatore *[quad]* che consente di inizializzare una lista con l'attuale valore di *quad*. É possibile utilizzare *[quad]* in una istruzione di assegnamento, come ad esempio in {*#namelist=[quad]*}.

Anche in questo caso il meccanismo introdotto é usato nell'istruzione di assegnamento, quindi, ancora una volta, la *&modificatore* invoca la *&splitassign* che ha il compito di riconoscere il nuovo operatore e di trattarlo.

La *&splitassign* restituisce una stringa contenente "... [\$quad]", che é una lista Perl inizializzata con la sola variabile non interpolata *$quad*.

Preparare la stringa in questo modo, durante la fase di analisi delle produzioni, é fondamentale. Cosí, a run-time, potremo assegnare, alla variabile d'ambiente a sinistra dell'assegnamento, il puntatore ad una lista Perl contenente *quad*.

push(#lista,posizione): Ricordiamo che questa nuova istruzione inserisce *quad* o un numero intero in *#lista*.

Quando *&modificatore* riconosce questa istruzione, richiede alla subroutine *&splitvar* di esaminare *#lista*; se risulta non essere una variabile d'ambiente viene generato un errore. Se questa variabile non contiene una lista, a run-time, l'interprete Perl genererá un errore.

Per quanto riguarda l'argomento *posizione*, giá in fase di pattern matching viene permesso solo l'inserimento di un numero intero o del valore speciale *quad*; quest'ultimo viene eventualmente espanso nella variabile non interpolata *\$quad*.

Alla fine *&modificatore* restituirá l'istruzione Perl, che consiste in una stringa contenente una chiamata alla funzione push di Perl da non eseguire subito, ma da eseguire a run-time, con i seguenti argomenti:

- Il primo sará la variabile d'ambiente non interpolata esaminata da *&splitvar*
- Il secondo sará la variabile non interpolata *\$quad* oppure un numero intero.

BK(#lista,posizione): É l'istruzione sintattica che consente il backpatching del codice intermedio e si usa in uno dei due seguenti modi:

$\{BK(\#lista, quad); BK(\#lista, 12)\}$.

La subroutine *&modificatore* riconosce questa istruzione ed esamina i due argomenti come si é descritto nel caso della *push*.

Alla fine *&modificatore* restituirá l'istruzione Perl, che consiste in una stringa contenente una chiamata alla funzione semantica *&Backpatch*, da non eseguire subito, ma da eseguire a run-time, con i seguenti argomenti:

- Il primo sará la variabile d'ambiente non interpolata esaminata da *&splitvar*
- Il secondo sará la variabile non interpolata *\$quad* oppure un numero intero.

emit(istruzione): Consente l'inserimento di istruzioni nell'array del codice intermedio. Il riconoscimento della versione sintattica della *emit* é sempre compito di *&modificatore* che peró si avvale delle funzioni:

- *&PrepareEmit*
- *&TransformQuadrupla*
- *&VarAssegn*
- *&Argomento*
- *&splitvar*
- *&GestErroriVar*

che hanno il compito di riconoscere l'istruzione e di preparare la quadrupla, verificando la correttezza sintattica di istruzione e argomenti.

La funzione *&modificatore*, una volta preso l'argomento *istruzione*, invoca l'esecuzione della funzione *&PrepareEmit* con due parametri:

- L'*istruzione* parametro della *emit*
- Il Non Terminale padre della produzione che contiene l'azione *emit* da trattare.

&PrepareEmit riconosce l'operazione del linguaggio intermedio, della quale si vuole fare l'emit, e ne estrae quattro sottostringhe. Queste quattro stringhe sono una prima rappresentazione delle componenti della quadrupla che alla fine rappresenterá l'istruzione in esame. Le componenti inesistenti della quadrupla sono rappresentate con "_".

A seconda dell'operazione riconosciuta, la *&PrepareEmit* chiama opportunamente la subroutine *TransformQuadrupla* che richiede i seguenti cinque parametri:

- Le stringhe che rappresentano l'operatore, il primo operando, il secondo operando, il result
- Il Non Terminale padre della produzione che contiene l'azione *emit* da trattare.

La *TransformQuadrupla*, a seconda dell'operazione in esame, richiede la collaborazione di altre due funzioni:

- *&Argomento*
- *&VarAssegn.*

La *&Argomento* prende in input due parametri:

- Una stringa che rappresenta il primo o il secondo operando della quadrupla
- Il Non Terminale padre della produzione che contiene l'azione *emit* da trattare.

Questa funzione analizza il primo parametro per mezzo della *&splitvar* e poi verifica la correttezza lessicale e sintattica di variabili o attributi usati dentro ad un salto o ad una espressione a destra di un assegnamento. Inoltre traduce opportunamente le variabili d'ambiente #name che vengono usate dai salti per indicare la posizione di arrivo. Alla fine restituisce una stringa contenente la variabile non interpolata che era rappresentata dal parametro in input.

Invece la funzione *&VarAssegn* prende in input la stringa che rappresenta il result della quadrupla, la analizza per mezzo della *&splitvar* e poi ne verifica la correttezza lessicale e sintattica. Alla fine restituisce una stringa contenente la variabile non interpolata che era rappresentata dal parametro in input.

Appena la *&TransformQuadrupla* riceve l'output di queste ultime due funzioni, restituisce le quattro componenti della quadrupla alla *&PrepareEmit* che a sua volta la restituirá alla *&modificatore*.

A questo punto la *&modificatore* restituisce l'istruzione Perl, che consiste in una stringa contenente una chiamata alla funzione semantica *&RunEmit* da non eseguire subito, ma da eseguire a run-time, con i seguenti argomenti:

- L'operatore
- La variabile non interpolata per il primo operando
- La variabile non interpolata per il secondo operando
- La variabile non interpolata per il result

5.2.2 Esecuzione delle azioni

Descriviamo adesso le funzioni che devono essere eseguite a run-time per permettere una corretta generazione di codice. Giá nella sezione precedente si era accennato all'esistenza di alcune di queste funzioni, dato che rappresentano la funzione semantica delle nuove istruzioni sintattiche offerte agli utenti. Le nuove funzioni semantiche presenti in LLParser 2 sono:

&newtemp: é una funzione che restituisce una sequenza di nomi distinti, £1, £2, ..., in risposta a successive chiamate, rappresentanti locazioni di memoria del 3-Address Code. Il codice della funzione *&newtemp* incrementa di 1 il contatore globale di interi *$globalloc* e ne usa il valore per costruire un nuovo nome, concatenando la stringa £ con il val-

ore attuale della variabile *$globalloc*. Alla fine restituisce il nome cosí generato.

&Backpatch: Questa funzione prende in input due parametri:

- Un puntatore ad un array che contiene le posizioni delle istruzioni incomplete
- Il numero intero da inserire nelle istruzioni incomplete.

Se il primo argomento é veramente un puntatore ad un array, ne usa il contenuto per ritrovare le istruzioni incomplete ed effettuarne il backpatch con il secondo argomento; altrimenti restituisce un messaggio di errore.

&RunEmit: La *&RunEmit* prende in input quattro argomenti:

- L'operatore
- Il primo operando
- Il secondo operando
- Il result.

Questi quattro argomenti rappresentano la quadrupla del 3-Address Code che deve essere inserita nell'array dinamico *@emit*, nella posizione indicata dalla variabile *$quad*.

Quando si arriva a questo punto del codice, si ha giá la certezza della correttezza lessicale degli argomenti, dato che i controlli sono stati eseguiti durante la fase di preparazione delle azioni. Resta peró da effettuare un controllo di esistenza dei valori. Per eseguire questo controllo, la *&RunEmit* fa uso delle due seguenti funzioni:

- *&EsisteArg*
- *&EsisteRes*

La *&EsisteArg* viene chiamata due volte per verificare l'esistenza del primo e del secondo operando. Il parametro passatogli puó essere o una variabile d'ambiente o uno dei diversi tipi di attibuto possibili; in ogni caso la *&EsisteArg* ne controlla l'esistenza sul corrispondente stack di sistema. In presenza di problemi viene presentata la situazione di errore mediante adeguati messaggi.

La *&EsisteRes* viene chiamata, invece, solo per result, la quarta componente della quadrupla. Il funzionamento é simile alla precedente funzione, anche se sono diversi i controlli effettuati.

Una volta terminata l'esecuzione di queste due funzioni, la *&RunEmit* puó, prima, eseguire l'inserimento della quadrupla nell'array dinamico *@emit* e, successivamente, incrementare il valore di *$quad*.

&Emit: Questa funzione aggiorna quella giá esistente e viene richiesta la sua esecuzione dalla funzione *&OutPage*. La *&Emit* trasforma l'array dinamico *@emit* in un formato piú adatto all'output, e inoltre esegue un'altra operazione utile per l'utente: avvisa quando ci si dimentica di eseguire un backpatch, e si sono quindi lasciate delle istruzioni incomplete.

Tutte le funzioni descritte in questo capitolo hanno un sistema di controllo dell'input che avvisa l'utente in caso di immissione di dati non corretti. Per permettere ció, sono state utilizzate le funzioni *&OutErrPage* e *&GestErrVar* che, per quanto possibile, cercano di indicare il problema con messaggi di errore adeguati.

5.3 Riconoscimento dei lessemi

Come giá anticipato nel capitolo delle scelte di progetto, LLParser 2 non ha un analizzatore lessicale, perché scopo dello strumento é quello di concentrare l'attenzione degli utenti sulla fase di parsing. Comunque con LLParser 2 si é deciso di permettere l'inserimento di frasi che al posto dei token *ide* e *num*

contengano dei loro lessemi. LLParser 2 é in grado di capire se in input é presente o meno un lessema appartenente ad *ide* o *num*, e di comportarsi di conseguenza.

Questa nostra scelta impone all'utente, quando necessario, di utilizzare il token *ide* per la categoria lessicale degli identificatori e di usare il token *num* come categoria lessicale dei numeri interi.

L'implementazione di questa funzionalitá é stata trattata in due modi diversi, a seconda dell'uso o meno delle azioni semantiche.

5.3.1 Grammatiche non attributate

In questo caso, l'implementazione si é realizzata basandoci su due osservazioni:

- LLParser 2 non ha meccanismi che necessitano di informazioni relative a lessemi

- Non é richiesta, dall'utente, l'esecuzione di azioni semantiche.

Conseguenza di queste osservazioni é la possibilitá di fare a meno, in questo particolare caso, di una Symbol Table, permettendo al sistema di risolvere il problema senza appesantirlo troppo.

É stato necessario, prima di tutto, costruire il nuovo hash *%Terminali* che contiene tutte le keywords della grammatica analizzata. Per permettere un corretto funzionamento del meccanismo, *%Terminali* é stato inizializzato come segue:

```
$Terminali{‘‘num’’}=undef;$Terminali{‘‘ide’’}=undef;
```

Questo é stato necessario per impedire ad LLParser 2 di riconoscere il token *num* come un lessema del token *ide*, evenienza che avrebbe implicato l'errata trasformazione di *num* in *ide*.

Il passo successivo é stato il riconoscimento di tutte le keywords della grammatica, con la conseguente memorizzazione in *%Terminali*. La routine

che é sembrata piú idonea a contenere questa operazione é stata la *&Decomponi*. Infatti, questa funzione esegue la sua elaborazione selezionando ripetutamente tutti gli oggetti della grammatica [2]. Aggiungendo al suo interno l'istruzione

```
....
if ($token =~ /^([a-z][A-Za-z0-9]*|
             [\+\-\*\/\(\)\:\;\&\?\!\~\,\[\]\'\"\=\^\@><])$/)
                                {$Terminali{"$token"}=undef};
....
```

si é ottenuto la memorizzazione di tutti i token presenti nella grammatica, come chiavi della nuova struttura dati. L'uso dei token come chiavi, con associato il valore undef, faciliterá successivamente l'algoritmo durante la fase di ricerca dei token all'interno della struttura.

La parte successiva di implementazione é stata realizzata all'interno della funzione *NuovaAnalisiLL1*. Quando, a run-time, LLParser 2 troverá in input un lessema che non é keyword, dovrá controllare se appartiene alla categoria lessicale *ide* o *num*

```
....
if (!exists $Terminali{"$tok"}) {
    if ($tok =~ /^[0-9]+$/) {
        $tok = "num";
        next
        }
    if ($tok =~ /^[a-z][A-Za-z0-9]*$/) {
        $tok = "ide";
        next
        }
    }
....
```

Se la risposta é positiva, sostituirá il lessema con il rispettivo token e continuerá con la derivazione, altrimenti interromperá l'esecuzione con il messaggio di errore

```
‘‘String not recognized: input symbol isn’t a Terminal\n’’
```

5.3.2 Grammatiche attributate

Questo secondo caso é piú complesso del precedente, in quanto é necessario tenere traccia dei lessemi in input, associandoli ai rispettivi token, e per fare ció si deve utilizzare una Symbol Table [4].

Le strutture dati

Tra le strutture dati che si sono dovute usare, ricompare l'hash *%Terminali*, descritto nel paragrafo precedente. Sará utilizzato per evitare di confondere le keywords della grammatica con possibili lessemi.

Per implementare la Symbol Table si é pensato di utilizzare la tecnica dei package messa a disposizione dal Perl [7]. Ad ogni package, il linguaggio Perl associa una specifica Symbol Table, che é raggiungibile dalla Symbol Table principale di sistema *$main::*. Queste Symbol Table, adeguatamente usate dalle funzioni offerte da LLParser 2, consentono all'utente di realizzare un meccanismo in grado di mantenere le informazioni relative ai lessemi riconosciuti durante la fase di analisi, e di poterle utilizzare all'interno delle azioni semantiche. In pratica sará possibile associare, all'n-esimo blocco di codice, la Symbol Table *Bloccon::*.

Per tenere traccia, durante l'analisi, della sequenza di incapsulamento di blocchi e funzioni, si usa uno stack di identificatori di blocchi che in ogni istante ha sul top il blocco attivo. Questo stack si chiama *@blocchi* ed é in grado di gestire l'annidamento perché, in fase di analisi sintattica, sia i blocchi che le funzioni hanno sostanzialmente lo stesso comportamento. In fase di inizializzazione si inserisce la stringa *“Blocco0::”* nello stack *@blocchi*.

Le funzioni

Per implementare questa nuova funzionalitá sono state modificate funzioni giá esistenti, oltre ad introdurne di nuove. La prima funzione a subire variazioni é *&modificatore*, che ora deve essere in grado di riconoscere le nuove quattro istruzioni per la gestione della Symbol Table:

insert(nome,token,tipo): inserisce le informazioni del lessema nella Symbol Table del blocco attivo

lookup(nome): cerca nella Symbol Table il lessema passato per parametro

setblock: da invocare quando si vuole entrare in un blocco

resetblock: da invocare quando si vuole uscire da un blocco.

La *&modificatore* riconosce l'istruzione di *lookup* per un identificatore o per un numero nel modo seguente

```
if ($i =~ /^lookup\((ide[0-9]+)\.name\)$/) {
   ....
} elsif ($i =~ /^lookup\((num[0-9]+)\.value\)$/) {
   ....
}
```

Come si puó vedere dal codice, all'interno delle azioni semantiche é possibile fare riferimento al lessema del token *ide* con l'attributo *ide.name*, mentre a quello di *num* con *num.value*. Come per i Non Terminali, c'é la convenzione che, se in una produzione ci sono piú *ide* o piú *num*, i loro attributi si differenziano con *ide1.name, ide2.name, ...* o *num1.value, num2.value, ...*

Una volta riconosciuta l'istruzione di *lookup*, la funzione *&modificatore* restituirá l'istruzione Perl, che consiste in una stringa contenente una chiamata alla funzione semantica *lookup* da non eseguire subito, ma da eseguire a run-time, con argomento la variabile d'ambiente non interpolata relativa a *ideN.name* o *numN.value*, come si puó vedere dal codice

```
....
$tot.="\&lookup(\$sin[\$depth]{$1}{name});"
....
$tot.="\&lookup(\$sin[\$depth]{$1}{value});"
```

La *&modificatore* riconosce anche la successiva istruzione sintattica *insert* con

```
....
 elsif ($i =~ /^insert\((ide.+|num.+)\,([a-z]+)\,([a-z]+)\)$/)
....
```

e mediante la nuova *&splitvar* verifica la correttezza del primo argomento. Se ció che si vuole inserire é corretto, la *&modificatore* restituirá un'istruzione Perl che consiste in una stringa contenente una chiamata alla funzione semantica *&insert* da non eseguire subito, ma da eseguire a run-time, con gli argomenti:

- la variabile d'ambiente non interpolata relativa a *ideN.name* oppure a *numN.value*
- il token
- l'eventuale tipo del lessema.

Si é appena detto che la *&splitvar* deve riconoscere gli attributi *ideN.name* e *numN.value*; per tale motivo sono stati inseriti due nuovi pattern match al suo interno, che permettono il nuovo riconoscimento.

Tornando alla *&modificatore*, il riconoscimento delle due ultime istruzioni, *&setblock* e *&resetblock*, avviene in modo simile, con le espressioni regolari

```
....
 /^setblock$/
....
/^resetblock$/
```

e in tal caso la *&modificatore* restituirá un'istruzione Perl che consiste in una stringa contenente una chiamata alla funzione semantica *&set* o alla *&reset*, da non eseguire subito, ma da eseguire a run-time.

Durante la fase di derivazione, quando LLParser 2 troverá in input un lessema che non é keyword, dovrá controllare se appartiene o meno ad una delle due categorie lessicali ammesse e comportarsi di conseguenza. Questo controllo viene effettuato dalla funzione *&NuovaAnalisiLL1a*.

Se la *&NuovaAnalisiLL1a* non riesce a verificare l'uguaglianza tra il *Terminale* in input e l'eventuale *Terminale* in testa allo stack, prima di dare un errore di non riconoscimento, cercherá di capire se in input c'é un lessema.

Se la *&NuovaAnalisiLL1a* riconosce in input un lessema del token *num*, allora preparerá il messaggio

```
....
[transforming lexema $in[0] in ".$stack[-1]."]
....
```

incrementerá la corrispondenza per distinguere altre eventuali istanze del token *num* presenti nella stessa produzione

```
&inc($stack[-1]);
```

ed inserirá sullo stack dei sintetizzati l'attributo value di *num*

```
$sin[$depth]{"$stack[-1]"."$corr[$depth]{$stack[-1]}"}
                                          {"value"}=$in[0];
```

Solo a questo punto effettuerá la vera trasformazione, in testa all'input, del lessema numerico nel suo corrispondente token *num*.

Tutto ció si verifica anche nel caso di lessemi di *ide*.

Nella *&NuovaAnalisiLL1a*, inoltre, si é dovuto fare una modifica nel calcolo della seconda chiave della Tabella del Parser, cioé la chiave relativa al *Terminale* in input. Nel caso in cui ci fosse un lessema in input, la chiave restituita sará il token corrispondente.

Con queste modifiche apportate alla *&NuovaAnalisiLL1a*, il sistema LL-Parser 2 mette a disposizione dell'utente i lessemi trovati in input. Sará a questo punto compito dell'utente scrivere delle azioni semantiche idonee, che facciano uso delle quattro istruzioni di gestione della Symbol Table elencate sopra, che andiamo a descrivere.

La funzione *&insert* riceve in input il lessema, il token e l'eventuale tipo. Questi valori vengono inseriti nell'hash dell'attuale Symbol Table, il cui nome é mantenuto nella variabile *$pkg_name*. Per esempio, se il lessema in input é la stringa "prova", e siamo all'interno della Symbol Table *Blocco0::*, allora saranno effettuati gli assegnamenti

```
$Blocco0::prova{‘‘name’’} = il lessema ‘‘prova’’
$Blocco0::prova{‘‘token’’} = il token
$Blocco0::prova{‘‘type’’} = il tipo
$Blocco0::prova{‘‘loc’’} = undef
```

Se, invece, il lessema in input é il numero 21, e siamo sempre all'interno della Symbol Table *Blocco0::*, allora saranno effettuati gli assegnamenti

```
$Blocco0::num21{‘‘token’’} = il token
$Blocco0::num21{‘‘type’’} = il tipo
$Blocco0::num21{‘‘loc’’} = il lessema 21
```

La funzione ha anche un controllo sul tipo di token passato; se non riceve un *ide* o un *num*, restituisce un messaggio di errore.

La funzione *&set*, come prima cosa, incrementa di uno la variabile che indica l'attuale livello dello stack dei blocchi. Poi crea un nome per il nuovo blocco che sta per iniziare e lo inserisce in testa allo stack. In questo modo, ogni nuovo riferimento sará relativo alla nuova Symbol Table.

La funzione *&reset*, é la complementare della precedente. Elimina la Symbol Table del blocco in testa allo stack, aggiorna lo stack *@blocchi* e decrementa la variabile che indica l'attuale testa dello stack dei blocchi.

La funzione *&lookup* deve verificare la presenza del lessema in input all'interno della Symbol Table. Inizia controllandone la presenza all'interno

della Symbol Table del blocco che si trova in testa all'array *@blocchi*. Se la ricerca non ha buon esito, continua la ricerca all'interno delle Symbol Table dei blocchi presenti nei livelli sottostanti dello stack, fino al ritrovamento del lessema o fino all'esaurimento delle Symbol Table. In caso di risposta negativa, viene segnalato con un messaggio il non ritrovamento del lessema.

Chapter 6

Conclusioni

Lo scopo di questo lavoro di tesi é stato quello di aggiungere delle nuove funzionalitá ad LLParser, un parser Top-Down Predittivo, Generativo/Adattivo, di tipo LL(1). Il nuovo sistema, cosí ottenuto, é stato chiamato LLParser 2.

Una prima parte del lavoro é stata dedicata allo studio del sistema di partenza, in modo da comprendere profondamente le interazioni tra le sue componenti. Questo studio é stato fatto nel tentativo di riutilizzare, il piú possibile, degli algoritmi ben funzionanti.

Il primo tipo di modifiche, apportate ad LLParser, ha consentito l'uso di Terminali e Non Terminali piú espressivi di quelli precedentemente permessi. Inoltre, aumentando la libertá di input nell'interfaccia grafica, si é cercato di migliorare le giá buone caratteristiche del sistema.

Il passo successivo ha introdotto un nuovo meccanismo di generazione di codice intermedio per effetti laterali, che permette l'uso di tecniche ad una passata. Queste estensioni sono state realizzate seguendo le tecniche presentate nel corso di Compilatori, in modo da permettere agli studenti una maggiore facilitá d'uso.

Con LLParser 2 si é pure pensato di permettere, agli utenti interessati, di inserire frasi del linguaggio che al posto dei token *ide* e *num* contengano dei lessemi adeguati. Quest'ultima estensione non ha la pretesa di sopperire alla mancanza di un analizzatore lessicale, ma é stata introdotta per consentire

l'uso di azioni semantiche piú corrispondenti a quelle utilizzate durante il corso di Compilatori.

Dovendo intervenire su un codice preesistente, a volte anche con modifiche delicate, si é preferito lavorare con un sistema che consentisse il controllo delle versioni realizzate, con la possibilitá del recupero di versioni precedenti. La scelta é caduta sull'RCS (Revision Control System) che si é rivelato un ottimo strumento di lavoro. Ovviamente é stato naturale adeguarsi a tutte le scelte ben fatte per la realizzazione di LLParser: dal linguaggio di programmazione Perl, alla tecnica CGI di trasferimento dei dati, al server web Apache.

In conclusione di questo lavoro, diamo solo un cenno sulla possibilitá di nuovi ulteriori sviluppi. LLParser 2 consente agli studenti del corso di Compilatori di affrontare problematiche di tipo Top-Down, ma, con qualche ulteriore modifica, sarebbe possibile rendere il sistema fruibile anche per personale specializzato. A tale scopo potrebbe essere interessante lavorare al suo front-end ed al suo back-end. Sarebbe utile non tanto la realizzazione di un analizzatore lessicale, quanto dare la possibilitá di interfacciare LL-Parser 2 con diversi analizzatori lessicali. Altrettanto interessante sarebbe rendere LLParser 2 parametrico rispetto al linguaggio oggetto. Queste due nuove funzionalitá consentirebbero ad LLParser 2 quel salto di qualitá che lo renderebbe uno strumento utile anche nel mondo della sperimentazione di linguaggi.

Appendix A

Un link a LLParser: Il software

Questa appendice é il capitolo 4 estratto dalla tesi del Dott. Samuele Manfrin

Questo capitolo descrive il software implementato, dapprima con una panoramica sulle strutture dati, quindi con il dettaglio realizzativo. Ogni subroutine viene analizzata nelle sue funzionalitá, scendendo nel particolare qualora siano state effettuate delle scelte implementative peculiari. Ulteriori finezze o dettagli implementativi sono comunque riportati nel codice in appendice, abbondantemente commentato allo scopo, ed a cui si fa costantemente riferimento.

A.1 Struttura generale del programma

Il software sostanzialmente si sviluppa in tre parti. Nella prima, si effettua l'acquisizione e conversione dell'input proveniente dal CGI e si inizializza una parte delle strutture dati; nella seconda, viene preparato il parser con la generazione della tavola di parsing; infine la terza parte si occupa del riconoscimento della stringa di input e dell'esecuzione delle azioni. Inoltre, per motivi di sicurezza, viene tenuta traccia, in un opportuno file, di tutte le query giunte al programma stesso.

A.2 Strutture dati

Come é stato anticipato durante la descrizione del parser, le strutture dati necessarie al parser sono sostanzialmente tre: una tavola di parsing, uno stack semantico, ed una coppia di stack con ambienti per l'esecutore delle azioni. Il tutto fa uso di ulteriori strutture, per memorizzare la grammatica con e senza azione, l'input, i First ed i Follower.

La grammatica senza azioni viene memorizzata nell'array associativo %g tale per cui ogni elemento della chiave corrisponde alla parte sinistra delle produzioni, mentre il valore contenuto é dato da un reference ad un array di parti destre. Esempio (si noti che il simbolo ϵ viene rappresentato internamente dal carattere "@"):

```
S ::= aS | B          $g{S} = [ aS, B]
B ::= c | @           $g{B} = [ c, @ ]
```

Inoltre ogni produzione viene memorizzata, privata delle eventuali azioni, nell'array @prod. La grammatica con attributi viene mantenuta negli array referenziati dall'hash %ga; in particolare, @$ga{partesx} referenzia un array che contiene i reference all'array di tutte le parti destre delle produzioni per partesx, divise per token. Esempio, se

```
S ::= aS{S.tot=1+S1.tot} | @ {S.tot=0}
```

allora

```
${$ga{S}}[0]=['a','S','{S.tot=1+S1.tot}'] # reference ad array
${$ga{S}}[1]=['@', '{S.tot=0}']  # reference ad array
```

Questa scelta é stata effettuata per la semplicitá con cui Perl consente di risalire alle azioni ed ai singoli token di ogni produzione della grammatica.

Il risultato calcolo dei First e dei Follower per la tavola di parsing viene memorizzato sotto forma di stringa nei due array associativi %first e %fw con chiave rispettivamente tutte le parti destre e tutti i nonterminali presenti. Esempio:

```
S ::= aS | B   %first = ('aS'=>'a','B'=>'c@','c'=>'c','@'=>'@')
B ::= c | @    %fw = ('S'=>'$','B'=>'$')
```

La tavola di parsing viene mantenuta in due diverse modalitá: una senza azioni, per la visualizzazione all'interno delle pagine web, ed una comprensiva di azioni, impiegata in fase di valutazione semantica delle azioni. Rispettivamente tali dati sono contenuti nei due hash %table e %tablea; in entrambi la chiave rappresentata dalla stringa formata dalla giustapposizione tra il nonterminale sullo stack del parser ed il terminale in input, per identificare l'elemento della tavola di parsing che sará impiegato in fase di analisi. Nel primo hash viene mantenuta la produzione associata, nel secondo, un reference all'elemento corrispondente dell'array %ga.

Infine, gli stack di "ambienti" sono contenuti negli array @ere, @sin, @corr ed @amb che contengono rispettivamente degli opportuni hash per gli attributi ereditati ed i sintetizzati, la corrispondenza tra il nome della parte sinistra della produzione ed il nome del proprio padre nell'albero di parsing, ed un hash per l'ambiente locale di variabili utilizzabili dall'utente all'interno delle produzioni stesse.

A.3 Dettaglio implementativo

Come anticipato, vengono compiuti dei "macropassi" in fase di esecuzione, che si sviluppano nel seguente ordine:

- Acquisizione dei dati provenienti dal server web, l'input dell'utente
- Memorizzazione in un opportuno file di log dell'input passato (vedi paragrafo 4.3.4)
- Valutazione dei First per la tavola di parsing
- Valutazione dei Follower per la tavola di parsing
- Verifica dell'appartenenza della grammatica in input alla classe LL(1)

- Generazione della tavola di parsing
- Analisi LL(1) della stringa in ingresso, con esecuzione (eventuale) delle azioni
- Output finale

Da questo punto in poi saranno spiegate le tecniche impiegate e le ottimizzazioni di cui si é fatto uso per l'esecuzione di ciascuno di questi punti.

A.3.1 Acquisizione dati e preparazione della grammatica

```
Subroutines: read_parse, Takedata{Analizza, Decomponi{
Espansione{modificatore{strsplit, splitassign{sostvar},
evalguardia, evalemit}}}}
```

Il protocollo CGI invia allo standard input dello script il nome del campo indicato nella form html seguito dal segno di uguaglianza seguito a sua volta dal valore passato; ogni coppia viene separata dal segno di "&", ed i caratteri non alfanumerici sono codificati nella forma %XY dove XY é il valore esadecimale del corrispondente codice ASCII del carattere codificato. La funzionalitá di conversione da tale formato ad un altro fruibile dal software é effettuato dalla procedura read_parse che genera l'array associativo %in in cui si lega al nome dei campi il corrispondente valore. Tale subroutine é stata realizzata come adattamento di uno script freeware.

La subroutine &Takedata si occupa di settare tutti i "flag" utili seguendo le indicazioni dell'utente, ad esempio al fine di visualizzare o meno l'ambiente, o gli attributi delle azioni o il codice generato. Quindi estrae tutte le produzioni, aggregate nell'unica variabile $in{area}, convertendo (ed infine eliminando) tale variabile in modo che l'hash associ ad ogni parte sinistra delle produzioni la corrispondente parte destra normalizzata. Infine vengono inizializzati gli array associativi %ga e %g attraverso la chiamata ripetuta delle subroutine &Decomponi ed &Analizza su ogni produzione normalizzata.

La subroutine &Decomponi viene invocata con una parte sinistra ed una destra di una produzione con azioni, e riporta la stessa produzione privata delle azioni semantiche. Inoltre la grammatica con attributi sará posta, per effetto laterale, nell'array referenziato dall'array associativo %ga secondo le modalitá sopra descritte. Inoltre, sempre in %ga, vengono giá poste le azioni sintattiche e semantiche di {push} e &pushstack in testa, e {pop} e &popstack in coda ad ogni produzione, che saranno utili al parser in fase di esecuzione: tali procedure creano o distruggono un nuovo "ambiente locale" simulato atto a contenere attributi e variabili nel linguaggio delle azioni. La conversione semantica delle azioni viene compiuta dalla subroutine &Espansione.

La subroutine &Espansione accetta una parte sinistra ed un token sintattico presente in quella parte sinistra; se il token é un terminale o un nonterminale, semplicemente li riporta; qualora invece il token sia un'azione, il controllo viene passato alla routine &modificatore, che si occupa di effettuare una "macrespansione" dell'azione. Le azioni infatti vengono convertite in modo da risultare istruzioni Perl, al fine di poter essere interpretate a runtime attraverso la funzione di valutazione semantica eval(istr) presente nel linguaggio stesso. In questo modo si puó sfruttare, come "esecutore" delle azioni, Perl stesso, e non é quindi necessario introdurre un nuovo linguaggio, né una nuova funzione di interpretazione semantica; si noti che l'ambiente in cui la eval viene eseguita é quello di attivazione, e tale funzione lo puó modificare tranquillamente.

Attraverso la subroutine &modificatore viene effettuata la conversione dalla sintassi del linguaggio delle azioni a quella Perl; a tale scopo, quindi, vengono prese le istruzioni presenti nell'azione e macroespanse una ad una. Per raggiungere tale obiettivo, la subroutine &strsplit effettua sia una prima semplice macroespansione di token quali then, begin, eccetera (v. oltre), sia suddivide le istruzioni presenti, concatenate da punto-e-virgola. Si tenga conto che, a parte alcune modifiche marginali, la sintassi utilizzabile nel linguaggio delle azioni é la medesima del linguaggio Perl. Pertanto, le istruzioni

che vengono macroespanse in &modificatore sono le seguenti:

1. Assegnamento:

   X.attrib = expr oppure Xn.attrib = expr per assegnare un valore ad un attributo. #nomevar = expr per assegnare un valore ad una variabile nell'ambiente locale delle azioni. Una volta riconosciuta una stringa rispondente alle caratteristiche suindicate, viene passato il controllo alla routine &splitassign che prende l'intera istruzione e la parte sinistra della produzione cui appartiene l'azione comprendente l'istruzione stessa, e riporta l'istruzione "macroespansa" per essere utilizzata nel parser. Nel caso di assegnamento di attributi, é necessario riuscire a distinguere se l'attributo in assegnamento si riferisce ad un sintetizzato della parte sinistra della produzione o ad un ereditato di uno dei nonterminali che seguono, mentre ció che si assegna potrebbe contenere attributi sintetizzati della parte destra della produzione (e relativi ai nonterminali che precedono l'azione), oppure ereditati della parte sinistra. Vengono pertanto trattati tutti i casi possibili, generando il codice facente riferimento alle variabili che saranno presenti in fase di valutazione semantica. In particolare, viene sfruttata la funzione &splitvar che prende in ingresso un nome di variabile, e riporta una quadrupla composta dal nome del nonterminale, il suo attributo, il nome della variabile senza indice, ed un flag che indica se si é verificato un errore in fase di valutazione. La subroutine é pertanto sostanzialmente divisa in due parti: una per la gestione della parte sinistra dell'assegnamento, ed una per la gestione della parte destra. Un altro problema che si pone in fase di assegnamento, é relativo al settaggio delle corrispondenze tra il nome del nonterminale della parte sinistra della produzione ed il nome con cui, nell'albero di parsing,

il padre della produzione indica tale parte sinistra (ossia il nonterminale stesso seguito da un indice numerico). A tale scopo é stata introdotta la struttura @corr che contiene, per ogni livello, un numero (o una stringa vuota) corrispondente all'indice che la parte sinistra ha nel nome del padre e che viene settata in fase di sostituzione di un nonterminale della produzione sullo stack del parser. Si é scelto inoltre, per semplicitá, di assegnare i valori degli attributi della parte sinistra della produzione al nonterminale corrispondente al padre della produzione stessa. A tale scopo viene mantenuta una variabile $depth che contiene l'indice relativo alla dimensione degli stack (e quindi alla profonditá dell'albero di parsing su cui sta lavorando il parser in quel momento), che essendo memorizzati array, consentono di avere informazioni anche sugli elementi posti sotto al top dello stack stesso. A titolo esemplificativo, si supponga di avere le seguenti produzioni (che in maniera macchinosa calcolano la lunghezza di una stringa):

```
S ::= {A1.in=0}Ax{A2.in=A1.lun+1}A{S.lun=A2.lun}
                # azioni 1, 2 e 3
A ::= a{A1.in=A.in+1}A{A.lun=A1.lun} |
      @{A.lun=A.in}
                # azioni 4, 5 e 6
```

La macroespansione delle azioni porta al seguente risultato:

```
Azione 1:
   $ere[$depth]{A1}{in}=0
Azione 2:
   $ere[$depth]{A2}{in}=
     $sin[$depth]{A1}{lun}+1
Azione 3:
```

```
   $sin[$depth-1]{S$corr[$depth-1]{S}}{lun}=
     $sin[$depth]{A2}{lun}
Azione 4:
   $ere[$depth]{A1}{in}=
     $ere[$depth-1]{A$corr[$depth-1]{A}}{in}+1
Azione 5:
   $sin[$depth-1]{A$corr[$depth-1]{A}}{lun}=
     $ere[$depth]{A1}{lun}
Azione 6:
   $sin[$depth-1]{A$corr[$depth-1]{A}}{lun}=
     $ere[$depth-1]{A$corr[$depth-1]{A}}{in}
```

Si noti l'utilitá delle corrispondenze: la $corr[$depth-1]{A} nella quarta azione puó valere 1 o 2 a seconda se il padre della produzione, nell'albero di parsing, é il primo o il secondo A in S.

2. Condizionale:

   La sintassi di tale costrutto ha la forma if then [else] endif al fine di manipolare efficacemente la guardia ed eventuali sequenze di istruzioni. In particolare, si fa uso del costrutto &evalguardia che, presa in ingresso una guardia (una espressione composta di costanti e di variabili dell'ambiente delle azioni) effettua la trasformazione del suo contenuto attraverso l'uso della funzione &sostvar (giá utilizzata anche in &splitassign). Quest'ultima, presa una stringa in ingresso, effettua la macroespansione di ogni variabile dell'ambiente delle azioni (sintatticamente, un segno # seguito da uno o piú caratteri alfanumerici). Ad esempio, la stringa

   ```
   A.a=(\#t+1)*A.b
   ```

   viene macroespansa in

```
A.a=($env[$depth]{t}+1)*A.b
```

3. Iterazione:

   Le iterazioni sono gestite dal costrutto while il quale, per la valutazione della guardia, si appoggia alle medesime subroutine del condizionale.

4. Blocco:

   La gestione dei blocchi in Perl viene effettuata in maniera analoga al C, ovvero attraverso l'uso di parentesi graffe. Al fine di non interferire con la sintassi delle azioni internamente alle produzioni, é stato necessario introdurre i costrutti begin-end per la delimitazione dei blocchi di istruzioni. In fase di macroespansione, questi costrutti vengono sostituiti con parentesi graffe, rispettivamente aperta e chiusa.

5. Emit:

   La emit rappresenta il "vero" effetto laterale effettuabile dal parser: suo scopo é quello di produrre un output, composto di costanti e di variabili dell'ambiente locale all'azione. Tale output viene mantenuto nella variabile globale $emit il cui contenuto viene poi visualizzato a fine esecuzione. La subroutine &evalemit viene utilizzata, insieme a &sostvar, anche per affrontare alcuni problemi di security.

Al termine della chiamata, la subroutine &Decomponi riporta la produzione "ripulita" delle azioni semantiche. Tale risultato viene quindi passato alla procedura &Analizza che verifica la correttezza sintattica della produzione ed assegna ogni elemento della parte destra della produzione all'array @$g{psx} dove psx la parte sinistra della produzione. Inoltre viene aggiunta la produzione passata ad un opportuno array @prod, con finalitá che saranno descritte in seguito.

A.3.2 La preparazione della tavola di parsing

Nel primo capitolo é stato spiegato come la tavola di parsing, il "cuore" di tutto il processo di riconoscimento di stringhe di una grammatica, necessita, per essere costruita, dei due insiemi First e Follow. La metodologia applicata per la creazione di tali insiemi non rispecchia esattamente la descrizione effettuata, ma impiega algoritmi studiati appositamente al fine di velocizzare ed ottimizzare, in funzione del linguaggio Perl, il calcolo di tali insiemi.

Il calcolo di First

`Subroutines: ValutaFirst{ValutaFirstToken, PulisciInsieme}`

Il metodo per la valutazione dei First é intrinsecamente ricorsivo, e tale caratteristica viene sfruttata anche a livello algoritmico. A tale scopo, viene fatto uso della subroutine &ValutaFirst che prepara ed inizializza opportunamente le strutture dati per la routine ricorsiva &ValutaFirstToken. Questa ultima subroutine viene utilizzata anche nel calcolo dei Follower, come si vedrá nel paragrafo successivo.

L'approccio seguito é analogo a quello descritto nel cap. 3. In particolare, la routine &ValutaFirst prende tutte le parti destre di ogni produzione senza azioni, e su queste invoca la routine &ValutaFirstToken che si occupa di effettuare, ricorsivamente, la ricerca del First sulla parte destra passata. Se il primo simbolo di tale parte destra é un terminale, la subroutine riporta semplicemente tale terminale al chiamante. In caso contrario viene valutato, sempre ricorsivamente, il First di tutte le parti destre delle produzioni che hanno come parte sinistra tale nonterminale (ad esempio, X), e vengono accodate ad un'unica stringa, che fa le veci di un insieme. Se al termine in tale stringa si ritrova un ϵ allora é necessario eliminarlo, ed aggiungere all'insieme risultante anche i First del simbolo che segue X, e ripetere l'operazione insiemistica finché un simbolo riporta un First senza ϵ (e quindi il First risultante non contiene ϵ) oppure si é analizzata tutta la parte destra (e quindi ϵ va nuovamente aggiunto alla stringa). Chiaramente, se si cerca di espan-

dere un nonterminale che si é giá analizzato in precedenza, viene segnalata la presenza di una ricorsione sinistra in uno o piú passi.

Come effetto laterale, la procedura &ValutaFirst inizializza la variabile $xt con la stringa contenente tutti e soli i terminali presenti nelle produzioni, ed effettua il corretto assegnamento dell'array associativo %first in cui le chiavi sono tutte le parti destre delle produzioni, ed i cui valori associati sono proprio gli insiemi First relativi.

La subroutine &PulisciInsieme svolge una funzione di comodo, ovverosia elimina i duplicati dall'insieme dei First (che internamente é mantenuto in una stringa).

Il calcolo di Follow

`Subroutines: ValutaFW{ValutaFirstToken, PulisciInsieme}`

L'approccio seguito per la generazione dell'insieme Follow é leggermente diverso da quello che é stato descritto nel Capitolo 3.

L'algoritmo di valutazione é effettuato in modo iterativo: nonostante la formulazione del metodo sembrerebbe definire una realizzazione ricorsiva, in realtá la metodologia applicata, che sfrutta al meglio le peculiaritá di Perl, non si presta a tale tipo di implementazione.

La valutazione viene effettuata in due distinte fasi. Nella prima, si prende ogni singolo simbolo di ogni parte destra di ogni produzione per ogni nonterminale; se questo é un terminale, lo si ignora e si passa al simbolo seguente (e alla parte destra successiva). Se é un nonterminale, ad esempio X, si valuta il First di tutto ció che lo segue (se X non é seguito da nulla, il First viene considerato pari ad ϵ). In tale First si sostituisce ϵ, se presente, con la parte sinistra della produzione; infine si aggiunge tale First all'insieme dei Follower di X. Cosí, nel Follow possono essere presenti, in questa prima fase, sia simboli terminali, sia nonterminali. Dato che non é possibile valutare fin da subito i Follower che andrebbero inseriti nell'insieme, ogni nonterminale inserito svolge un ruolo di "marca". Scopo della seconda fase, quindi, é quello

di sostituire opportunamente ciascuna "marca" nonterminale con il proprio insieme di Follower.

Il problema fondamentale da gestire nella seconda fase é quello che si puó presentare in caso di sostituzione di un nonterminale con un Follower che, in una o piú successive sostituzioni, genera nuovamente se stesso. Si osservi ad esempio il caso seguente:

```
$fw{C} = 'tuvN';
$fw{M} = 'N';
$fw{N} = 'MS';
$fw{S} = '$';
```

Si supponga che si stia analizzando il Follow di C. Allora quello che accadrebbe, iterativamente, sarebbe:

```
$fw{C} = 'tuvN';   sostituzione di N con $fw{N}
$fw{C} = 'tuvMS';  sostituzione di M con $fw{M}
$fw{C} = 'tuvNS';  nuova sostituzione di N con $fw{N} ma...
$fw{C} = 'tuvMSS'; ...entra in un loop infinito
```

Per evitare questo problema, si procede in questo modo. Si supponga di dover "espandere" il Follow(X). Innanzi tutto, si rimuove dal Follow il simbolo X, quello della parte sinistra della produzione. Quindi si prendono tutti i nonterminali V_i presenti in Follow(X), li si sostituiscono con i corrispettivi Follow(V_i) e si controlla se il Follow(X) non cambia anche a seguito di queste sostituzioni: in tal caso viene trovato un "punto fisso" che non puó ulteriormente essere espanso, e quindi le sostituzioni per Follow(X) terminano. I nonterminali eventualmente presenti possono essere quindi eliminati (infatti una loro ulteriore espansione non modifica l'insieme), e si puó passare ad espandere un ulteriore Follow.

Come effetto laterale, la procedura calcola quindi l'array associativo %fw che contiene, appunto, gli insiemi di Follower di ogni nonterminale della grammatica, nonché aggiorna la variabile $xt con tutti i terminali presenti.

La verifica del tipo di grammatica

`Subroutine: CheckLL1{Controlla}`

Una volta calcolati First e Follow, viene controllato il tipo di grammatica, attraverso la chiamata della subroutine &CheckLL1. Per ogni nonterminale, vengono aggiunti ad una stringa tutti i First delle parti destre della produzione che ha come parte sinistra tale nonterminale; al termine, se é presente un ϵ questo viene sostituito con i Follower del nonterminale in esame. La stringa risultante viene passata alla subroutine &Controlla che verifica che la stringa risultante non abbia caratteri ripetuti (quindi che l'intersezione fra tutti gli elementi della stringa sia vuota). In caso questo controllo non sia passato positivamente, viene segnalato un avviso, e la generazione di una tavola avverrá comunque; ciononostante, il parser non analizzerá la stringa con azioni passata dall'utente.

A.3.3 L'output: il parser in azione

```
Subroutines: Outpage{MkTable, AnalisiLL1a{ShowAttrib,
inc, pushstack, popstack}, AnalisiLL1{ToS}, Emit}
```

Il parsing vero e proprio viene fatto in concomitanza con l'output della pagina web che viene inoltrato dal server al client. Questa scelta é stata fatta per due motivi. Innanzi tutto si voleva evitare di generare tutto l'output in memoria per poi inviarlo in blocco in output al termine dell'elaborazione: in caso di piú accessi conconcorrenti al server, questo avrebbe potuto provocare dei rallentamenti o dei timeout di collegamento, oltre, ovviamente, a sprechi inutili di risorse. In secondo luogo, questa soluzione effettua una sorta di "pipelining" tra l'elaborazione vera e propria e l'output generato: mano a mano che il parser lavora, manda al client il risultato dell'elaborazione.

La subroutine &OutPage si occupa d'inviare in output il risultato della analisi top-down. Il formato generico della pagina che viene generata é contenuto in un opportuno file esterno, llparser.dat, all'interno del quale possono

essere presenti sei diversi "tag" che vengono rimpiazzati, a tempo di esecuzione, da varie parti degli output disponibili, secondo lo schema seguente:

- %%PRODUZIONI%% : Questo tag viene sostituito dal corpo di una tavola html e comprenderá tutte le produzioni presenti nella grammatica

- %%FIRST%% : Qui viene messo il corpo di una tavola che mostrerá tutti i First di ogni parte destra di ogni produzione della grammatica

- %%FOLLOWER%% : Tag sostituito dal corpo di una tavola contenente tutti i Follower di ogni nonterminale

- %%ERRORS%% : Gli errori vengono visualizzati al posto di questo tag

- %%TAVOLA%% : Questo tag segnala il luogo dove deve comparire la tavola di parsing

- %%ENGINE%% : Al posto di questo tag viene messo tutto il risultato dell'elaborazione della stringa in input, il suo riconoscimento, ed ogni altra informazione che il parser in grado di emettere

La subroutine, quindi, legge una alla volta le righe di questo file esterno; se in una riga non é presente nessuno di questi tag, allora la riga viene semplicemente inviata in output. Qualora invece sia presente uno di tali tag, allora l'intera riga viene sostituita con i contenuti specifici relativi al tag considerato.

Chiaramente, a questo punto dell'elaborazione sono completamente disponibili tutti i dati per la sostituzione dei primi quattro "tag". Il quinto tag viene espanso attraverso la chiamata della subroutine &MkTable in cui vengono sia creati i due hash %table e %tablea precedentemente descritti. Si noti che l'hash %tablea é stato costruito per comoditá: infatti ogni suo elemento é semplicemente del tipo

```
$tablea{‘‘$k$c’’} = \@{${$ga{$k}}[$c]}
```

L'ultimo tag, %%ENGINE%%, viene sostituito, in caso di errori nella valutazione dei First o dei Follower (ad es. in caso di grammatica ambigua), da un messaggio opportuno, e non viene compiuta nessun'altra azione. In caso non vi siano stati problemi, invece, si verifica se l'utente ha deciso di non richiedere la valutazione delle azioni della grammatica. In caso positivo, viene richiamata la subroutine &AnalisiLL1 che non effettua alcuna valutazione delle azioni, velocizzando quindi l'output e l'elaborazione, senza introdurre inutilmente le strutture dati necessarie alle grammatiche attributate. Questa subroutine non é altro che una semplificazione della subroutine &AnalisiLL1a che effettua il lavoro nella sua totalitá. Questa scelta é stata fatta al fine di mantenere piú leggibile il codice, a scapito di una maggiore lunghezza di una ventina di righe del programma stesso.

La subroutine &AnalisiLL1a si occupa della maggior parte del lavoro. Preleva la stringa da analizzare, prepara gli stack di parsing e di input, quindi entra nel ciclo principale che implementa l'algoritmo descritto nel paragrafo 2.1. Si noti che nello stack del parser puó esserci un simbolo (terminale o nonterminale) oppure un'azione. In presenza di un terminale, si verifica semplicemente la presenza nello stack di input del medesimo simbolo, si effettua un "pop" da entrambi gli stack e si procede oltre. In fase di sostituzione di un nonterminale con la parte destra, si prepara l'azione facendola precedere da un'istruzione di incremento della profonditá dello stack, quindi si mette tutto ció al posto del nonterminale in sostituzione. Un'azione, invece, é caratterizzata dal fatto di essere sullo stack sotto forma di reference ad un array di due elementi. Il primo elemento racchiude semplicemente il testo che deve essere visualizzato in fase di output per l'utente, mentre il secondo contiene l'azione Perl che deve effettivamente essere valutata dall'interprete semantico. Viene quindi estratto questo reference, passato il primo elemento alla funzione di valutazione semantica, intercettato e visualizzato ogni eventuale messaggio di errore proveniente dall'interprete Perl, quindi, se l'utente lo ha richiesto, viene chiamata la subroutine &ShowAttrib che genera una tavola con il contenuto degli stack degli attributi e dell'ambiente. Si noti che

é stato necessario, subito dopo la valutazione, introdurre una istruzione di "rumore" per ovviare ad un probabile "bug" nella funzione di eval (che per inciso non é stato possibile isolare): é semplicemente necessario usare un elemento degli hash usati nella eval stessa per farlo "comparire" nell'ambiente globale; se tale azione non viene compiuta, silenziosamente (e misteriosamente) le variabili scompaiono. Si presume si tratti di una mancata riattivazione della visibilitá di tali variabili nell'ambiente locale al termine della eval, che peró viene effettuata semplicemente nominandole in una istruzione di assegnamento. Tra le azioni, come introdotto in paragrafo 4.3.1, possono esservi delle "pushstack" o "popstack": queste non fanno altro che aggiungere in coda o rimuovere l'ultimo elemento agli stack (array) degli attributi, degli ambienti, dei riferimenti. Con la "pushstack" viene congelato l'ambiente delle azioni corrente con la creazione di un nuovo ambiente vuoto che viene posto opportunamente negli stack (array) degli ambienti delle azioni, e viene quindi passato il controllo ad un figlio del nodo corrente. La "popstack" rappresenta l'operazione inversa, ovvero il completamento delle azioni (e della visita) di un nodo dell'albero di parsing: quindi distrugge l'ambiente locale corrente (poiché il padre ha giá ricevuto il valore dei suoi attributi sintetizzati dal nodo corrente), ripristinando l'ambiente del livello superiore. Una terza azione, di "inc" si occupa di settare la giusta corrispondenza tra il nonterminale in esame ed il suo indice corrispondente all'interno della produzione, cosicché quando il parser "scende" lungo l'albero la corrispondenza resta corretta; tale azione viene compiuta ad ogni sostituzione di un nonterminale sullo stack con la corrispondente parte destra. Il ciclo viene ripetuto finché ci sono elementi in uno dei due stack del parser o di input, quindi viene verificato il riconoscimento della stringa da parte del parser.

Una volta completata l'analisi, se richiesto dall'utente viene mostrata una ulteriore tabella contenente il risultato delle azioni di emit interne al codice delle azioni attraverso la chiamata della routine &Emit che formatta opportunamente l'output.

A.3.4 Security e Tainting

Perl offre la possibilitá di programmare in tutta sicurezza applicazioni i cui dati possono provenire da fonti insicure, esterne al programma stesso, quali possono essere le variabili di ambiente, o quelle passate da script CGI, o i parametri offerti sulla linea di comando, o quelli acquisiti da input. Infatti utenti maliziosi potrebbero trovare il modo di passare al programma dei parametri che potrebbero forzare il software stesso ad eseguire delle istruzioni diverse da quelle per cui é stato concepito. Un classico esempio é il passaggio di parametri scritti in un formato tale per cui, una volta assegnati a variabili interne al programma, possono essere interpretati da Perl in modo da essere eseguiti in una shell di sistema. Per ovviare a questo tipo di attacchi, é stato sviluppato un sistema abbastanza sottile per trattare questo tipo di informazioni intrinsecamente insicure, detto di tainting. Quando Perl é in taint mode, vengono prese delle precauzioni speciali: non possono essere utilizzati dati provenienti dall'esterno per influenzare qualcos'altro (variabili, file, eccetera), esterno anch'esso al programma stesso. Tutto ció che proviene dall'esterno, pertanto, viene "marcato" come tainted; tutto ció che si trova in questo stato non puó essere usato direttamente o indirettamente in alcuna operazione che coinvolga l'apertura di shell di sistema, o in operazioni che modifichino file, directory o processi. Questo stato, inoltre, viene passato in "ereditá" ad eventuali oggetti interni al programma stesso che usino strutture giá tainted; ad esempio, una variabile locale che venga assegnata facendo uso di una variabile tainted diventa tainted anch'essa.

La creazione di script CGI che effettuino delle "eval" di input passato dall'utente é sostanzialmente un grosso rischio di security: un utente malizioso potrebbe inserire delle azioni in grado di accedere a dati o informazioni sensibili presenti nel sistema su cui si trova il server web stesso. A tale scopo, pertanto, si é pensato in un primo momento di fare uso del meccanismo di tainting per avere un sistema automatico di controllo sulla "pulizia" dei dati inseriti. Si é notato peró che praticamente tutti gli identificatori presenti nel software sarebbero risultati tainted in brevissimo tempo, e quindi la maggior

parte del codice di questo lavoro di tesi avrebbe dovuto essere dedicata a controllare e gestire attivazione e disattivazione del taint. Inoltre é stato rilevato un bug nelle espressioni regolari che coinvolge gli hash e gli array tainted il cui controllo sarebbe risultato di notevole complessitá (vedi Appendice A). Pertanto si é preferito abbandonare questa idea, ed applicare un sistema diverso, costruito ad hoc per l'applicazione ed in grado di offrire un sufficiente grado di affidabilitá.

Innanzi tutto viene invocata, ad ogni chiamata del software, la subroutine &QueryLog che memorizza, in un opportuno file di log, l'indirizzo IP della macchina che ha invocato lo script, data ed ora della chiamata, ed il contenuto di tutti i campi passati. Inoltre, in fase di analisi sintattica delle azioni, vengono rimosse sistematicamente alcune parole chiave (ad esempio system, exec, eval, la virgoletta rovesciata, eccetera) che potrebbero essere sfruttate per compiere azioni inopportune. In questo modo si é resa l'applicazione sufficientemente sicura, anche se non é del tutto garantito che sia possibile utilizzare esotiche combinazioni di funzionalitá Perl per ottenere dei risultati "insicuri". Ciononostante il presente lavoro di tesi non ha lo scopo di studiare dei metodi altamente efficienti in tema di sicurezza, quindi si é ritenuto opportuno non approfondire ulteriormente l'argomento.

Appendix B

Il codice

Bibliography

[1] A. V. Aho, R. Sethi, J. D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley Publishing Company, 1986.

[2] LLParser: lavoro di tesi di S. Manfrin, 2001.

[3] Parson, Thomas W. Introduction to compiler construction.

[4] Jean-Paul Tremblay, Paul G. Sorenson. The Theory and practice of compiler writing.

[5] Lee, John A. N. The anatomy of a compiler.

[6] Randal L. Schwartz & Tom Phoenix. Learning Perl. O'Reilly, 2001 - 3rd Edition.

[7] Srinivasan, Sriram. Advanced Perl Programming. O'Reilly.

[8] L. Wall, T. Christiansen, J. Orwant. Programming Perl, 3rd Edition. O'Reilly & Associates, 2000.

[9] T. Christiansen, N. Torkington. Perl Cookbook. O'Reilly & Associates, 1998

[10] Johan Vromans. Perl 5 Pocket Reference. O'Reilly & HOPS, 2000 - 3rd Edition.

[11] Michael Moncur. JavaScript 1.3. Tecniche Nuove, 1999.

[12] Tecniche di programmazione in Html e Javascript: http://www.html.it

[13] Hein, Jochen. Linux companion for system administrator.

[14] Dilip C. Naik. Internet: Standard e protocolli. Microsoft Press & Mondadori Informatica, 1999.

[15] John Viega, Gary McGraw. Software sicuro. Apogeo, 2002.

[16] S. C. Johnson. YACC - Yet Another Compiler-compiler. Bell Laboratories, Murray Hill, NJ, July 1978

[17] B. Laurie e Peter Laurie. Apache: La guida. Apogeo 1999.

[18] The Apache Software Foundation. The Apache/Perl Integration Project. http://perl.apache.org/, 2001.

[19] J. Orwant, J. Hietaniemi, J. Macdonald. Mastering Algorithms with Perl. O'Reilly & Associates, 1999.

[20] Manuale in linea. perldata(1) - Perl data types. Perl Programmers Reference Guide, V. 5.6.0, 2000.

[21] Manuale in linea. Perlfaq7(1) - Perl language Issue, What's the difference between dynamic and lexical (static) scoping? Between local() and my()? - User Contributed Perl Documentation, Rev. 1.38, 1999.

[22] Manuale in linea. perlfunc(1) - Perl builtin functions. Perl Programmers Reference Guide, V. 5.6.0, 2000.

[23] Manuale in linea. perlLoL(1) - Manipulating Lists of Lists in Perl. Perl Programmers Reference Guide, V. 5.6.0, 2000.

[24] Manuale in linea. perlobj(1) - Perl objects. Perl Programmers Reference Guide, V. 5.6.0, 2000.

[25] Manuale in linea. perl(1) - Practical Extraction and Report Language. Perl Programmers Reference Guide, V. 5.6.0, 2000.

[26] Manuale in linea. perlre(1) - Perl regular expressions. Perl Programmers Reference Guide, V. 5.6.0, 2000.

[27] Manuale in linea. perlref(1) - Perl references and nested data structures. Perl Programmers Reference Guide, V. 5.6.0, 2000.

[28] Manuale in linea. perlsec(1) - Perl security. Perl Programmers Reference Guide, V. 5.6.0, 2000.

[29] Manuale in linea. perlsub(1) - Perl subroutines. Perl Programmers Reference Guide, V. 5.6.0, 2000.

[30] Manuale in linea. perlsyn(1) - Perl syntax. Perl Programmers Reference Guide, V. 5.6.0, 2000.

[31] Manuale in linea. Perlmod(1) - Perl modules (packages and symbol tables). Perl Programmers Reference Guide, V.5.6.0, 2000.