TESI
Ausilio informatico per non vedenti per la manipolazione di espressioni matematiche in TEX

Relatori: Ch.mo Prof. Giuliano Artico
Ch.mo Prof. Michele Moro

Laureando: Massimiliano Zattera

Anno accademico 1996–1997

Indice

Sommario

Attualmente gli studenti non vedenti od ipovedenti che si avvicinano alla matematica incontrano grosse difficoltà per la mancanza di un valido strumento per la rappresentazione di formule matematiche: il braille non contiene un set standard di simboli ad uso matematico e gli screen reader non possono leggere agevolmente le formule. Il problema si fa più sentito a mano a mano che si prosegue nella carriera di studi passando alle scuole medie superiori od all'università.

Con questo lavoro ci si propone di fornire un ausilio informatico che permetta a chi non può leggere un testo scritto di comprendere e manipolare agevolmente espressioni matematiche.

Il programma che è stato realizzato: TESI, accetta come input un file TEX contenente delle formule matematiche; di queste formule viene costruito un albero sintattico la cui visita viene resa possibile tramite sintesi vocale. In questo modo si fornisce all'utente un'idea dell'organizzazione "spaziale" della formula, supplendo alla mancanza di informazioni sulla sua struttura disponibili a chi può accedere normalmente al testo scritto. In alternativa, l'utente può esplorare il file TEX spostandosi al suo interno col cursore, oppure può modificarlo richiamando un editor esterno di sua scelta.

L'intero programma basa il suo funzionamento su file di configurazione che definiscono i comandi TEX da riconoscere, indicando la loro sintassi e la modalità con cui vanno letti all'utente.

Per l'output, TESI si appoggia su un qualsiasi screen reader: il testo da sintetizzare viene infatti stampato in un'apposita finestra video da cui può essere intercettato e letto, non ci soffermiamo sugli ovvi vantaggi di una simile soluzione, sia in termini di praticità che di economicità.

Durante l'utilizzo del programma compaiono a video informazioni di supporto per un eventuale istruttore vedente: ad esempio viene evidenziato, all'interno del file TEX, il frammento di formula di cui sta avvenendo la sintesi.

TESI risulta in grado di riconoscere e di presentare in modo comprensibile la quasi totalità delle formule matematiche presenti in testi di livello universitario su cui è stato testato.

I risultati conseguiti suggeriscono l'opportunità di spingere alla diffusione fra i non vedenti del TEX e di altri strumenti orientati all'organizzazione logica del testo (es. l'HTML), e non al suo aspetto tipografico (come gli editor WYSIWYG). In questo modo si scavalcherebbe la barriera costituita dall'interfaccia grafica ed anche i non vedenti potrebbero produrre testi stampati di notevole qualità tipografica, nonché facilmente convertibili in formato digitale, braille, etc.

TESI potrebbe diventare il primo modulo di un editor per non vedenti orientato alla struttura del documento in grado di gestire indifferentemente testi TEX o pagine WWW, il tutto senza dover gravare sull'utente imponendogli l'acquisto di hardware dedicato.

Introduzione

Questo capitolo vuole fornire alcune informazioni basilari sul programma TeSI, oggetto di questa tesi di laurea.

Dopo aver brevemente spiegato cos'è il TeX, si darà una descrizione sintetica di TeSI e degli obiettivi che questo lavoro si propone di raggiungere, nonché si illustreranno i concetti chiave alla base del funzionamento del programma e si introdurranno alcuni termini di cui si farà uso nella trattazione seguente. Inutile dire che si consiglia caldamente la lettura di queste ultime sezioni prima di proseguire nella lettura degli altri capitoli.

Il capitolo 1 costituisce il manuale per l'utente: qui viene spiegato come installare il programma e come lanciarlo; inoltre, sono descritti i comandi disponibili durante l'esecuzione del programma.

Nel capitolo 2 si spiega come configurare TeSI mediante i file di configurazione, la lettura del capitolo è consigliata solo agli utenti più esperti.

Il capitolo 3 definisce i formalismi impiegati per la trattazione dei linguaggi formali; inoltre, viene illustrato il funzionamento e la struttura interna dei compilatori. Qui sono anche presentati `flex` e Bison, due strumenti chiave per la realizzazione di TeSI. Tutta la terminologia introdotta in questa parte del lavoro sarà utilizzata nei capitoli seguenti. Chi è già familiare con gli argomenti trattati in questo capitolo può naturalmente evitarne la lettura.

I restanti capitoli descrivono la struttura interna di TeSI e sono pertanto riservati a che è interessato alla parte puramente tecnica del lavoro: il capitolo 4 descrive il modulo che esegue l'elaborazione dei file di configurazione di TeSI. Il capitolo 5 spiega le modalità secondo cui viene processato il file TeX in input. Il capitolo 6 illustra il funzionamento delle routine di interfaccia con l'utente. Infine, il capitolo 7 raccoglie la descrizione di alcune funzioni importanti non trattate negli altri capitoli.

Chiudono il testo alcune considerazioni di chi scrive sul lavoro svolto.

Convenzioni tipograche

Si illustrano ora alcune convenzioni tipografiche che verranno usate in seguito.

I comandi da impartire al computer, il contenuto dei file o l'output generato a video

dal programma saranno evidenziati nel modo seguente:

```
C:\ DIR

PIPPO.TEX   21345   12/3/97

21345 BYTES USED
124213432 BYTES FREE
```

Singoli tasti che l'utente deve premere durante l'interazione col programma vengono evidenziati come: **Tasto**. Un + indica che i tasti vanno premuti contemporaneamente, una virgola separa diverse combinazioni di tasti. Ad esempio:

Ctrl + **C**

indica che l'utente deve premere il tasto "C" tenendo contemporaneamente premuto il tasto "Control", solitamente indicato come "CTRL" sulle tastiere.

L'output del sintetizzatore vocale viene indicato con doppie parentesi uncinate: ≪...≫. Così la scrittura:

≪*1, 2, 3, prova*≫

indica che il sintetizzatore vocale pronuncierà il testo "uno, due, tre, prova".

Cosa sono TEX, LATEX, $\mathcal{AMS}$-TEX

TEX, secondo la definizione del suo creatore Donald E. Knuth, è un "typesetting system", termine traducibile all'incirca come "sistema di composizione tipografica". TEX è un programma pensato per la creazione di stampati di alta qualità estetica che assolve le funzioni di un word processor, ma non è un word processor.

L'utente di TEX scrive il suo testo come file ASCII all'interno del quale specifica alcuni parametri di formattazione, ad esempio per definire l'inizio di un nuovo capitolo si usa il comando[1]:

```
\chapter{Titolo del capitolo}
```

in fase di creazione del documento TEX inserirà in quel punto un nuovo capitolo, scrivendone il titolo con font opportuni, saltando pagine se necessario, il tutto in obbedienza ad uno "stile" specificato all'inizio del documento.

Come si può intuire da questo esempio, chi scrive documenti usando TEX non deve preoccuparsi di questioni "estetiche" come avviene nei normali word processor, soprattutto quelli di tipo WYSIWYG[2], bensì è incoraggiato un approccio "logico" al testo in cui l'utente specifica le relazioni di tipo logico che intercorrono fra le varie parti che compongono il suo documento.

I vantaggi di questo approccio sono evidenti[3]: l'utente è libero di concentrarsi solo sul contenuto del testo e non sul suo aspetto finale, inoltre l'approccio di tipo

[1] Questo in realtà è un comando LATEX ma, per l'esempio in questione, la cosa non è rilevante.

[2] WYSIWYG è l'acronimo di "What You See Is What You Get" che significa che quello che compare a video durante la stesura di un testo sarà pure l'aspetto finale del documento.

[3] Per una trattazione sul rapporto fra questo tipo di approccio e l'accessibilità dei documenti da parte di non vedenti vedi [1].

logico spinge a pensare in termini di unità logiche che compongono il testo portando indirettamente ad un documento più ordinato e comprensibile.

Inoltre TEX, nel generare il documento finale, utilizza le stesse regole che usano i tipografi umani nella composizione dei testi, regole che l'utente di TEX può ignorare e che generalmente portano a documenti esteticamente molto migliori di quelli che una persona che non sia un tipografo di professione può creare.

Merita un accenno il concetto di "stile" prima menzionato: quando TEX genera il documento finale ubbidisce ad alcune regole contenute in alcuni file di configurazione, file che di solito sono scritti non dall'utente di TEX ma dall'editore del documento. Così l'utente specifica dove iniziano i vari capitoli ma è la persona che fornisce i file con gli stili a decidere come i capitoli vanno formattati nel documento finale: nel caso di un articolo da pubblicare in una rivista i capitoli saranno stampati, ad esempio, l'uno di seguito all'altro con un piccolo titolo in grassetto mentre, nel caso di un libro, i capitoli inizieranno in una nuova pagina, magari sempre sulla facciata destra del libro, scritti con grossi caratteri e preceduti da un numero. Questo tipo di approccio fornisce l'ovvio vantaggio di una flessibilità impensabile con i word processor WYSIWYG.

Come ultimo punto va sottolineato il fatto che il TEX si presta particolarmente bene all'utilizzo da parte di utenti non vedenti: il fatto che il testo venga scritto in ASCII e non contenga informazioni estetiche quali tipo di font, posizionamento dei caratteri, etc. che risultano ridondanti per un non vedente, rende la stesura dei testi facilmente gestibile da un utente disabile munito di una postazione di lavoro opportuna (con screen reader o barra braille). Contemporaneamente, la possibilità di controllare tutti gli elementi stilistici mediante comandi contenuti nel testo ASCII rende l'utente in grado di produrre testi graficamente pregevoli.

L'unico piccolo prezzo da pagare per tutta questa serie di benefici è che l'utente deve imparare i comandi TEX per poter scrivere un testo. Inizialmente i comandi TEX implementati da Knuth (il cosiddetto "plain TEX") erano piuttosto ostici ma molte persone hanno scritto delle librerie di "macro" ossia hanno creato comandi di alto livello che, basandosi sui comandi TEX, facilitano il lavoro dell'utente. Due librerie di questo tipo fra le più diffuse sono $\mathcal{AMS}$-TEX: un package creato dalla American Mathematical Society per la stesura di articoli ad argomento matematico, e LATEX (con la sua nuova versione LATEX 2_ε): un gruppo di macro più generiche pensate per la stesura di una grande varietà di testi (questa tesi è stata scritta usando LATEX 2_ε).

Non è scopo del presente lavoro fornire una trattazione approfondita sul TEX e sui package come LATEX, per una trattazione di questo tipo si rimanda a [9], [10], [7] e [14].

In seguito quando si parlerà di TEX ci si riferirà implicitamente anche a LATEX, $\mathcal{AMS}$-TEX e package simili che, come vedremo, possono lavorare con TESI. Per indicare il TEX nella sua forma base (così come è stato creato da Knuth) useremo esplicitamente il termine "plain TEX".

Cos'è TESI e quali sono i suoi obiettivi

TESI è un acronimo per "TEX to Speech Interface": si tratta di un programma che riceve in input un testo TEX e che ne esamina le formule matematiche presenti costruendone l'albero sintattico[4] e permettendo la visita del medesimo da parte di un utente non vedente mediante sintesi vocale.

L'obiettivo di questo lavoro è quello di fornire un mezzo economico per permettere a non vedenti di manipolare espressioni matematiche, gli strumenti in questo campo

[4] Vedi par. 3.1.3 a pag. 28.

sono infatti tutt'ora piuttosto carenti. Quello che manca è un mezzo che fornisca all'utente non vedente tutta quella serie di informazioni implicite contenute nelle formule matematiche presentate in forma stampata.

Consideriamo la seguente formula matematica:

$$\frac{\displaystyle\sum_{i=1}^{\infty} \frac{1}{(\sin 2\vartheta)^i}}{\displaystyle\sum_{i=1}^{\infty} \frac{1}{q^i}} = \frac{1-q}{1-\sin 2\vartheta}$$

una simile espressione, proprio per la veste tipografica con cui si presenta, contiene una serie di informazioni implicite che mettono subito in rilievo che essa è costituita da due termini uniti in un'uguaglianza, che ognuno dei due termini è una frazione, il riconoscimento di numeratore e denominatore di ogni frazione avviene automaticamente, quasi a livello inconscio, e solo in una fase successiva il lettore ne esamina l'esatta natura. Per rendersi conto di questo basti osservare la precedente espressione in questa forma:

$$(\sum_{i=1}^{\infty}(1/(\sin 2\vartheta)^i))/(\sum_{i=1}^{\infty}(1/q^i)) = (1-q)/(1-\sin 2\vartheta)$$

il significato matematico è lo stesso ma la difficoltà di comprensione è aumentata di molto.

Purtroppo questa è la situazione in cui si trova comunemente ad operare l'utente non vedente: i vari ausili quali sintetizzatori vocali o barre braille sono infatti incapaci di fornire il "colpo d'occhio" che dà una formula presentata in una veste tipografica attraente, si unisca a questo il fatto che una simile espressione, scritta in TEX appare come:

```
$$
{
    \sum_{i=1}^\infty {1 \over (\sin 2\vartheta)^i} \over
    \sum_{i=1}^\infty {1 \over q^i}
} = { 1 - q \over 1 - \sin 2\vartheta }
$$
```

ossia le informazioni spaziali sono completamente rimosse, e questo proprio per la natura del TEX che, come abbiamo visto, vuole sgravare l'utente dal preoccuparsi di informazioni estetiche quali la sistemazione tipografica della formula nel testo finale.

TESI si propone appunto di supplire queste informazioni "spaziali" della formula costruendo una struttura dati che ne rifletta le relazioni gerarchiche fra i termini (l'albero sintattico sopra menzionato) e permettendo all'utente di viaggiare attraverso questa struttura dati.

Concetti base di TESI

In questa sezione definiremo alcuni concetti base necessari per spiegare il funzionamento di TESI. Questi concetti verranno approfonditi in seguito.

Il funzionamento di TESI in breve

TESI basa il suo funzionamento su file di configurazione: questi sono semplici file ASCII nei quali l'utente può specificare i comandi TEX che il programma deve riconoscere indicandone il significato dal punto di vista matematico nonché il modo in cui il comando va letto. Ad esempio, inserendo nel file di configurazione il comando:

```
Accento "\overline": "sopralineato";
```

l'utente informa TESI che il comando TEX `\overline` rappresenta un comando di accentazione e che, in fase di lettura della formula che contiene il comando, l'accento va letto "sopralineato". Così, se nel file TEX è presente il comando:

```
$$ \overline x $$
```

che genera l'output: "$\overline{x}$", in fase di esplorazione della formula TESI pronuncierà il testo: *≪x sopralineato≫*.

Dopo aver letto i file di configurazione, TESI passa a leggere il file TEX di input; quando incontra una formula matematica il programma ne costruisce l'albero sintattico basandosi sulla grammatica codificata al suo interno (una trattazione più tecnica su questo punto verrà fatta nel capitolo 5).

Brevemente possiamo dire che l'albero sintattico di una formula è una struttura dati che riflette le relazioni logiche fra le componenti della formula stessa, in questa struttura sono presenti vari "nodi" che rappresentano gli operatori presenti nella formula. Il termine "operatore" va inteso qui nel senso più ampio: possiamo definire un operatore come un elemento costitutivo della formula legato ad altri elementi (che chiameremo i suoi "operandi") da precise relazioni gerarchiche. Ad esempio nella formula:

$$x + \sin 2\alpha$$

il carattere + indica un operatore che ha come operandi rispettivamente "x" e "$\sin 2\alpha$". I nodi che rappresentano gli operatori sono legati all'interno dell'albero sintattico ai nodi che rappresentano i propri operandi, mediante i tasti cursore l'utente può spostarsi da un nodo ad un suo operando (il che corrisponde ad analizzare solo una parte dell'intera formula), tornare da un operando al suo operatore (ossia esplorare una parte più ampia della formula) o scorrere gli operandi di uno stesso operatore. A seconda della complessità della formula in esame un nodo può essere un operatore con uno o più operandi e, contemporaneamente, essere operando di un altro operatore. Nel nostro esempio il nodo corrispondente all'operatore "sin" è contemporaneamente operatore del nodo "2α" ma è pure operando del nodo "+".

Il programma permette dunque all'utente di esplorare gli alberi sintattici così costruiti mediante sintesi vocale. Continuando col nostro esempio TESI inizierà l'esplorazione della formula descrivendola interamente: *≪x più seno di 2 alfa≫*, se l'utente preme [↓] l'esplorazione passerà al primo operando del nodo "+" ossia verrà letto *≪x≫*, con il tasto [→] l'utente può esplorare gli altri operandi del nodo "+", in questo caso il termine "$\sin 2\alpha$", che verrà letto da TESI come *≪seno di 2 alfa≫*. A questo punto l'utente può scendere maggiormente in dettaglio esplorando i nodi "2" e "α", oppure tornare al livello superiore premendo [↑] .

Per attuare la sintesi vocale si è pensato di non interfacciarsi direttamente con un sintetizzatore vocale ma si è ricorso all'output di stringhe a video tramite il BIOS del computer. Gli elaboratori per non vedenti sono solitamente muniti di un sintetizzatore vocale e di un software opportuno (detto "screen reader") che intercetta l'output effettuato in questo modo e ne effettua la sintesi vocale. TESI ottiene così la più ampia compatibilità con le installazioni esistenti senza costringere l'utente all'acquisto di un hardware o software specifici. Come contropartita viene perso il controllo sulla prosodia(5) ma questo è un aspetto volutamente trascurato: lo scopo di questo lavoro

[5] In questo contesto per prosodia intendiamo tutta quella serie di informazioni collegate al parlato implicite nell'uso delle pause e dell'intonazione. In italiano per esempio l'intonazione di una frase è

era quello di legare la struttura di una formula matematica ad informazioni di tipo "spaziale" quali quelle contenute in un albero sintattico.

Lo schermo

Durante il suo funzionamento il programma divide lo schermo in quattro zone evidenziate con l'uso di diversi colori:

Riga di stato La prima riga dello schermo contiene alcune informazioni quali il nome del file in esame, il numero di formule presenti al suo interno e così via.

Finestra TEX Questa finestra occupa la maggior parte dello schermo ed al suo interno viene evidenziato il file TEX in esame.

Finestra Sintetizzatore Questa finestra situata in fondo allo schermo contiene il testo che verrà catturato dal software di sintesi vocale e pronunciato dal sintetizzatore. Inoltre qui vengono stampati, evidenziati con colori opportuni, messaggi di errore che possono essere utili per un istruttore vedente ma che, se letti, rallenterebbero troppo il lavoro di un utente non vedente.

Finestra di Input L'ultima riga dello schermo viene usata per l'input di dati (es. i nomi dei file da leggere).

Modalità di funzionamento

Durante il suo funzionamento il programma può trovarsi in cinque modalità diverse:

Input File In questa modalità il programma attende in finestra di input un nome di file dall'utente: questa è la prima modalità in cui entra il programma quando parte.

Esplora Questa è la modalità principale di funzionamento del programma in cui l'utente può eseguire la visita dell'albero sintattico di una formula. Mentre il pezzo di formula in esame viene letto dal sintetizzatore vocale, il testo corrispondente nel file TEX viene evidenziato nella finestra TEX usando colori opportuni: questa particolarità può costituire un valido aiuto per un istruttore vedente.

Storia Questa modalità è un caso particolare della modalità precedente: l'utente può spostarsi attraverso l'albero sintattico solo in due direzioni: verso la radice o verso il nodo corrente. Durante lo spostamento il programma fornisce alcune informazioni che permettono di comprendere chiaramente il percorso che porta dalla radice della formula al nodo corrente.

Questa modalità è pensata per aiutare l'utente a capire l'esatta posizione in cui si trova all'interno della formula.

TEX In questa modalità è possibile scorrere il file TEX nella finestra TEX; il programma non invia nulla al sintetizzatore ma si limita a spostare il cursore seguendo i comandi impartiti dall'utente. Sarà il particolare screen reader utilizzato a fornire all'utente le informazioni opportune.

Editor Questa modalità richiama un editor esterno per permettere la modifica del file TEX.

---

l'informazione prosodica che permette di distinguere una frase interrogativa da una affermativa. Per un esempio sull'importanza della prosodia nell'interpretazione di formule matematiche vedi [12].

Capitolo 1

Manuale per l'utente

Il presente capitolo spiega come installare e lanciare TeSI nonché i comandi disponibili nelle varie modalità di funzionamento del programma.

1.1 Installazione

TeSI è un programma scritto per PC-IBM e compatibili in configurazione essenziale: bastano un processore 8088 ed una scheda video CGA. È naturalmente richiesto che il computer su cui verrà fatto girare il programma sia dotato di uno screen reader, in caso contrario il programma funzionerà comunque, ma non sarà di estrema utilità.

L'installazione del programma richiede semplicemente che tutti i file forniti all'utente vengano copiati in una medesima directory. Affinché sia possibile lanciare TeSI da qualsiasi directory è necessario indicare dove si trova il programma coi comandi `PATH` ed `APPEND` nel file `AUTOEXEC.BAT`. Rinviamo al manuale di MS-DOS per la spiegazione dell'utilizzo di questi comandi.

Con TeSI deve essere distribuito il file `LICENSE.TXT` che contiene i termini per l'utilizzo, la distribuzione e la copia del programma in conformità alla GNU general public license.

1.2 Riga di comando

Per lanciare TeSI basta usare il comando:

```
TESI
```

al prompt del DOS.

È possibile specificare il nome del file TeX da esaminare indicandolo dopo il nome del programma; in caso contrario TeSI, appena lanciato, entrerà in modalità Input File e chiederà all'utente il nome del file di input.

Se l'utente lo desidera può indicare uno o più file di configurazione per TeSI a patto di precederli con lo switch `-C`, ad esempio il comando:

```
TESI -CUTENTE.CFG
```

indica a TeSI di utilizzare il file `UTENTE.CFG` come file di configurazione (notare che non va inserito uno spazio fra `-C` ed il nome del file di configurazione), i file verranno letti nell'ordine in cui sono scritti nella linea di comando. Nel capitolo 2 viene svolta una trattazione più approfondita sull'utilizzo dei file di configurazione.

Infine se si specifica l'opzione `-L` il programma creerà il file `ERROR.LOG` nel quale verranno salvati messaggi di errore e di debug del programma, questa opzione serve essenzialmente al sottoscritto per correggere errori di codifica.

1.3 Modalità Input File

In questa modalità l'utente deve fornire il nome di un file che verrà analizzato da TESI.

Premendo i tasti [↑] o [↓] è possibile scorrere i file TEX presenti nella directory attuale, quando viene trovato il file che si vuole esplorare lo si può selezionare semplicemente premendo ›**Enter**.

Per cambiare directory basta fornire il nome della directory in cui ci si vuol spostare e battere ›**Enter**. Se l'utente inserisce un nome file contenente i caratteri `*` o `?` il programma userà questo nome come nuova maschera per il file di input, ad esempio inserendo:

```
\DATI\*.TXT
```

il programma si sposterà nella directory `\DATI` e permetterà di scorrere con le frecce tutti i file con estensione `.TXT`.

Naturalmente è possibile inserire direttamente il nome di un file senza usare i tasti cursore semplicemente scrivendolo, eventualmente preceduto dal path.

Se il file specificato non esiste il programma ci permetterà di crearlo entrando in modalità editor o di specificare un altro nome di file.

Se viene inserito un nome di file vuoto o se si preme **Esc** il programma termina.

1.4 Modalità Esplora

Questa è la modalità principale in cui lavora TESI. Mentre si trova in questa modalità il programma permette all'utente di selezionare una particolare formula contenuta nel file TEX e di navigare attraverso il suo albero sintattico.

Quando deve sintetizzare una formula TESI ne considera la sua complessità (che corrisponde all'incirca al numero di termini elementari che la compongono), se tale complessità supera un certo valore, impostato dall'utente, TESI eviterà di sintetizzarne alcune parti sostituendole con la dicitura: ≪*espressione complessa*≫ o qualcosa di analogo. Questo permette al programma di fornire all'utente una visione globale della struttura di formule complesse prima di analizzarne le singole componenti.

Mentre si trova in questa modalità il programma non solo effettua la sintesi vocale del nodo in esame ma si preoccupa di evidenziarne il testo all'interno del file TEX segnandolo in un diverso colore nella finestra TEX, questa funzionalità è stata pensata per facilitare il compito ad un istruttore vedente.

Ecco una descrizione dei comandi riconosciuti dal programma in modalità Esplora:

[↓] Passa dall'esame del nodo corrente all'esame dei suoi operandi.

[↑] Passa dall'esame del nodo corrente a quello del nodo padre (l'operatore di cui il nodo corrente è operando).

[ ] , [→] Passa dal nodo corrente agli altri nodi che si trovano al suo stesso livello e che stanno alla sua destra od alla sua sinistra. In pratica permette di scorrere gli operandi di uno stesso operatore.

Eventuali operatori compresi fra gli operandi vengono letti solo spostandosi da sinistra verso destra.

Ctrl + ☐ , **Ctrl** + [→] Come sopra ma questa volta gli operandi fra i nodi vengono sempre letti.

Ctrl + [↑] Passa ad esplorare l'apice[1] del nodo corrente.

Per tornare dall'apice di un nodo al nodo stesso bisogna usare [↑] o **Home**. La differenza è che il tasto **Home** funziona anche quando ci si trova molto all'interno della formula che sta come apice mentre [↑] funziona solo se ci si trova alla radice dell'apice.

Ctrl + [↓] Come il precedente ma per il pedice di un nodo.

Home Torna al nodo di cui il nodo corrente è apice o pedice.

Page Up, Page Down Visita la formula precedente o seguente nel file TEX.

Ctrl + Page Up, Ctrl + Page Down Visita rispettivamente la prima o l'ultima formula nel file TEX.

V, v Chiede di specificare il numero d'ordine della formula da visitare.

C, c Permette di cambiare la soglia di complessità oltre la quale una formula non viene letta per esteso.

F, f Forza la lettura completa del nodo corrente indipendentemente dalla sua complessità.

S, s Entra in modalità Storia descritta in seguito.

E, e Entra in modalità Editor descritta in seguito. Il cursore viene posizionato all'inizio del testo TEX relativo al nodo corrente.

Tab Entra in modalità TEX descritta in seguito.

Esc Entra in modalità Input File.

F1 Mostra un breve testo di aiuto che riassume questi comandi.

1.5 Modalità Storia

Questa modalità serve per comprendere più chiaramente la posizione del nodo che stiamo esaminando all'interno di una formula.

Ecco un elenco dei comandi disponibili in questa modalità:

[↑] Avvicina alla radice dell'albero sintattico passando al nodo padre del nodo corrente.

[1] In seguito con "apice" e "pedice" si intenderà del testo che compare rispettivamente sopra o sotto il livello usuale della riga, questi sono termini usati nel linguaggio della composizione tipografica e, nelle formule matematiche, coincidono praticamente coi termini "esponente" (come in x^2) ed "indice" (come in x_2).

[↓] Allontana dalla radice dell'albero sintattico muovendosi verso il nodo che si stava esaminando nella modalità Esplora.

TESI fornisce la posizione del nuovo nodo rispetto al nodo corrente con un messaggio del tipo: «*Operando 1 di 3*».

Esc Ritorna in modalità Esplora.

F1 Mostra un breve testo di aiuto che riassume questi comandi.

1.6 Modalità TEX

In questa modalità si scorre il file TEX nella finestra TEX, TESI non invia nulla al sintetizzatore ma si limita a spostare il cursore seguendo i comandi dell'utente. L'ammontare di informazioni che arrivano all'utente dipendono dal software utilizzato come screen reader.

I comandi disponibili in questa modalità sono molto simili a quelli utilizzati dal programma TAB[(2)].

[ ] , [↑] , [↓] , [→] I tasti cursore permettono di spostarsi all'interno del file TEX un carattere alla volta.

Page Up, Page Down Sposta il cursore verticalmente di una schermata alla volta.

Home, End Sposta il cursore rispettivamente all'inizio od alla fine della riga corrente.

Ctrl + Home, Ctrl + End Sposta il cursore rispettivamente all'inizio od alla fine del file TEX.

V, v Permette di specificare la linea alla quale spostare il cursore.

E, e Entra in modalità Editor descritta in seguito.

Tab, Esc Ritorna in modalità Esplora riposizionandosi sul nodo che si stava esaminando prima di entrare in modalità TEX.

Enter Ritorna in modalità Esplora ma si posiziona sulla formula dove attualmente si trova il cursore (o sulla formula che si stava visitando precedentemente se il cursore non si trova attualmente su una formula).

F1 Mostra un breve testo di aiuto che riassume questi comandi.

1.7 Modalità Editor

Questa modalità permette all'utente di chiamare un editor esterno per modificare il file TEX in esame.

Nel file di configurazione è possibile specificare quali comandi TESI dovrà eseguire per richiamare l'editor[(3)].

[2] TAB è un semplice editor di testi orientato all'utente non vedente e scritto dal Prof. Giuliano Artico, relatore di questa tesi di laurea. Il programma viene distribuito gratuitamente.

[3] Vedi il comando `EditCommand` a pag. 17 per maggiori informazioni.

Capitolo 2

File di configurazione

Il file di configurazione permette all'utente di istruire TeSI sui comandi TeX da riconoscere: ogni comando TeX deve venire definito con opportuni comandi nel file di configurazione; inoltre l'utente deve indicare una stringa che rappresenterà il testo letto dal sintetizzatore vocale quando il comando TeX viene incontrato in una formula.

Nel file di configurazione è anche possibile impostare alcune opzioni che permettono di adattare il funzionamento di TeSI alle proprie esigenze: parametri quali il modo in cui il programma segnala le condizioni di errore, ad esempio, possono essere impostati in questo modo.

Al momento del suo avvio TeSI legge il file `DEFAULT.CFG`, è possibile specificare altri file da leggere mediante una direttiva di inclusione file all'interno di `DEFAULT.CFG`. Assieme al programma sono forniti i file `TEXPLAIN.CFG` e `LATEX209.CFG` che definiscono i comandi TeX e LaTeX che vengono anch'essi letti all'avvio del programma.

Un altro modo di indicare a TeSI quali sono i file di configurazione da leggere è quello di specificarli nella linea di comando preceduti dall'opzione `-C`, ad esempio per indicare a TeSI di leggere il file `UTENTE.CFG` come file di configurazione si può usare il comando:

```
TESI -CUTENTE.CFG
```

Nel caso un comando venga ridefinito (perché viene definito in due modi diversi in due file di configurazione diversi) la seconda definizione ha sempre la precedenza: in questo modo è sempre possibile per l'utente ridefinire i comandi TeX secondo le proprie esigenze.

La procedura consigliata per personalizzare il programma è dunque la seguente: l'utente può specificare nuovi comandi (o modifiche di quelli esistenti) in un proprio file di configurazione, poi indica a TeSI il nome del file con l'opzione `-C`. In alternativa all'interno del file `DEFAULT.CFG` si può specificare un comando di inclusione per il file preparato: tale comando di inclusione deve seguire i comandi di inclusione già presenti per permettere alle definizioni dell'utente di sovrascrivere quelle fornite nei file di default. L'utente può poi personalizzare il programma settando le opzioni di funzionamento desiderate in coda al file `DEFAULT.CFG`.

La definizione dei comandi TeX comporta anche alcune valenze sintattiche: se definiamo il termine "+" come un operatore che indica la somma, esso potrà apparire all'interno delle formule in un preciso contesto, ossia fra due operandi $(x + y)$ o di fronte ad un singolo operando $(+12)$ ma non dopo un'espressione $(x+)$. Il modo con cui TeSI interpreta le espressioni è codificato all'interno del programma e non può essere alterato. In altri termini possiamo, mediante i file di configurazione, specificare

che "+" è un operatore che indica la somma ma il modo in cui è permesso agli operatori somma di comparire all'interno delle formule matematiche dipende dalla grammatica[1] definita nel programma[2]. A questo punto dovrebbe essere chiaro che quando definiamo un nuovo comando dobbiamo considerarne la valenza sintattica e non semantica: se esiste un operatore che ha la stessa sintassi e precedenza dell'operatore "+" bisognerebbe definirlo con lo stesso comando usato per definire l'operatore "+" nel file di configurazione, anche se il nuovo operatore non indica una somma.

Passiamo ora a descrivere i comandi disponibili nel file di configurazione, nell'appendice A sono contenuti i file di configurazione distribuiti col programma.

2.1 Convenzioni tipograche

Per specificare la sintassi dei comandi da utilizzare nei file di configurazione utilizzeremo le seguenti convenzioni:

- `Comandi` — In questo modo indichiamo una stringa che deve apparire così com'è nel file di configurazione.
- *StringaN* — Una stringa racchiusa fra doppi apici: `"..."`, la stringa non può contenere dei caratteri newline ed ovviamente non può contenere dei doppi apici.

  Stringhe di questo tipo indicano del testo che va letto dal sintetizzatore, il testo verrà stampato a video così come appare all'interno della stringa, il risultato della sintesi vocale dipenderà da come lo screen reader cattura ed interpreta il testo mandato a video.
- *ComandoTEX* — Una stringa analoga alla precedente che però identifica un comando TEX.

  Per le stringhe di questo tipo si deve usare una sintassi analoga a quella che il TEX usa per i suoi comandi. In breve si può dire che un comando TEX è formato dal carattere "\" seguito da una o più lettere (lettere maiuscole e minuscole sono considerate diverse dal TEX) oppure da "\" seguito da un singolo carattere che non sia una lettera, caratteri singoli possono comparire all'interno di stringhe di questo tipo (ad esempio per definire operatori come "+").
- *n* – Un numero intero.
- `[ ... ]` — Il testo fra quadre indica una parte opzionale di un comando che può essere omessa.

2.2 Formato delle linee di comando

Nel file di configurazione è possibile inserire dei commenti facendoli precedere da `%`, tutto il testo che segue il `%` fino alla fine della riga verrà ignorato da TESI.

Non viene fatta distinzione fra lettere maiuscole o minuscole nei comandi del file di configurazione (ma tale distinzione vale per le stringhe specificate fra apici), in seguito si distingueranno maiuscole e minuscole solo a scopo di chiarezza.

[1] Vedi il par.3.1.3 per una definizione di grammatica.
[2] Vedi par. 5.3.2.

2.3 Inclusione di le

`Input` *NomeFile*
Questo comando permette di includere un file di configurazione in un altro, come abbiamo visto questo è un meccanismo che permette all'utente di specificare i propri file di configurazione di TESI.

NomeFile specifica il nome del file da includere, deve essere un nome di file DOS valido non incluso in doppi apici.

Al contrario degli altri comandi presenti nel file di configurazione questo non termina con un "`;`".

2.4 Personalizzazione di TESI

`EditCommand` *ComandoDOS* `;`
Definisce il comando per invocare l'editor esterno.

ComandoDOS è una stringa fra doppi apici che verrà usata come comando DOS per chiamare l'editor usato in modalità Edit, al suo interno è possibile usare i seguenti comandi:

`#f` — Questa stringa verrà sostituita dal nome del file in esame prima di passare *ComandoDOS* al DOS.

`#r` — Questo comando verrà sostituito dalla riga in cui si trova il cursore, se si chiama l'editor dalla modalità TEX, o dalla riga dove inizia la formula in esame se si chiama l'editor dalla modalità Esplora.

`#c` — Come il comando precedente ma per la colonna ove si è posizionati.

Ad esempio il comando:

```
EditCommand "Q.EXE -n#r #f";
```

farà sì che TESI usi Qedit come editor passandogli il nome del file attualmente in esame e la riga su cui si trova il cursore, informazione che Qedit userà, grazie allo switch `-n`, per posizionare il cursore a quella riga quando l'editor viene lanciato.

`MaxComplexity` *n* `;`
Definisce la massima complessità per una formula affinché questa venga letta per intero da TESI.

Se la complessità di una formula (che coincide all'incirca col numero di termini elementari che la compongono) supera il valore specificato da *n* allora TESI ne leggerà solo una parte "mascherando" il resto della formula con una dicitura del tipo: ≪*espressione complessa*≫.

Quando ci si trova in modalità Esplora è possibile cambiare questo parametro premendo **C**.

`PausaBreve` *Stringa1* `;`
`PausaLunga` *Stringa1* `;`
Quando sintetizza una frase TESI inserisce delle pause in punti opportuni per aumentare la chiarezza del parlato, questo è l'unico tipo di prosodia attuata dal programma e,

visto che il controllo della prosodia non è fra i nostri scopi, si tratta di una funzionalità piuttosto rudimentale.

Il programma utilizza due tipi di pause: una breve ed una lunga, con questi comandi è possibile specificare quali stringhe stampare a video in corrispondenza di una pausa breve e di una lunga.

Per il modo con cui TESI si interfaccia al sintetizzatore vocale non è purtroppo possibile specificare caratteri di controllo per comandare direttamente il sintetizzatore, così solitamente si usano i comandi:

```
PausaBreve ","; PausaLunga ".";
```

che specificano una virgola ed un punto rispettivamente per la pausa breve e quella lunga.

Alcuni screen reader possono avere difficoltà nell'interpretare stringhe con punteggiatura se non viene selezionata una opportuna modalità di funzionamento e può essere fastidioso dover cambiare continuamente fra diverse modalità dello screen reader mentre si usa TESI, per questo, in condizioni di default, le pause sono disabilitate coi comandi:

```
PausaBreve ""; PausaLunga "";
```

`ErrorBell ;`
`NoErrorBell ;`
Durante l'utilizzo di TESI l'utente può incorrere in una serie di errori banali, ad esempio la pressione di un tasto non riconosciuto dal programma. In simili situazioni, se è specificato il comando `ErrorBell`, TESI emetterà un breve suono e stamperà un messaggio di errore evidenziato con colori opportuni nella finestra sintetizzatore, senza però pronunciarlo. In questo modo l'utente non vedente non è rallentato dalla sintesi di stringhe ridondanti mentre un eventuale istruttore vedente riceve comunque utili informazioni sulla condizione di errore. Se si specifica invece il comando `NoErrorBell` il messaggio di errore verrà letto dal sintetizzatore.

`LeftDiscrim ;`
`NoLeftDiscrim ;`
Alcuni operatori vengono letti diversamente a seconda che si attraversino i loro operandi da destra a sinistra o viceversa quando viene specificato il comando `LeftDiscrim`.

Ad esempio nella formula $x > y$ se ci troviamo sul nodo "x" e ci spostiamo sul nodo "y" TESI pronuncierà: $\ll$*maggiore y*$\gg$, tornando al nodo "x" il programma leggerà $\ll$*minore x*$\gg$ (se è specificato `LeftDiscrim`).

Il comando `NoLeftDiscrim` annulla invece questa diversità di lettura dell'operatore.

Questa opzione può essere indispensabile in alcuni casi, pensiamo ad un eventuale operatore `\pot` che indichi un elevamento a potenza, la formula:

```
$$ 2 \pot x $$
```

andrebbe letta $\ll$*due elevato a x*$\gg$ spostandosi da sinistra a destra oppure $\ll$*x con base 2*$\gg$ se letta in senso inverso.

`ReadAllFormula ;`
`NoReadAllFormula ;`
TEX distingue fra due diverse modalità in cui vengono scritte le formule matematiche: nel "TEX display math mode" una formula viene scritta da sola centrandola su una

riga, come nel caso seguente:

$$\sum_{i=0}^{\infty} \frac{1}{q} = \frac{1}{1-q}$$

per specificare questa modalità in plain TEX si racchiude la formula fra `$$...$$`. Nel "TEX math mode" invece la formula viene inserita all'interno della riga di testo corrente, come in: $\sum_{i=0}^{\infty} \frac{1}{q} = \frac{1}{1-q}$. Per specificare questa modalità in plain TEX si deve racchiudere la formula fra `$...$`

Come si vede il TEX usa criteri diversi per stampare le formule nei due modi, si faccia caso ad esempio alla posizione in cui sono stampati i limiti della sommatoria o alla dimensione dei caratteri usati. Il display mode viene solitamente usato per formule di complessità medio alta mentre il math mode si usa per includere semplici espressioni da inserire nel testo come: x, $f(x)$ o simili.

In condizioni di default (o se si usa il comando `NoReadAllFormula`) TESI si occupa solo delle formule in display mode, se si usa il comando `ReadAllFormula` TESI leggerà anche le formule in math mode.

`SynthPause` *n* `;`
Specifica una pausa di *n* millisecondi dopo aver eseguito l'output di una stringa attraverso il BIOS. Questo comando può rivelarsi utile con alcuni screen reader i quali, se il cursore non resta fermo in un punto per un dato tempo, non eseguono la sintesi di quanto stampato a video.

2.5 Inizio e ne delle formule

`InizioMathMode` *ComandoTEX* `;`
`FineMathMode` *ComandoTEX* `;`
Con questi comandi è possibile informare TESI che *ComandoTEX* rappresenta rispettivamente un comando TEX di inizio o fine del "TEX math mode".

`InizioDisplayMathMode` *ComandoTEX* `;`
`FineDisplayMathMode` *ComandoTEX* `;`
Come i comandi precedenti ma per il "TEX display math mode".

Nel file `LATEX209.TEX` ad esempio vengono usati i comandi:

```
InizioDisplayMathMode "\["; FineDisplayMathMode "\]";
InizioMathMode "\("; FineMathMode "\)";
```

per riconoscere le combinazioni `\(...\)` e `\[...\]` che il LATEX usa per marcare blocchi di testo contenenti formule matematiche.

I comandi TEX `$$` e `$` sono già codificati all'interno di TESI e non hanno bisogno di essere definiti nei file di configurazione.

2.6 Termini elementari

In questa sezione ci occuperemo dei componenti più basilari delle formule matematiche: numeri, lettere e simboli speciali.

Il modo con cui TeSI riconosce i numeri non è definibile dall'utente ma è codificato all'interno del programma. Brevemente possiamo dire che TeSI riconosce i numeri interi come $1, 2, 123$, riconosce numeri reali quali $1.2, 0.15, .28$ ed infine riconosce quantità espressa nella "notazione scientifica" ossia con mantissa ed esponente come $1.12E+15$ fintanto che non appaiono nel testo comandi di selezione di font del tipo: `$ 1.12{\rm E}+15 $`.

`Garbage` *ComandoTEX* `;`
ComandoTEX verrà ignorato da TeSI.

Questo costrutto permette di ignorare comandi di formattazione del testo che appaiono all'interno di formule matematiche ma che non hanno relazione con le stesse.

`Testo` *ComandoTEX* `;`
ComandoTEX indica un comando TeX che è seguito da un blocco di testo (ad esempio `\mbox`).

`Lettera` *ComandoTEX* `:` *Stringa1* `;`
ComandoTEX rappresenta un singolo termine letterale, *Stringa1* rappresenta la stringa corrispondente da leggere.

Con "termine letterale" si intende una singola lettera che compare nelle formule (come x, y, etc.), questo termine ha un senso diverso dal nome di una funzione: se specifichiamo che la lettera f rappresenta un termine di questo tipo l'espressione $f(x)$ verrà interpretata come una sequenza di due termini: f e (x) legati da una moltiplicazione implicita. Questo comportamento nella maggior parte dei casi non dà problemi dato che la lettura fornita da TeSI: *«f aperta tonda x chiusa tonda»* permette comunque all'utente di capire il senso dell'espressione, anzi questo modo di trattare le formule si rivela in molti casi provvidenziale.

Consideriamo le espressioni:

$$x = F(y, z) \qquad A \subset F \qquad F = (x_1, y_1) \qquad 1 + 2F$$

in ognuna di esse F compare con un significato diverso: rispettivamente indica una funzione, un insieme, un punto ed una variabile. TeSI non ha nessun modo di riconoscere il diverso uso della scrittura F nelle varie formule quindi è preferibile dare ad F il generico significato di "termine letterale" e lasciare all'utente il compito di capire cosa il termine F rappresenti realmente.

Ovviamente questo modo di trattare i nomi di funzione potrebbe causare qualche errore di interpretazione delle espressioni quindi, se la lettera f compare sempre col significato di nome di funzione, è preferibile definirla non col comando `Lettera` ma col comando `Funzione` descritto più avanti. Ciò assicura che l'espressione $f(x)$ venga interpretata come una funzione con argomento x, di contro f non potrà più comparire nelle formule con significato di termine letterale e l'espressione $f + 1$ genererebbe un errore sintattico.

Il file `TEXPLAIN.CFG` usa il comando `Lettera` per definire tutte le lettere dell'alfabeto inglese, maiuscole e minuscole, ad eccezione di f, g ed h che sono definite come funzioni; le lettere greche sono pure definite usando questo comando.

`Simbolo` *ComandoTEX* `:` *Stringa1* `;`
Analogo al precedente ma definisce un comando TeX che esprime un simbolo matematico del tipo "∞", "$\Re$", etc.

2.7 Apici, pedici ed accenti

`Apice` *ComandoTEX* ;
`Pedice` *ComandoTEX* ;
ComandoTEX indica un comando TEX che definisce rispettivamente un apice od un pedice.

Nel file di configurazione `TEXPLAIN.CFG` vengono usati i comandi

```
Apice "^"; Apice "\sp";
Pedice "_"; edice "\sb";
```

che definiscono i comandi TEX di default per specificare apici e pedici.

`Accento` *ComandoTEX* : *Stringa1* ;
Usato per definire i comandi di accentazione.

I file di configurazione standard definiscono praticamente tutti i tipi di accenti matematici usati.

Questo comando viene usato per definire anche i comandi `\overline` ed `\underline` che hanno una sintassi analoga a quella degli accenti.

2.8 Operatori

Esistono vari comandi che definiscono operatori, visto che tutti hanno la medesima sintassi ne descriveremo solo uno, poi forniremo una tabella che elenca i comandi che definiscono operatori descrivendo nel contempo le caratteristiche degli operatori stessi (priorità, associatività, ecc.).

`OperatoreSomma` *ComandoTEX* : *Stringa1* `[, toleft` *Stringa2*`]` `[,` *Stringa3*`]` ;
Questo comando definisce *TEXcommand* come un operatore analogo a "+".

Stringa1 dice come leggere l'operatore.

Stringa2 indica come leggere l'operatore quando ci si sposta da destra a sinistra e l'opzione `LeftDiscrim` (descritta sopra) è attivata, se questa parte del comando è assente l'operatore verrà sempre letto come *Stringa1*.

Stringa3 invece indica come leggere l'operatore nel caso di una formula complessa.

La tabella 2.1 descrive i vari comandi che definiscono operatori ed il tipo di operatori definiti da questi comandi.

A titolo di esempio riportiamo le seguenti definizioni di default per gli operatori "+" e ">":

```
OperatoreSomma "+": "più", "somma complessa";

OperatoreRelazione ">":
    "maggiore", toleft "minore", "diseguaglianza complessa";
```

Comandi nel file di configurazione	Operatori definiti		
	Sintassi	Associatività	Esempi
`Separatore`	Infisso	A sinistra	, |
`Relazione`	Infisso	A sinistra	$> \leq$
`OperatoreSomma`	Infisso	A sinistra	$+ -$
`OperatoreOR`	Infisso	A sinistra	$\vee$
`OperatoreProdotto`	Infisso	A sinistra	$* /$
`OperatoreAND`	Infisso	A sinistra	$\wedge$
`OperatoreNOT`	Prefisso	A destra	~
`OperatoreSomma`	Prefisso	A sinistra	$-$ (unario)
`OperatoreFattoriale`	Postfisso	A sinistra	!
`OperatoreEsponente`	Infisso	A destra	^

Tabella 2.1: Comandi che definiscono operatori. Gli operatori sono elencati in ordine di precedenza crescente dall'alto al basso.

2.9 Parentesi

`ParentesiAperta` *ComandoTEX* `:` *Stringa1* `;`
`ParentesiChiusa` *ComandoTEX* `:` *Stringa1* `;`
Questi comandi definiscono rispettivamente una parentesi aperta ed una chiusa.

Attualmente TESI si aspetta di trovare una parentesi chiusa accoppiata con ogni parentesi aperta ma non forza l'essatto "matching" delle parentesi così, ad esempio, le espressioni $(1 + 1)$ e $(1 + 2]$ sono entrambe riconosciute.

2.10 Funzioni, frazioni, sommatorie

`Funzione` *ComandoTEX* `:` *Stringa1* `;`
ComandoTEX indica il nome di una funzione che si legge come specificato in *Stringa1*.

Per una discussione sulla differenza fra lettere e funzioni si veda il comando `Lettera` descritto sopra.

`Radice` *ComandoTEX* `:` *Stringa1* `[,` *Stringa2*`]` `;`
Indica un comando analogo a `\sqrt`.

Stringa1 indica cosa leggere, *Stringa2* cosa leggere nel caso di un'espressione complessa.

`Frazione` *ComandoTEX* `;`
Usato per definire il costrutto `\over`.

`Atop` *ComandoTEX* `:` *Stringa1* `;`
Usato per definire il costrutto `\atop`. *Stringa1* indica cosa leggere quando si incontra *ComandoTEX*.

`OperatoreEsteso` *ComandoTEX* `:` *Stringa1* `[,` *Stringa2*`]` `;`
ComandoTEX indica un operatore tipo $\sum$ o $\prod$, anche comandi per integrali e limiti (che posseggono la stessa sintassi) rientrano in questa categoria.

Ancora una volta *Stringa1* indica cosa leggere e *Stringa2* cosa leggere nel caso di un'espressione complessa.

Capitolo 3

Linguaggi formali e compilatori

3.1 Un po' di denizioni

In questa sezione verrà introdotta la terminologia riguardante i linguaggi formali che sarà utilizzata in seguito per spiegare la struttura interna di TESI. Questa non vuole essere una trattazione rigorosa ed esaustiva dell'argomento, di per sé vastissimo: si rimanda pertanto a [2] e [8] per un eventuale approfondimento. Al solito il lettore già familiare con queste definizioni può saltare questa sezione.

3.1.1 Simboli, vocabolari e stringhe

Si definisce **simbolo** un'entità non ulteriormente scomponibile. Un simbolo è solitamente costituito da una sequenza di uno o più caratteri.

La totalità dei simboli considerati in un dato contesto viene detta **vocabolario**. Un vocabolario si indica con la sequenza non ordinata dei simboli che lo compongono all'interno di parentesi graffe, separati da virgole. Ad esempio: $V_1 = \{a, b, c\}$.

Una **stringa** è una sequenza ordinata di simboli solitamente indicata come:

$$s = s_1 s_2 \dots s_i \dots s_n$$

dove s indica la stringa formata dalla giustapposizione degli n simboli $s_1 \dots s_n$.

La **lunghezza** della stringa è il numero di simboli che la compongono e si indica con $|s|$: nell'esempio di cui sopra $|s| = n$.

Si definisce ϵ come la **stringa vuota**, da cui risulta $|\epsilon| = 0$.

Date due stringhe x ed y, si definisce la loro **concatenazione**, indicata come xy, la stringa formata dalla stringa x seguita dalla stringa y. Si definisce **potenza n-esima** della stringa s, e si indica con s^n, la stringa ottenuta concatenando n stringhe uguali ad s. Risulta $s^1 = s$ e, per definizione, si pone $s^0 = \epsilon$.

Si dirà che y è una **sottostringa** di x se vale: $x = wyz$ con w e z stringhe qualunqe, eventualmente vuote. In particolare y è **prefisso** di x se $x = yw$, y è **suffisso** di x se $x = wy$.

3.1.2 Linguaggi

Si definisce **linguaggio** un insieme di stringhe; si dirà che la stringa s **appartiene** al linguaggio L ($s \in L$) se e solo se s è un elemento di L.

Il linguaggio vuoto, ossia che non contiene nessuna stringa, si indica col simbolo Φ.

Dato un vocabolario V si indica con $L(V)$ il linguaggio che contiene tutte e sole le stringhe di lunghezza uno formate dai simboli appartenenti a V, $L(V)$ è chiamato **linguaggio associato** al vocabolario V.

Ai linguaggi si possono applicare le normali operazioni sugli insiemi, tipicamente unione, intersezione e differenza; inoltre possiamo definire la **concatenazione** di due linguaggi L_x e L_y come:

$$L_x L_y = \{xy \mid x \in L_x, y \in L_y\}$$

Analogamente a quanto fatto con le stringhe, possiamo definire la **potenza n-esima** del linguaggio L come il linguaggio dato dalla concatenazione di n linguaggi uguali ad L; vale $L^1 = L$ e $L^0 = \{\epsilon\} \neq \Phi$.

La **chiusura** del linguaggio L, indicata con L^*, è l'unione di tutte le potenze n-esime di L con $n \geq 0$:

$$L^* = \bigcup_{n=0}^{\infty} L^i$$

mentre la **chiusura positiva** di L (L^+) è l'unione di tutte le potenze n-esime di L per $n > 0$:

$$L^+ = \bigcup_{n=1}^{\infty} L^i$$

Si definisce **linguaggio universale** sul vocabolario V, indicato con V^*, la chiusura del linguaggio associato a V: $V^* = L(V)^*$, analogamente si definisce il **linguaggio universale positivo** come $V^+ = L(V)^+$.

3.1.3 Grammatiche

Le grammatiche sono un mezzo per definire formalmente un linguaggio in modo semplice, senza dover ricorrere alla enumerazione delle sue stringhe. Si tratta in pratica di un insieme di regole per generare tutte e sole le stringhe appartenenti ad un dato linguaggio.

Possiamo definire una **grammatica** G come una quadrupla (VT, VN, P, S) dove:

- VT è un vocabolario di simboli detti **simboli terminali**. Questi simboli formano le stringhe del linguaggio generato dalla grammatica G.
- VN è un vocabolario di simboli detti **simboli non terminali**. I simboli non terminali, pur non comparendo nelle stringhe del linguaggio generato da G, sono indispensabili per specificare il meccanismo generativo delle medesime.
- P è l'insieme delle **regole** della grammatica.
- S è il **simbolo iniziale** od **assioma** della grammatica, $S \in VN$.

I vocabolari VT e VN sono disgiunti; è utile definire la loro unione indicata con V: $V = VT \cup VN$.

Le grammatiche possono essere classificate in base alla forma assunta dalle regole contenute in P; queste ultime sono formate da una **parte destra** e da una **parte sinistra** separate dal simbolo "$\rightarrow$":

- **Grammatiche a struttura di frase** — La parte sinistra delle regole deve contenere almeno un simbolo non terminale: esse pertanto assumono la forma: $z \rightarrow w$ dove $z = aXb$ con $a, b, w \in V^*$ e $X \in VN$.

- **Grammatiche contestuali** – le regole possono assumere due forme equivalenti:
  1. $z \to w$ dove $z = aXb$ con $a, b \in V^*$, $X \in VN$ e $w \in V^+$. Inoltre dev'essere $|z| \leq |w|$.
  2. $uXw \to uxw$ dove $u, w \in V^*$, $X \in VN$ e $x \in V^+$.
- **Grammatiche non contestuali** — Ogni regola è nella forma: $X \to x$ dove $X \in VN$ e $x \in V^*$.
- **Grammatiche regolari** — Si dividono in:

  Grammatiche regolari destre — con regole nella forma: $X \to uY$ o $X \to u$ con $X, Y \in VN$ e $u \in VT^*$.

  Grammatiche regolari sinistre — con regole nella forma: $X \to Yu$ o $X \to u$ con $X, Y \in VN$ e $u \in VT^*$.

  Come si può vedere la parte destra di ogni regola contiene al più un simbolo non terminale.

  Si dicono **grammatiche regolari canoniche** quelle in cui u è un simbolo terminale o la stringa vuota.

Si può facilmente vedere che le grammatiche definite sono state date in ordine di generalità decrescente: ogni grammatica contiene quelle seguenti secondo lo schema in figura 3.1.

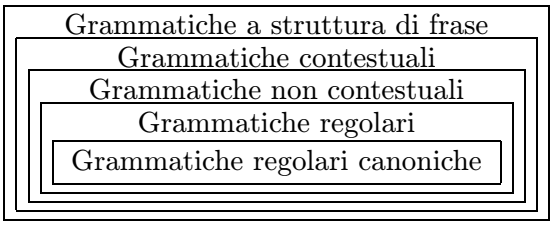

Figura 3.1: Gerarchia delle grammatiche

TESI basa il suo funzionamento su due grammatiche non contestuali: una per definire la sintassi dei comandi nel file di configurazione, l'altra per descrivere la sintassi delle espressioni matematiche scritte in TEX.

Il meccanismo con cui vengono generate le stringhe del linguaggio descritto da una grammatica G viene detto **derivazione**: in questo processo è possibile generare, a partire da una stringa contenente almeno un simbolo non terminale, una nuova stringa in cui il simbolo non terminale è sostituito dalla parte destra di una regola nella cui parte sinistra compare il simbolo non terminale in questione. Ad esempio: se $X \to cY$ è una regola della grammatica, a partire dalla stringa $abXba$ posso generare la stringa $abcYba$ dove X è stato sostituito dalla stringa cY.

Data una grammatica G e le stringhe $a = wXz$ e $b = wxz$, con $w, z, x \in V^*$ e $X \in VN$, si dirà che a **produce direttamente** b, o che b **deriva direttamente** da a se $X \to x$ è una regola di G, in tal caso si scriverà: $a \Rightarrow b$. Si dice invece che a **produce (indirettamente)** b o che b **deriva (indirettamente)** da a, e si indica con $a \Rightarrow^+ b$, se esiste una catena di derivazioni del tipo:

$$a \equiv x_0 \Rightarrow x_1 \Rightarrow x_2 \Rightarrow \ldots \Rightarrow x_n \equiv b \qquad \text{con } n > 0$$

La notazione $a \Rightarrow^* b$ indica che a produce b o che $a = b$.

Una grammatica G è **ricorsiva a sinistra** se permette derivazioni del tipo: $X \Rightarrow^+ Xw$, è **ricorsiva a destra** se ammette derivazioni del tipo: $X \Rightarrow^+ wX$. Inoltre si dirà che G possiede **autoinclusione** o **ricorsione interna** se ammette derivazioni di tipo: $X \Rightarrow^+ wXz$.

La stringa f costituisce una **forma di frase** per la grammatica G se e solo se $S \Rightarrow^* f$ con S simbolo iniziale di G. Una forma di frase formata da soli simboli terminali è detta **frase** di G.

A questo punto è facile definire il **linguaggio generato dalla grammatica** G, che indicheremo con L_G (o semplicemente L), come l'insieme delle frasi di G, più formalmente:

$$L_G = \{x \mid S \Rightarrow^* x, x \in VT^*\}$$

La gerarchia introdotta sulle grammatiche si riflette anche sui linguaggi da esse prodotti:

- **Linguaggi ricorsivamente enumerabili** — sono generati da grammatiche a struttura di frase.
- **Linguaggi contestuali** — sono generati da grammatiche contestuali.
- **Linguaggi non contestuali** — sono generati da grammatiche non contestuali.
- **Linguaggi regolari** — sono generati dalle grammatiche regolari.

naturalmente valgono le relazioni esemplificate nella figura 3.2.

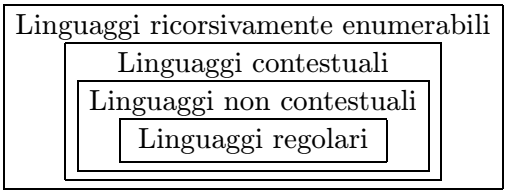

Figura 3.2: Gerarchia dei linguaggi formali.

Alberi sintattici

Una notazione molto comoda per evidenziare la struttura di una forma di frase appartenente ad una grammatica G è l'**albero sintattico**. Si tratta di un albero per cui valgono le seguenti proprietà:

1. Ogni nodo dell'albero contiene un simbolo in V.
2. Un nodo contenente il simbolo X ha come figli i nodi $x_1, x_2, \ldots, x_n$ ordinati da sinistra a destra, con $X \in VN$ e $x_i \in V^*$ se e solo se $X \to x_1 x_2 \ldots x_n$ è una regola di G.
3. La radice dell'albero e' l'assioma di G.

Come si può intuire, l'albero sintattico si ottiene facilmente durante il processo di derivazione di una forma di frase: basta cominciare dal simbolo iniziale, posto alla radice, ed aggiungere nuovi rami che uniscono un simbolo non terminale, che appare nella parte sinistra della regola utilizzata durante la derivazione, con nuovi nodi corrispondenti alla parte destra della medesima regola. Naturalmente la corrispondenza

fra derivazione ed albero sintattico non è biunivoca: per un albero sintattico possono esistere più derivazioni, equivalenti perché generano la medesima forma di frase, corrispondenti ad un diverso ordine di espansione dei nodi dell'albero.

Due derivazioni sono particolarmente importanti: la **derivazione canonica destra** e la **derivazione canonica sinistra** ottenute espandendo ad ogni passo il simbolo non terminale che compare rispettivamente più a destra o più a sinistra. Intutivamente questo equivale a costruire l'albero sintattico privilegiando di volta in volta il nodo più a destra o più a sinistra nella struttura.

Una frase si dice **ambigua** se ammette due alberi sintattici distinti per una data grammatica G. Una grammatica è ambigua se genera frasi ambigue. È stata dimostrata l'impossibilità di costruire un algoritmo che decida in tempo finito se una data grammatica è ambigua o meno; esistono comunque alcuni criteri che, se rispettati, permettono di costruire grammatiche non ambigue.

Nel seguito risulterà importante il concetto di **riduzione** di una stringa, corrispondente al processo inverso della sua derivazione. Anche in questo caso parleremo di **riduzione destra** e **riduzione sinistra** quali processi inversi rispettivamente della derivazione canonica destra o sinistra.

Grammatiche LR(0), LR(1) e LALR(1)

Un **analizzatore sintattico** o **parser** è un programma che, data una forma di frase di una grammatica G, ne ricostruisce l'albero sintattico. Esistono dei programmi in grado di generare automaticamente efficienti analizzatori sintattici per alcune classi di grammatiche, in questa sezione definiremo alcune di queste classi. Nel paragrafo 3.2.2 esamineremo Bison: un generatore di parser per le grammatiche appartenenti alla classe delle grammatiche LALR(1).

Per introdurre la categoria delle grammatiche LR(0) dobbiamo prima introdurre alcune definizioni: il simbolismo $X \Rightarrow_{rm} x$ indica che x si ottiene da X mediante un singolo passo di una derivazione canonica destra, inoltre si scriverà $X \Rightarrow^*_{rm} x$ per indicare che esiste una catena di derivazioni:

$$X \Rightarrow_{rm} x_1 \Rightarrow_{rm} x_2 \Rightarrow_{rm} \ldots \Rightarrow_{rm} x$$

ossia x si può derivare da X mediante derivazione canonica destra in uno o più passi.

Diremo che f è una **forma di frase destra** se $S \Rightarrow^*_{rm} f$ con S simbolo iniziale di una grammatica G.

A questo punto possiamo definire la **parte destra riducibile** di una forma di frase destra f come una sottostringa x tale che:

$$S \Rightarrow^*_{rm} uXw \Rightarrow_{rm} uxw \equiv f$$

In altri termini la parte destra riducibile di f è una sottostringa che può essere introdotta nell'ultimo passo di una derivazione canonica destra di f. Definiamo **prefisso LR riducibile** di una forma di frase destra f come un prefisso di f che possiede come proprio suffisso la parte destra riducibile di f; inoltre diciamo **prefisso LR** un qualsiasi prefisso di un prefisso LR riducibile.

Ad esempio, data la grammatica:

$$\begin{aligned} S' &\rightarrow Sc \\ S &\rightarrow SA \mid A \\ A &\rightarrow aSb \mid ab \end{aligned}$$

possiamo avere la seguente derivazione canonica destra:

$$S' \Rightarrow Sc \Rightarrow SAc \Rightarrow SaSbc$$

In questo caso $SaSbc$ è una forma di frase destra, la sua parte destra riducibile è formata dalla sottostringa aSb, il suo prefisso LR riducibile è dunque $SaSb$, i prefissi LR sono: ϵ, S, Sa e SaS.

Le **regole puntate** per una grammatica G sono regole del tipo: $X \rightarrow u.w$, dove il punto è un metasimbolo usato per spezzare la parte destra di una regola di modo che $uw = x$ e $X \rightarrow x$ è una regola della grammatica G, con $u, w, x \in V^*$. Se G contiene la regola $X \rightarrow \epsilon$ allora $X \rightarrow .$ è la regola puntata corrispondente.

Diremo che una regola puntata $X \rightarrow u.v$ è **candidata** per un prefisso LR y se:

$$S \Rightarrow_{rm}^* wXz \Rightarrow_{rm} wuvz \quad \text{e } wu = y$$

Una regola puntata si dice **completa** se è del tipo:

$$X \rightarrow x.$$

ossia il punto è il simbolo più a destra della regola.

A questo punto abbiamo un possibile meccanismo per effettuare la riduzione destra di una forma di frase $f = yw$: se $X \rightarrow x.$ è una regola puntata candidata per il prefisso LR y allora $X \rightarrow x$ potrebbe essere stata l'ultima regola utilizzata nella derivazione di f la cui forma precedente era del tipo uXw. Naturalmente questo è solo un ragionamento ipotetico in quanto la regola puntata $X \rightarrow x.$ potrebbe, ad esempio, essere valida per il prefisso LR y per via di una derivazione:

$$S \Rightarrow_{rm}^* zXv \Rightarrow_{rm} yv$$

diversa da quella che ha portato ad f, oppure potrebbero esserci più regole puntate valide per y. Intuitivamente possiamo dire che una grammatica è LR(0) quando il metodo di riduzione proposto si può sempre applicare, ma vediamo ora una definizione più rigorosa.

Una grammatica G è una **grammatica LR(0)** se:

1. Il simbolo iniziale non compare nella parte destra di nessuna regola (questa requisito si può sempre soddisfare introducendo un simbolo iniziale fittizio S' e la regola $S' \rightarrow S$).

2. Per ogni prefisso LR y di G, se $X \rightarrow x.$ è una regola puntata candidata per y allora nessuna altra regola puntata completa o nessuna altra regola puntata con un simbolo terminale alla destra del punto è una regola puntata valida per y.

Le grammatiche LR(0) sono importanti perché esiste un procedimento che è in grado di creare automaticamente parser per i linguaggi generati da questo tipo di grammatiche. Intuitivamente possiamo dire che questi parser riconoscono il prefisso LR riducibile

di una forma di frase f ed individuano l'unica regola puntata completa che, per definizione di grammatica LR(0), è candidata per f, determinando così la regola applicata nell'ultimo passo della derivazione di f. Questo procedimento applicato ripetutamente permette la riduzione completa di una frase.

Passiamo ora a definire un'altra classe di grammatiche: le grammatiche LR(1).

Una **regola puntata LR(1)** è una regola del tipo:

$$X \to u.w/F$$

dove F è un insieme di caratteri detto **insieme dei simboli susseguenti**:

$$F = \{x \mid x \in (VT \cup \{\$\})\}$$

dove \$ è il **simbolo terminatore** usato per denotare la fine di una frase. L'insieme F è formato da tutti e soli i simboli che possono seguire X in una qualunque forma di frase. Ancora una volta la regola puntata LR(1) è **completa** se è del tipo: $X \to x./F$ mentre diremo che una regola puntata LR(1) $X \to u.w/\{a\}$ è **candidata** per un prefisso riducibile y se esiste una derivazione:

$$S \Rightarrow^*_{rm} vXz \Rightarrow_{rm} vuwz \qquad \text{con } y = vu$$

inoltre:

1. a è il primo simbolo di z, oppure
2. $z = \epsilon$ e $a = \$$.

Una regola puntata LR(1) $X \to u.w/F$ con $F = \{a_1, a_2, \ldots, a_n\}$ è candidata per il prefisso riducibile y se $X \to u.w/\{a_i\}$ è candidata per ogni a_i in F.

Una grammatica è una **grammatica LR(1)** se:

1. Il simbolo iniziale non appare nella parte destra di nessuna regola (come abbiamo visto questo è sempre possibile).
2. Dato l'insieme I delle regole puntate LR(1) valide per un certo prefisso riducibile, se $X \to x./F_x$ e $Y \to y./F_y$ sono in I allora F_x e F_y sono disgiunti.

Le grammatiche LR(1) ci permettono di avere un metodo più efficiente di riduzione di frasi: intuitivamente possiamo dire di essere guidati nel processo di riduzione dall'insieme dei simboli susseguenti F che restringe il campo di ricerca di una regola candidata applicabile nel procedimento di riduzione. Anche per le grammatiche LR(1) è possibile costruire automaticamente un parser, quest'ultimo programma è molto più efficiente dell'analogo parser per grammatiche LR(0) proprio perché può sfruttare l'informazione contenuta nell'insieme dei simboli susseguenti.

Senza dare ulteriori definizioni formali possiamo accennare all'esistenza di una classe di grammatiche, le grammatiche LALR(1), che costituiscono una sottoclasse piuttosto ampia delle grammatiche LR(1), per questa classe di grammatiche è possibile scrivere generatori di parser particolarmente efficienti, uno di questi programmi, detto Bison, è stato utilizzato come strumento di sviluppo di TESI, Bison verrà descritto nel paragrafo 3.2.2.

Notazioni BNF e EBNF per le grammatiche

Il formalismo di Backus-Naur (Backus-Naur Form o BNF) è stato il primo formalismo introdotto per specificare grammatiche, oltre all'utilizzo del metasimbolo "$\rightarrow$" per separare la parte destra e sinistra delle regole utilizza il metasimbolo "|" per raggruppare le parti destre di regole che hanno nella parte sinistra il medesimo simbolo non terminale. Se un simbolo della grammatica coincide con un metasimbolo della notazione BNF viene racchiuso fra apici.

Ad esempio la grammatica:

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow E + E \\ E &\rightarrow n \end{aligned}$$

può pertanto essere scritta come:

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow E + E \mid n \end{aligned}$$

L'introduzione dei costrutti { }, [ ] e ( ) ha portato alla notazione EBNF (Extended BNF): il costrutto $\{x\}$ indica la stringa x ripetuta zero o più volte, la presenza di un esponente come in $\{x\}^n$ indica che il numero massimo di ripetizioni possibili è n.

La scrittura $[x]$ indica che x può apparire una sola volta o non comparire affatto.

Le parentesi tonde consentono di raggruppare più termini alternativi:

$$X \rightarrow uY \mid uv$$

può essere scritto come:

$$X \rightarrow u(Y \mid v)$$

In seguito useremo la notazione EBNF per specificare i linguaggi riconosciuti da TESI.

3.2 Struttura dei compilatori

La programmazione dei computer necessita di linguaggi "ad alto livello" ossia linguaggi astratti che permettano al programmatore di specificare con facilità quali compiti la macchina dovrà eseguire, esempi di tali linguaggi sono il C, il Pascal, il FORTRAN od il BASIC. Purtroppo i computer non possono comprendere linguaggi di questo tipo: si rende quindi necessario un processo di traduzione di un programma in linguaggio ad alto livello (chiamato "programma sorgente" o semplicemente "sorgente") al cosiddetto "codice macchina" che l'elaboratore comprende. Questo procedimento prende il nome di "compilazione" ed i programmi che lo attuano sono perciò chiamati "compilatori".

Nel corso degli anni è stato sviluppato tutto un impianto teorico che consente la generazione quasi automatica di compilatori a partire da una loro descrizione che si basa sui formalismi fin qui introdotti. Non è certo intenzione di chi scrive illustrare queste tecniche anche se in seguito descriveremo brevemente il funzionamento dei compilatori nella misura in cui questo servirà a capire la struttura di TESI che, in un certo senso,

attua una compilazione delle formule matematiche scritte in TEX, al fine di individuarne la struttura sintattica e di presentarla all'utente.

Il primo passo svolto da un compilatore è l'analisi lessicale, il suo scopo è quello di leggere il sorgente un carattere alla volta e raggruppare i singoli caratteri in elementi lessicali atomici, considerati indivisibili nella successiva analisi sintattica, detti **token**. I token possono appartenere a diverse categorie lessicali quali parole chiave (ossia comandi del linguaggio), identificatori (nomi definiti dall'utente e che, ad esempio, descrivono le variabili utilizzate dal programma), numeri, separatori e così di seguito. La seguente espressione TEX ad esempio:

```
$$ \sin (12 + \alpha) $$
```

contiene i token:

- Le parole chiave `$$`, `\sin` e `\alpha`.
- Il numero 12.
- L'operatore +.
- I delimitatori (e).
- Le sequenze di spazi che separano i token appena descritti.

Mano a mano che i token vengono individuati si procede all'analisi sintattica del sorgente.

Durante l'analisi sintattica si costruisce l'albero sintattico del programma sorgente basandosi sulla grammatica che descrive il linguaggio di programmazione utilizzato, l'albero sintattico costituisce la base della successiva analisi semantica.

Con l'analisi semantica si controlla che il significato dei simboli sia coerente con la struttura sintattica individuata. Ad esempio, se un nodo dell'albero sintattico rappresenta un operatore che agisce su numeri interi, si controlla che i suoi figli siano nodi che descrivono numeri interi. L'analisi semantica genera inoltre una rappresentazione elementare delle operazioni descritte nel programma detta codice intermedio.

Una successiva fase, che qui non viene presa in considerazione, permette di passare dal codice intermedio al codice macchina finale.

3.2.1 Analisi lessicale

Come si è visto, lo scopo dell'analisi lessicale è quello di individuare sequenze di caratteri nel sorgente che identificano elementi lessicali del linguaggio detti token. La parte del compilatore che esegue questo compito è chiamata **analizzatore lessicale** o **scanner**.

Espressioni regolari

Un metodo per descrivere i token da riconoscere fa uso delle **espressioni regolari**, le espressioni regolari possono essere così definite: sia VT un vocabolario di simboli terminali allora:

1. $\emptyset$ è un'espressione regolare che rappresenta il linguaggio Φ.
2. ϵ è un'espressione regolare che rappresenta il linguaggio $\{\epsilon\}$.
3. per ogni $t \in VT$, t è un'espressione regolare che individua il linguaggio $\{t\}$.

4. Se r ed s denotano due espressioni regolari che individuano i linguaggi L_r ed L_s allora r^+, r^*, rs, $r|s$, e $[r]$ sono espressioni regolari che denotano rispettivamente i linguaggi L_r^+, L_r^*, $L_r L_s$, $L_r \cup L_s$ e $L_r \cup \{\epsilon\}$.

È possibile dimostrare l'equivalenza fra espressioni regolari e grammatiche regolari, possiamo cioè dire che le espressioni regolari sono un formalismo alternativo (e più coinciso) per la rappresentazione di linguaggi regolari.

Esistono programmi che sono in grado di costruire uno scanner a partire da una descrizione dei token fornita mediante espressioni regolari, per la scrittura di TESI si è utilizzato uno di questi programmi: `flex`, il cui funzionamento viene descritto qui di seguito.

Lex e `flex`

Lex è un programma che genera scanner scritti in linguaggio C, `flex` è un programma compatibile con Lex che viene distribuito gratuitamente.

L'input di `flex` è costituito da un file che presenta la seguente struttura:

dichiarazioni
%%
regole lessicali
%%
funzioni ausiliarie

Le *dichiarazioni* si riferiscono a variabili globali, costanti od espressioni regolari che verranno utilizzate nelle *regole lessicali*. Se in questa sezione sono presenti i simboli `%{` e `%}` tutto il testo contenuto fra di loro sarà copiato nel file di output, in questo modo è per esempio possibile specificare codice C che comparirà in testa allo scanner generato.

La parte *regole lessicali* contiene una descrizione delle espressioni regolari da riconoscere, la sintassi utilizzata è molto simile a quella già vista per le espressioni regolari. Si rimanda a [13] per ulteriori approfondimenti, alcuni dei costrutti disponibili sono qui presentati in ordine di priorità decrescente.

- `x` — indica il linguaggio $\{x\}$.
- `.` — qualsiasi carattere eccetto newline[1].
- `[`$R_1 R_2 \ldots R_n$`]` — una classe di caratteri formata dall'unione degli insiemi R_i.

  R_i indica un singolo carattere od un intervallo, in quest'ultimo caso l'insieme è indicato dagli estremi dell'intervallo separati da "`-`".

  Ad esempio, la scrittura `[a-zA-Z]` indica una qualsiasi lettera dell'alfabeto.
- `[^`$R_1 R_2 \ldots R_n$`]` — indica qualsiasi carattere che non appartiene all'unione degli insiemi R_i, ad esempio `[^0-9]` indica un qualsiasi carattere che non sia una cifra.
- $r*$ — il linguaggio L_r^*.
- $r+$ — il linguaggio L_r^+.

[1] Con "newline" intendiamo il codice utilizzato dalla macchina per segnalare l'andata a a capo all'interno di un file, nel caso dei sistemi DOS ad esempio il newline è rappresentato dalla sequenza di byte 13 e 10.

- `{`*Nome*`}` — una stringa individuata dall'espressone regolare *Nome* che deve essere stata definita nella sezione *dichiarazioni*.
- `\n` — newline.
- `\t` — tabulazione.
- `\`x — il carattere x interpretato letteralmente, questo costrutto è usato per poter rappresentare i matasimboli: ad esempio `\\` rappresenta il carattere " \ ".
- `(`r`)` — le parentesi tonde vengono usate per specificare le precedenze fra i vari operatori.
- rs – il linguaggio $L_r L_s$.
- r`|`s — il linguaggio $L_r \cup L_s$.
- `<`*Condizione*`>`r — una stringa descritta dall'espressione regolare r viene riconosciuta dallo scanner solo se quest'ultimo si trova nella condizione *Condizione*.

  È possibile specificare varie condizioni in cui lo scanner può trovarsi per poter trattare l'input in modo diverso. Ad esempio TESI tratta il testo fra `$$`...`$$` come formula matematica scritta in TEX e quindi processata seguendo le regole lessicali per le formule matematiche, il rimanente testo di input invece viene trattato come un'unica sequenza di singoli caratteri. Per specificare questo diverso trattamento dell'input TESI definisce due diversi modi di funzionamento dell'analizzatore lessicale.
- `<<EOF>>` — specifica il raggiungimento della fine del file di input.

Nel caso due regole siano in conflitto (ossia possano essere entrambi applicate all'input) verrà utilizzata quella che consente di riconoscere il massimo numero di caratteri, a parità di numero di caratteri riconosciuto verrà utilizzata la regola che compare per prima nelle *regole lessicali*.

La descrizione lessicale viene usata da `flex` per costruire l'analizzatore sintattico costituito dalla funzione C `yylex()`.

Quando viene attivato dall'analizzatore sintattico (mediante chiamata alla funzione `yylex()`) l'analizzatore lessicale inizia a scandire il file di input fintanto che non trova una stringa descritta da una delle espressioni regolari contenuta fra le regole lessicali (e che non sia prefisso di una stringa più lunga, pure riconosciuta da una regola lessicale). Ad ogni espressione è possibile associare delle **azioni semantiche**: codice C che viene eseguito quando una stringa corrispondente all'espressione regolare viene riconosciuta. Un'azione semantica può ritornare un valore intero all'analizzatore sintattico (mediante un'istruzione `return`) indicando in questo modo il riconoscimento di un token appartenente ad una data categoria lessicale, vedremo meglio in seguito come questo venga realizzato.

L'informazione fornita dallo scanner è duplice: oltre ad un eventuale valore intero restituito da `yylex()`, usato per identificare la categoria lessicale alla quale il token riconosciuto appartiene, nella variabile globale `yytext` viene salvata anche una stringa che corrisponde al token riconosciuto.

Infine, nella sezione *funzioni ausiliarie*, compare del codice C che verrà attaccato in coda al codice dello scanner e che solitamente contiene funzioni richiamate dalle azioni semantiche.

3.2.2 Analisi sintattica

Abbiamo già definito l'analizzatore sintattico o parser come quella parte di codice che, data una frase di un linguaggio, ne ricostruisce l'albero sintattico dopo che l'analisi lessicale ne ha individuato i token che la compongono.

Un altro scopo, non meno importante, dell'analisi sintattica è costituito dalla gestione degli errori sintattici: il parser deve individuare condizioni di errore, ossia casi in cui la frase in input non corrisponde ad una frase del linguaggio riconosciuto dal parser. La rilevazione degli errori deve:

1. Segnalare gli errori sintattici, fornendo quante più informazioni possibili sulla loro natura.

2. "Recuperare" gli errori, ossia gestire le situazioni di errore in modo da poter continuare l'analisi sintattica della porzione di input rimasta.

3. Non rallentare eccessivamente il processo di analisi sintattica.

Grammatiche ad attributi

Come abbiamo visto l'analisi sintattica è la fase che precede la successiva analisi semantica: si rende quindi necessario un metodo che permetta di gestire il flusso di informazioni semantiche prodotto dall'analisi sintattica.

Il formalismo che permette la gestione delle informazioni semantiche durante l'analisi sintattica è costituito dalle cosiddette **grammatiche ad attributi**: le grammatiche ad attributi sono un'estensione delle grammatiche non contestuali in cui ad ogni simbolo della grammatica può essere associato un attributo che possiede un tipo ed un valore. Inoltre, le regole sintattiche vengono integrate con delle **regole semantiche** costituite da una serie di istruzioni dette **azioni semantiche**. Le azioni semantiche vengono eseguite quando la regola sintattica cui sono associate viene utilizzata nel processo di riduzione e solitamente modificano il valore degli attributi associati ai simboli che compaiono nella regola.

Per permettere di rappresentare gli attributi in una frase si ricorre ad una versione estesa dell'albero sintattico definito nella sezione 3.1.3 in cui ogni nodo, oltre a contenere il simbolo della grammatica cui è associato, ne contiene pure gli attributi.

Gli attributi possono distinguersi in **sintetizzati** ed **ereditati**: queste definizioni hanno a che fare con la direzione del flusso di informazioni a loro associate. Se

$$X \to x_1 x_2 \dots x_n$$

è una regola della grammatica cui è associata una azione semantica A, quest'ultima può:

1. Assegnare una valore ad un attributo sintetizzato del simbolo X.

   A sarà del tipo: $s := f(z_1, z_2, \dots, z_m)$, dove s è un attributo sintetizzato di X e ciascun z_i è un attributo ereditato di X oppure un attributo sintetizzato di x_j con $1 \leq j \leq n$.

2. Assegnare un valore ad uno degli attributi ereditati degli x_i.

   A sarà del tipo: $e := f(z_1, z_2, \dots, z_m)$, dove e è un attributo ereditato di x_j con $1 \leq j \leq n$ e ciascun z_i è un attributo ereditato di X oppure un attributo sintetizzato di x_j con $1 \leq j \leq n$.

Nel primo caso questo equivale a propagare l'informazione semantica dalle foglie verso la radice dell'albero sintattico, nel secondo caso invece l'informazione viaggia in senso inverso: gli attributi ereditati contengono quindi informazioni associate al contesto del simbolo cui appartengono.

Le foglie dell'albero sintattico possiedono attributi sintetizzati che, come vedremo in seguito, sono forniti dall'analizzatore sintattico. A questo punto non dovrebbe essere difficile immaginare che gli attributi conterranno informazioni sul token corrispondente al simbolo terminale associato.

Vediamo ora di fornire un esempio di grammatica ad attributi che dovrebbe chiariere i concetti appena introdotti. In questo esempio l'attributo di nome *Attrib* associato al simbolo X viene indicato con `X.Attrib`. Se uno stesso simbolo appare più volte in una regola, ogni istanza sarà indicata con un pedice progressivo, es.:

$$X_1 \rightarrow uX_2w$$

gli attributi corrispondenti saranno indicati con: `X1.Attrib` e `X2.Attrib`. La grammatica del nostro esempio descrive semplici espressioni matematiche:

$$\begin{array}{rcll} S & \rightarrow & E & \{\texttt{S.V = E.V}\} \\ E_1 & \rightarrow & E_2 + T & \{\texttt{E1.V = E2.V + T.V}\} \\ E & \rightarrow & -T & \{\texttt{E.V = -T.V}\} \\ E & \rightarrow & T & \{\texttt{E.V = T.V}\} \\ T_1 & \rightarrow & T_2 * F & \{\texttt{T1.V = T2.V * F.V}\} \\ T & \rightarrow & F & \{\texttt{T.V = F.V}\} \\ F & \rightarrow & (E) & \{\texttt{F.V = E.V}\} \\ F & \rightarrow & n & \{\texttt{F.V = n.VAL}\} \end{array} \tag{3.1}$$

In questa grammatica n è un simbolo terminale che rappresenta un numero, l'attributo `n.VAL` è il valore corrispondente; tale attributo deve venire opportunamente inizializzato quando l'analizzatore lessicale riconosce il token corrispondente al simbolo terminale n.

Se consideriamo la frase: $1 + 2 * 3$ vediamo che è composta dai token "1", "2", "3", "+" e "$*$", ai primi tre token che, in quanto numeri, corrispondono al simbolo terminale n della nostra grammatica, sono associati tre attributi che sono rispettivamente inizializzati ai valori 1, 2 e 3. Nel processo di riduzione della frase la prima regola applicata sarà:

$$T_1 \rightarrow T_2 * F \quad \{\texttt{T1.V = T2.V * F.V}\}$$

in relazione alla sotostringa $2 * 3$, verrà quindi creato un nodo "+" dell'albero sintattico cui sarà associato un attributo il cui valore sarà il prodotto fra gli attributi dei nodi figli ossia $2 * 3 = 6$. È ovvio che procedendo in questo modo l'attributo associato al simbolo iniziale S conterrà il valore dell'espressione matematica di cui si è costruito l'albero sintattico. Volendo possiamo stampare il valore dell'espressione modificando la prima regola della grammatica in:

$$S \rightarrow E \quad \{\ \texttt{Writeln (E.V)}\ \}$$

Quando compaiono regole semantiche di questo tipo, in cui vengono eseguite azioni che richiamano procedure esterne per svolgere compiti non strettamente collegati con

l'analisi sintattica, la grammatica viene detta **grammatica ad attributi operazionali**.

Yacc e Bison

Yacc[2] è un generatore di parser: a partire da un file di specifiche contenente una grammatica LALR(1) con attributi, genera un codice C che costituisce l'analizzatore sintattico per la grammatica data. Bison è un programmma compatibile con Yacc (nel senso che ne accetta gli stessi file di input) e che viene distribuito gratuitamente. Bison è stato utilizzato per la scrittura di TESI.

L'analizzatore sintattico è costituito dalla funzione C `yyparse()` la quale è predisposta per lavorare con gli scanner generati da `flex`: ogni volta che ha bisogno di un token per l'analisi sintattica, `yyparse()` chiama la funzione `yylex()` che, come abbiamo visto, costituisce lo scanner generato da `flex`. È chiaro a questo punto come sia possibile automatizzare analisi lessicale, sintattica e semantica: basta scrivere una serie di espressioni regolari che individuano i token del linguaggio che si vuole riconoscere e fornirle come input a `flex` il quale genera l'analizzatore lessicale. La grammatica del linguaggio da compilare viene poi scritta sotto forma di grammatica ad attributi (le cui azioni semantiche si occuperanno dell'analisi semantica): fornita questa grammatica come input a Bison, quest'ultimo genererà l'analizzatore sintattico sotto forma di una funzione C, `yyparse()` appunto, la quale si interfaccierà automaticamente con l'analizzatore lessicale chiamando la funzione `yylex()`.

La struttura del file di input di Bison (che chiameremo sorgente Bison) è molto simile a quella del file sorgente per `flex`:

dichiarazioni
%%
regole sintattiche
%%
funzioni ausiliarie

La sezione *dichiarazioni* contiene una serie di definizioni: è in questa sezione che vengono specificati i simboli terminali della grammatica mediante la direttiva `%token`. Ad ogni simbolo definito in questo modo Bison associa univocamente un numero intero. Con opportune direttive è possibile far generare a Bison un file C che contiene tali valori. Questo file può essere usato da `flex` per conocere la corrispondenza fra token e numeri interi: tale informazione può essere usata all'interno della funzione `yylex()` per ritornare un valore corrispondente al token riconosciuto. Bisogna dire che, se lo scanner generato da `flex` non riconosce il prossimo carattere di input come facente parte di qualche espressione regolare, viene ritornato all'analizzatore sintattico il codice ASCII del carattere. Per come Bison definisce i simboli terminali non è possibile che si verifichi un conflitto fra codici ASCII e valori generati dal comando `%token`: è quindi possibile usare singoli caratteri come simboli terminali nelle regole sintattiche di Bison racchiudendoli fra apici, senza bisogno di dichararli con il comando `%token`.

Ad esempio: immaginiamo di creare un file `MATH.Y`[3], che contenga la grammatica usata nell'esempio 3.1 ed utilizziamo il comando:

```
%token NUMERO
```

[2] Yacc è l'acronimo di "Yet Another Compiler-Compiler" ossia "Ancora un altro compilatore di compilatori".

[3] `.Y` è l'estensione solitamente usata per i file contenenti una grammatica per Bison.

Bison associerà all'identificatore C `NUMERO` un valore intero, tale valore verrà scritto nel file `MATH.H` sotto forma di una direttiva del tipo:

```
#define NUMERO 257
```

a questo punto creiamo un file `MATH.L`[4] di questo tipo:

```
{%
#include "math.h"
%}

%%

[0-9]+ {return NUMERO;}
```

Vediamo che, quando viene incontrata una sequenza di cifre, lo scanner generato da `flex` ritorna il valore associato a `NUMERO`, in questo modo il parser creato da Bison ha modo di sapere che nel file di input è stato riconosciuto un numero. Come detto sopra `yylex()` salva un'altra informazione: la variabile globale `yytext` conterrà la stringa corrispondente al numero riconosciuto.

A questo punto vediamo come vanno specificate le regole della grammatica da riconoscere, regole che costituiscono la sezione *regole sintattiche*.

Data ad esempio una grammatica come la seguente:

$$\begin{aligned} X &\rightarrow x_1^1 x_2^1 \dots x_n^1 \{Azioni\ semantiche\ 1\} \\ X &\rightarrow x_1^2 x_2^2 \dots x_n^2 \{Azioni\ semantiche\ 2\} \\ Y &\rightarrow y_1 y_2 \dots y_n \{Azioni\ semantiche\ 3\} \end{aligned}$$

essa deve essere tradotta in un sorgente Bison con le seguenti *regole sintatiche*:

```
X
    : x_11 x_12 ... x_1n {
            Azioni semantiche 1
        }
    | x_21 x_22 ... x_2n {
            Azioni semantiche 2
        }
;

Y
    : y_1 y_2 ... y_n {
            Azioni semantiche 3
        }
;
```

Come si vede ogni regola è costituita da due sezioni separate da `":"`. La prima sezione costituisce la parte sinistra di una regola della grammatica da riconoscere, per definizione di grammatica non contestuale questa parte contiene solo un simbolo non

[4] `.L` è l'estensione solitmente usata per i file di input per `flex`.

terminale. La seconda sezione è costituita da parti destre di regole che hanno come parte sinistra il simbolo che compare nella prima sezione, ciascuna di queste parti destre è separata dalle altre da un "`|`". Ad ogni regola specificata in questo modo sono associate delle azioni semantiche che vengono eseguite quando la regola corrispondnte viene utilizzata nel processo di riduzione di una frase. Chiude il blocco un "`;`".

Tutti i simboli che compaiono nelle regole e non sono stati definiti con il comando `%token` vengono considerati simboli non terminali della grammatica e pertanto devono comparire nella parte sinistra di almeno una regola. Il simbolo iniziale è quello che compare come parte sinistra della prima regola o, in alternativa, quello specificato mediante il comando `%start` nella sezione *dichiarazioni*.

Si possono specificare commenti alle regole grammaticali racchiudendoli fra `/*...*/`.

All'interno delle regole semantiche è possibile avere accesso agli attributi dei simboli che compaiono in una regola: col simbolo `$$` si indica l'attributo associato alla parte sinistra di una regola, con il simbolo $\$n$ si indica l'attributo associato al simbolo che compare nell'n-esima posizione della parte sinistra di una regola. Abbiamo visto che gli attributi sintetizzati appartenenti ai simboli teminali della grammatica vengono inizializzati dallo scanner: questo può essere fatto nelle regole semantiche associate alle espressioni regolari contenute nel file di input di `flex`: la variabile globale `yylval` contiene l'attributo associato al token appena riconosciuto, è compito del programmatore inizializzare opportunamente `yylval`. Quando all'interno delle azioni semantiche si userà il simbolo $\$n$ per riferirsi ad un attributo di un simbolo terminale, automaticamente verrà restituito il valor messo in `yylval` dallo scanner.

Il tipo degli attributi viene specificato in due modi diversi nella parte delle *dichiarazioni*: se tutti i simboli della grammatica possiedono attributi dello stesso tipo questo viene dichiarato definendo esplicitamente il simbolo `YYSTYPE`, ad esempio se tutti i simboli hanno come attributi numeri interi bisognerà fornire la dichiarazione:

```
#define YYSTYPE int
```

all'interno del sorgente Bison. Se invece i simboli posseggono attributi di tipo diverso bisogna specificare tutti questi tipi con il comando `%union` associando un nome ad ogni tipo di attributo. Il comando `%union` va specificato nella sezione *dichiarazioni*. Se, ad esempio, gli attributi possono essere numeri interi o stringhe di caratteri possimo usare la dichiarazione:

```
%union {
   int numero;
   char* stringa;
}
```

Il tipo di un token va specificato nella sua dichiarazione, ad esempio il comando:

```
%token <numero> NUM
```

specifica che il simbolo terminale `NUM` possiede attributi di tipo `numero` definito prima come `int`. Nel caso gli attributi siano di tipo diverso bisogna specificare il tipo di attributi associati a ciascun simbolo non terminale, questo viene fatto mediante il comando `%type` nella sezione *dichiarazioni*.

A questo punto abbiamo visto come indicare le regole sintattiche e le azioni semantiche, passiamo a considerare il modo in cui Bison effettua la riduzione di una frase.

Il parser creato da Bison utilizza nel processo di riduzione uno stack[5]: nello stack vengono salvati i simboli, terminali e non, incontrati durante il processo di riduzione, il processo di immissione di un simbolo nello stack viene detto **shift**. Quando nello stack appare una sequenza di simboli che coincide con la parte destra di una regola della grammatica (ossia un prefisso riducibile), il parser rimuove dallo stack questi simboli sostituendoli col simbolo non terminale che compare nella parte sinistra della regola, contemporaneamente vengono eseguite le azioni semantiche associate alla regola applicata: questa procedura viene detta **reduce**. Il processo di reduce non è comunque automatico: il parser prima controlla il valore del prossimo simbolo in input non ancora inserito nello stack, che chiameremo **simbolo susseguente**, per decidere se eseguire subito l'operazione di reduce o se fare lo shift del simbolo stesso. Tramite sequenze di shift e reduce il parser tenta di arrivare ad una situazione in cui, una volta scandito l'intero input, sullo stack compaia solo il simbolo iniziale della grammatica, in tal caso la frase è stata ridotta con successo.

In pratica il parser è in grado di riconoscere i prefissi LR riducibili della frase in input e di individuare la regola adatta alla loro riduzione, per fare questo esso sfrutta l'indicazione fornita dal simbolo susseguente. Intutivamente possiamo dire che il parser controlla se tale simbolo compare nell'insieme dei simboli susseguenti di una regola puntata LR(1) (queste regole sono generate automaticamente da Bison a partire dalla grammatica fornita) per capire se l'applicazione di tale regola può far progredire nel processo di riduzione. Tutto questo procedimento è reso possibile dal fatto che la grammatica fornita è una grammatica LALR(1), in caso contrario possono verificarsi due tipi di errore detti rispettivamente **conflitto shift/reduce** e **conflitto reduce/reduce**.

Un conflitto shift/reduce compare quando sia un'operazione di shift che un'operazione di reduce sono ammesse. Consideriamo la grammatica:

```
%token IF THEN ELSE VAR

stmt
    : expr
    | if_stmt
;

ifstmt
    : IF expr THEN stmt
    | IF expr THEN stmt ELSE stmt
;

expr:
    VAR
;
```

Supponiamo che sullo stack sia già presente la sequenza di simboli `IF`, `expr`, `THEN` e `stmt` ed il simbolo susseguente sia `ELSE`. In questo caso si verifica un conflitto shift/reduce perché il parser può, in virtù della prima regola relativa a `ifstmt`, ridurre i simboli sullo stack, oppure può eseguire uno shift del simbolo `ELSE`, seguito da un reduce dettato dalla seconda regola relativa a `ifstmt`. In questi casi il parser opta sempre per uno shift, se questa scelta non è conforme al linguaggio che si vuole

[5]Uno stack è una struttura dati in cui dei valori possono essere salvati e venire in seguito recuperati nell'ordine inverso a quello in cui sono stati immessi.

definire è necessario riscrivere la grammatica: conflitti shift/reduce nascono infatti da grammatiche ambigue o grammatiche che non sono LALR(1).

Il conflitto shift/reduce e la sua risoluzione giocano un ruolo importante nello stabilire precedenza ed associatività degli operatori. Consideriamo la seguente regola:

```
exp
    : exp '+' exp
    | exp '*' exp
    | NUM
;
```

Questo tipo di regola genera sicuramente un conflitto shift/reduce: basta considerare la frase: `1 + 2 * 3`. Quando nello stack è presente l'espressione `1 + 2` come deve comportarsi il parser? Deve esegure un reduce od uno shift di "`*`"? Nel caso il parser esegua un reduce l'input verrebbe interpretato come: `(1+2) * 3`, uno shift invece metterebbe `*` sullo stack, seguito poi da `3`, a questo punto due successivi reduce porterebbero a riconoscere la frase come `1+(2*3)`. Vediamo dunque che un diverso modo di risolvere il conflitto shift/reduce porta ad una diversa priorità degli operatori.

L'ambiguità si presenterebbe pure con l'espressione `1 + 2 + 3`: una volta che sullo stack si trova `1 + 2` come si deve trattare il successivo "`+`"? È facile vedere come uno shift porta all'interpretazione `1 + (2+3)`, mentre un reduce porta a `(1+2) + 3`. In questo caso il diverso modo di risolvere il conflito shift/reduce porta a considerare l'operatore `+` rispettivamente come associativo a destra od a sinistra.

Bison mette a disposizione un metodo per definire priorità ed associatività degli operatori in casi come questi: anziché definire un operatore con il comando `%token` si utilizzano i comandi `%left`, `%right` e `%nonassoc` che definiscono rispettivamente un operatore associativo a sinistra, a destra od un operatore che non può comparire due volte nella stessa espressione. La precedenza degli operatori dipende dall'ordine in cui vengono dichiarati con questi comandi: gli operatori dichiarati per primi hanno priorità minore di quelli seguenti. Le regole della grammatica hanno una priorità pari a quella dell'ultimo simbolo terminale che compare nella loro parte destra; ossia, se due regole possono essere entrambi applicate in un medesimo istante, verrà applicata la regola il cui simbolo terminale più a destra ha priorità maggiore. Con questi comandi il Bison è in grado di risolvere il tipo di conflitti shift/reduce appena descritti.

Per inciso, esiste la possibilità di fornire esplicitamente la priorità di una regola usando il comando `%prec` che va posto alla fine della parte destra, seguito da un simbolo terminale. La priorità della regola sarà pari a quella del simbolo terminale indicato.

L'altro tipo di conflitto, il conflitto reduce/reduce, è invece molto più insidioso: si ha un conflitto di questo tipo quando sono ammissibili due diverse riduzioni per una

forma di frase. Ecco un esempio di grammatica che genera questo errore:

```
sequenza
    : /* Regola vuota: seq può essere la stringa nulla */
    | forseparola
    | sequenza forseparola
;

forseparola
    : /* Regola vuota */
    | PAROLA
;
```

Si vorrebbe definire una `sequenza` come una succesione di zero o più `PAROLA` ma, se consideriamo ad esempio una frase formata da una singola `PAROLA`, questa può essere ridotta in più modi:

$$\texttt{sequenza} \Rightarrow \texttt{forseparola} \Rightarrow \texttt{PAROLA}$$

ma anche:

$$\texttt{sequenza} \Rightarrow \texttt{sequenza forseparola} \Rightarrow \texttt{forseparola} \Rightarrow \texttt{PAROLA}$$

Anche se la frase viene comunque riconosciuta possono in questo modo venire attivate azioni semantiche diverse.

Bison risolve il problema applicando nella riduzione la regola che compare per prima nella grammatica. Solitamente però un conflitto di questo tipo indica un serio errore nella grammatica che va studiato caso per caso. Un'altra causa di un errore di questo tipo può essere semplicemente il fatto che la grammatica, pur non ambigua, non è una grammatica LALR(1). Sebbene sia possibile scrivere generatori di parser per grammatiche LR(0), le quali costituiscono un superset delle grammatiche LALR(1), e che quindi eliminerebbero alcuni conflitti reduce/reduce, tali parser sono inefficienti rispetto a quelli generati da Bison: si preferisce quindi lasciare al programmatore il compito di determinare le cause dei conflitti reduce/reduce e risolverli modificando la grammatica.

Un altro argomento da considerare è la gestione degli errori sintattici che, come abbiamo visto, costituisce un compito importante di un parser. Bison mette a disposizione il simbolo `error`: quando il parser incontra una situazione di errore, inizia ad eliminare simboli dallo stack fintanto che non si presenta una situazione in cui un simbolo `error` sarebbe valido. Una volta che questa situazione si è verificata, il parser legge e scarta simboli in ingresso finché non ne trova uno che sia valido per la regola in cui il simbolo `error` compare: a questo punto il simbolo viene messo sullo stack e l'analisi sintattica può proseguire normalmente.

Consideriamo ad esempio un linguaggio di programmazione in cui tutte le linee siano formate da comandi, eventualmente seguiti da un'espressione, e separati da "`;`". Una regola per riconoscere una linea potrebbe essere la seguente:

```
linea
    : comando ';'
    | comando espressione ';'
    | comando error ';'
;
```

La terza riga della regola dice a Bison che un comando seguito da un errore e da un punto e virgola costituisce una valida linea di comando. Se, durante la scansione

di un'espressione, il parser incontra un errore sintattico elimina dallo stack i simboli riconosciuti fino a quel momento (che costituiscono la parte di espressione processata prima dell'errore) finché resta solo un simbolo `comando`. A questo punto il simbolo `error` può venire accettato in virtù dell'ultima regola: il parser quindi inizia a leggere e scartare simboli finché non incontra un "`;`" che viene messo sullo stack. La regola viene dunque attivata e la linea contenente l'espressione errata viene comunque riconosciuta di modo che l'analisi può proseguire normalmente per le linee seguenti.

Quando viene incontrato un errore il parser chiama la funzione `yyerror()` che deve essere fornita dall'utente ed a cui viene passata una stringa contenete un messaggio di errore.

Bison consente di specificare delle azioni semantiche all'interno di una regola, come nel seguente esempio:

```
cmd
    : stmt {x = 1;} exp ';' {
            printf ("Ok!");
        }
;
```

In questo caso l'istruzione `x = 1` viene eseguita non appena il token `stmt` è presente sullo stack. Questo tipo di costrutto rende possibile la gestione del contesto della frase mediante alcune tecniche note come "kludges"[6].

Vediamo una applicazione pratica di questo concetto: nelle formule matematiche i numeri sono considerati singoli token, quindi lo scanner dovrebbe interpretare una sequenza di cifre come un numero intero che forma un token a sé stante. Il TEX però permette di specificare apici e pedici usando rispettivamente i caratteri "`^`" e "`_`": il primo carattere, e solo il primo, che segue questi comandi viene usato come apice o pedice. Così il testo:

```
$ x^12 $
```

viene interpretato da TEX come "$x^1 2$" e non come "x^{12}". Vediamo quindi che una sequenza di cifre non sempre si può interpretare come un singolo token: in questo caso la sequenza va spezzata in due.

Per risolvere questo problema esistono due modi: nello scanner si potrebbe controllare ogni volta che vengono riconosciuti dei token per vedere se essi introducano un apice od un pedice quindi, facendo entrare lo scanner in una modalità di funzionamento opportuna, si potrebbe gestire la lettura dell'apice/pedice; salvo ritornare poi nella modalità standard di funzionamento per processare il resto della formula.

Il secondo metodo consiste nell'usare una regola Bison come la seguente:

```
esponente
    : '^' {flag = 1;} numero {
            /* Azioni semantiche */

            flag = 0;
        }
;
```

Dove `flag` è una variabile globale visibile da `yylex()` che viene inizializzata a zero. Appena nell'input si incontra un "`^`", che indica un apice, tale simbolo viene messo sullo

[6] Con questo termine si indica una tecnica funzionante per risolvere un problema di programmazione ma che non è né molto elegante, né molto robusta.

stack e l'azione semantica `flag = 1` viene subito eseguita. Lo scanner, quando incontra una sequenza di cifre, controlla il valore di `flag`: se è uguale ad uno significa che si sta processando un esponente e quindi solo il primo carattere della sequenza va restituito all'analizzatore sintattico. In caso contrario lo scanner restituisce l'intera sequenza di cifre come un numero. Una volta che l'analizzatore sintattico ha riconosciuto l'apice il valore di `flag` viene riportato a zero.

Nella stesura di TESI si è preferito utilizzare quest'ultima tecnica per la gestione di apici e pedici; innanzitutto per evitare il proliferare delle condizioni definite nello scanner e poi per facilitare il recupero dalle situazioni di errore, maggiormente gestibili durante l'analisi sintattica.

Un ultimo argomento riguarda la posizione dei token all'interno del file di input: nelle regole semantiche il simbolo `@`n indica una struct come questa:

```
struct {
    int first_line, last_line;
    int first_column, last_column;
}
```

che contiene informazioni sulla posizione all'interno del file di input dell'n-esimo simbolo che compare nella regola. Ad esempio, per indicare la riga dove inizia il token corrispondente al secondo simbolo che compare nella parte destra di una regola, si usa `@2.first_line`. Se si usa questo costrutto `yylex()` deve preoccuparsi di fornire queste informazioni, per farlo deve utilizzare la struct `yylloc`, Bison si preoccuperà di leggere le informazioni contenute in questa variabile e di salvarle nella struct `@`n opportuna.

Capitolo 4

Struttura del programma: elaborazione del file di configurazione

TESI si compone essenzialmente di quattro parti:

1. Il modulo che si occupa di processare il file di configurazione.

   Questo modulo crea una tabella dei vari comandi TEX che vengono definiti memorizzando per ognuno, oltre alla classe lessicale di appartenenza, le modalità di lettura.

2. Il modulo che esegue l'analisi sintattica del file TEX in input.

   Questo modulo, basandosi sulle informazioni contenute nella tabella creata dal modulo precedente, analizza il file TEX in input individuando le formule matematiche presenti al suo interno e costruendo una struttura dati che ne ricalca l'albero sintattico.

   In questa parte del programma vanno incluse alcune routine che, terminata l'analisi sintattica, svolgono un'azione di "potatura" dell'albero sintattico per semplificarne la struttura e facilitare la presentazione delle formule all'utente.

3. Le routine di interfaccia con l'utente che permettono la visita degli alberi sintattici creati dal modulo precedente.

4. Funzioni ausiliarie quali gestione della memoria, routine sonore etc.

I primi due moduli: quello che processa il file di configurazione e quello che svolge l'analisi sintattica delle formule, sono costituiti da due coppie scanner-parser generati rispettivamente con `flex` e Bison. Il resto del programma è costituito da codice C.

Durante la stesura del programma si è posta particolare attenzione alla sua portabilità: ad eccezione di poche funzioni che si interfacciano direttamente all'hardware (come quelle che gestiscono il suono) e che sono state confinate all'interno di un paio di file, il codice è ANSI C perfettamente portabile.

Il sistema su cui TESI è stato sviluppato è un PC IBM compatibile con sistema operativo MS-DOS 6.0 + Windows 3.1. Come ambiente di sviluppo è stato utilizzato il Microsoft C/C++ 7.0, naturalmente in congiunzione con `flex` (versione 2.5.1) e Bison (GNU Bison versione 1.25).

In questo capitolo viene esaminata la struttura del primo di questi quattro moduli, le restanti componenti di TESI saranno esaminate nei capitoli seguenti.

4.1 File sorgenti

Il modulo che esegue l'analisi del file di configurazione è composto dai seguenti file sorgente:

- `CFGDEF.H` — Questo file contiene alcune definizioni dei tipi di dati usati per la creazione della tabella coi comandi TEX riconosciuti, oltre che alla dichiarazione di alcune variabili globali.
- `CFGSCAN.L` — Il sorgente `flex` per lo scanner del file di configurazione.
- `CFGSCAN.C` — Il file creato da `flex` che contiene la funzione `yycfg_lex()`, la quale implementa lo scanner vero e proprio.

  Avevamo detto che `flex` solitamente chiama questa funzione `yylex()` ma, visto che nel programma TESI sono presenti due scanner (il secondo relativo al file TEX in input), si creerebbe un conflitto se entrambi gli scanner avessero lo stesso nome. `flex` rende disponibile lo switch `-P` che, se specificato sulla linea di comando, cambia il prefisso "`yy`", usato di default per tutti i simboli globali, in un prefisso scelto dall'utente: nel nostro caso il prefisso scelto è `yycfg_` per lo scanner del file di configurazione e `yytex_` per lo scanner del file TEX.

  Bisogna ricordare che il prefisso viene cambiato a tutti i simboli globali, ad esempio, all'interno dello scanner del file di configurazione, la variabile che contiene il testo del token riconosciuto si chiamerà `yycfg_text` e non più `yytext`.
- `CFGPARSE.Y` — Il sorgente Bison del parser del file di configurazione.
- `CFGPARSE.C` — File creato da Bison, il parser vero e proprio è costituito dalla funzione `yycfg_parse()`.

  Anche per questo file possiamo ripetere i discorsi fatti in precedenza sui prefissi dei simboli globali.
- `CFGPARSE.H` — Questo file viene creato da Bison e contiene la definizione dei valori numerici associati ai simboli terminali della grammatica definita in `CFGPARSE.Y`.

  Nel paragrafo 3.2.2, ove abbiamo descritto il funzionamento di Bison, abbiamo spiegato le modalità di impiego di questo file.

4.2 Il le `CFGDEF.H`

Come abbiamo visto questo file contiene la dichiarazione di alcuni tipi di dati necessari alla creazione della tabella che contiene i comandi TEX riconosciuti. Questa tabella è costituita da una lista di elementi di tipo `TSYMTAB`:

```
typedef struct tsymtab {
    TTOKEN *entry;
    struct tsymtab *next;
} TSYMTAB;
```

`next` punta al prossimo elemento della lista mentre `entry` punta ad una struttura

`TTOKEN` che contiene i dati veri e propri sul comando TEX da riconoscere:

```
typedef struct ttoken {
    char *name;
    int  code;
    char *read;
    char *readleft;
    char *readlong;
} TTOKEN;
```

i vari campi di questa struttura contengono rispettivamente:

- `name` — Una stringa con il comando TEX (es. `\sum` oppure "`+`") che costituisce l'entry nella tabella.
- `code` — Un numero che rappresenta il valore da restituire all'analizzatore sintattico del file TEX quando il comando `name` viene incontrato nel file di input.

  Abbiamo già spiegato nel paragrafo 3.2.2 come lo scanner indichi in questo modo al parser il simbolo terminale corrispondente al token riconosciuto.

  I valori numerici da restituire sono definiti nel file `TEXPARSE.H` generato da Bison.
- `read` — La stringa da mandare al sintetizzatore vocale quando il comando TEX deve venire letto all'utente.
- `readleft` — La stringa da leggere nel caso sia specificata l'opzione `LeftDiscrim`[1] nel file di configurazione.
- `readlong` — La stringa da leggere nel caso il comando identifichi un'espressione troppo complessa per essere letta interamente.

Non tutti gli elementi di questa struttura saranno utilizzati: ad esempio per il comando TEX `\alpha`, che indica la lettera α, i campi `readleft` e `readlong` non hanno senso.

4.3 Il le `CFGPARSE.Y`

Questo file è il sorgente Bison della grammatica che definisce il linguaggio di programmazione utilizzato nel file di configurazione.

4.3.1 Variabili globali

In questo file vengono innanzitutto definite alcune variabili globali, le più importanti sono:

```
TSYMTAB *symtable = NULL;
```

`symtable` è un puntatore ad una struttura `TSYMTAB` che costituisce la testa della lista che forma la tabella dei comandi TEX riconosciuti da TESI.

Seguono alcune variabili corrispondenti ad opzioni di personalizzazione di TESI definibili nel file di configurazione:

[1] Vedi pag. 18.

```
/*
Massima complessita' affinche' una formula venga letta (comando
MaxComplexity)
*/
int maxcomplexity = 5;

/*
Stringhe da stampare per pause brevi e pause lunghe
(comandi PausaBreve e PausaLunga
*/
char shortpause[STRLEN] = "";
char longpause[STRLEN] = "";

/*
Flag: se TRUE bisogna leggere alcuni operatori diversamente
a seconda della direzione in cui si esplora l'albero sintattico
(comandi LeftDiscrim e NoLeftDiscrim)
*/
int lftdiscrim = FALSE;

/*
Pausa in millisecondi dopo aver stampato una stringa usando il BIOS
(comando SynthPause)
*/
unsigned long synthpause = 0L;
```

4.3.2 Grammatica per il le di congurazione

La sezione seguente del file `CFGPARSE.Y` definisce la sintassi per i comandi nel file di configurazione, nell'appendice B è mostrata la grammatica priva di commenti, per una lettura più immediata.

I simboli della grammatica possiedono attributi di tre tipi diversi:

```
%union {
    char *str;
    int tok;
    int integer;
}
```

`integer` indica un attributo che contiene un numero intero. Attributi di questo tipo sono associati al simbolo terminale `C_INTEGER`, usato per indicare parametri numerici per alcune opzioni definibili nel file di configurazione.

`str` indica attributi che contengono una stringa. Attributi di questo tipo sono associati al simbolo terminale `C_STRING` che rappresenta una stringa racchiusa fra doppi apici.

`tok` viene usato per attributi associati a istruzioni che definiscono un comando TEX: tali attributi contengono il valore intero che il parser del file TEX si aspetta di ricevere in corrispondenza ad una certa categoria lessicale, questo valore è salvato nella tabella dei comandi TEX riconosciuti e sarà utilizzato dallo scanner del file TEX ogni volta che il comando TEX in questione verrà individuato.

Facciamo un esempio per chiarire questo procedimento: il parser del file TEX associa ad un operatore che abbia la stessa sintassi dell'operatore "+" il simbolo terminale `T_OPSOMMA`, un operatore di questo tipo viene definito nel file di configurazione dal comando:

```
OperatoreSomma "+": "più";
```

Quando lo scanner del file di configurazione riconosce il comando `OperatoreSomma` vi associa un attributo di tipo `tok` che vale `T_OPSOMMA` (ricordiamo che Bison genera un file `.H` che contiene i valori associati ai simboli terminali della grammatica). Tale valore viene salvato nella tabella dei comandi TEX riconosciuti da TESI nella posizione corrispondente al comando "+". Quando lo scanner del file TEX individuerà il token "+" all'interno di una formula, consulterà la tabella e restituirà al parser proprio il valore `T_OPSOMMA`. Questo meccanismo permette all'utente di definire il modo in cui TESI deve interpretare i vari comandi che compaiono nelle espressioni scritte in TEX.

Vediamo ora le varie regole che compongono la grammatica definita in questo file: nella descrizione che segue i nomi minuscoli indicano simboli non terminali, i simboli terminali sono indicati con lettere maiuscole o racchiusi fra apici se costituiti da un singolo carattere.

`input` $\rightarrow \epsilon$ | `input statement ';'`
`input` $\rightarrow$ `error ';'`
Queste regole definiscono `input`, che costituisce il simbolo iniziale della grammatica, come una sequenza di zero o più `statement` separati da ";". Come vedremo uno `statement` rappresenta un singolo comando del file di configurazione.

L'ultima di queste regole implementa la gestione degli errori di sintassi, le azioni semantiche associate alla regola provvederanno a segnalare all'utente il numero di linea del file che ha generato l'errore.

`statement` $\rightarrow$ `C_LFT_DISCR` | `C_NO_LFT_DISCR`
`statement` $\rightarrow$ `C_ERROR_BELL` | `C_NO_ERROR_BELL`
`statement` $\rightarrow$ `C_READALL` | `C_NO_READALL`
Queste regole implementano la gestione di alcune opzioni che non necessitano di parametri aggiuntivi e che corrispondono ai comandi `(No)LeftDiscrim`, `(No)ErrorBell` e `(No)ReadAllFormula`.

Le azioni semantiche associate a questi comandi si occupano di impostare alcuni flag booleani che saranno controllati dagli altri moduli del programma, ad esempio la regola semantica associata al comando `LeftDiscrim`:

```
statement:

    (...)

    | C_LFT_DISCR { /* Comando LeftDiscrim */
            lftdiscrim = TRUE;
            $$ = $1;
        }
```

imposta il flag `leftdiscrim` che sarà controllato dal modulo di interfaccia utente ogni volta che l'utilizzatore del programma preme il tasto  per sapere se l'operatore corrente debba essere letto o meno.

`statement` → `C_MAXCPX C_INTEGER`
`statement` → `C_SYNTHPAUSE C_INTEGER`
Queste regole riguardano i comandi di configurazione che necessitano di un singolo parametro numerico: in questo caso `MaxComplexity` e `SynthPause`.

Le azioni semantiche loro associate impostano la variabile globale corrispondente al parametro scelto al valore specificato, per esempio:

```
statement:

        (...)

    | C_SYNTHPAUSE C_INTEGER { /* Pausa dopo l'output */
            synthpause = $2;
            $$ = $1;
        }
```

`statement` → `C_EDITCMD C_STRING`
`statement` → `(C_PAUSAB | PAUSA_L) C_STRING`
Queste regole sono analoghe alle precedenti ma hanno a che fare coi comandi che necessitano di una stringa come parametro: `EditCommand`, `PausaBreve` e `PausaLunga`.

Le regole grammaticali viste fino a qui hanno a che fare con le opzioni di personalizzazione di TESI; le regole seguenti invece sono utilizzate per specificare la sintassi delle istruzioni utilizzate per definire i comandi TEX riconosciuti dal programma.

`statement` → `C_NULL C_STRING`
Usati per definire comandi TEX che non necessitano di specificare il modo in cui vanno letti, la stringa rappresentata da `C_STRING` indica il comando TEX che viene definito.

```
InizioMathMode "\(";
```

ad esempio, è un tipo di comando che usa questa sintassi.

Come abbiamo detto sopra, al simbolo `C_NULL` è associato un attributo di tipo `tok` che contiene il valore che il parser del file TEX associa alla categoria lessicale cui appartiene il comando che viene definito da questa istruzione. Continuando col nostro esempio, quando lo scanner del file di configurazione incontra la stringa "`InizioMathMode`", ritorna al parser il valore corrispondente al simbolo terminale `C_NULL`, nel contempo imposta l'attributo associato a tale simbolo al valore `T_MMBEGIN` che, come vedremo, segnala al parser del file TEX un token di "inizio TEX math mode". Abbiamo già spiegato come questo valore sarà poi utilizzato dallo scanner dl file TEX. Ecco il frammento del file `CFGSCAN.L` che riconosce il comando `InizioMathMode` e che esegue le azioni appena

descritte:

```
"InizioMathMode" {

        /* yycfg_lval è l'attributo associato all'espressione regolare
        riconosciuta */
        yycfg_lval.tok = T_MMBEGIN;

        return C_NULL;
}
```

`statement` → `C_STR C_STRING ':' C_STRING`
Qui definiamo comandi che necessitano di un parametro stringa. Un esempio di questi comandi è:

```
Lettera "\alpha" : "alfa";
```

dove, oltre a definire il comando TEX `\alpha`, ne viene indicata la modalità di lettura ("alfa").

`statement` → `C_STR_OPTSTR C_STRING ':' C_STRING [',' C_STRING]`
I comandi che obbediscono a questa sintassi differiscono dai precedeenti in quanto è possibile specificare un secondo parametro stringa opzionale. Un esempio è costituito dal comando:

```
OperatoreEsteso "\sum" : "sommatoria", "sommatoria complessa";
```

dove l'ultima stringa, opzionale, indica cosa leggere nel caso di una sommatoria complessa.

`statement` → `C_STR_LFTSTR_OPTSTR C_STRING ':' C_STRING [C_TOLEFT ','`
`C_STRING] [',' C_STRING]`
Questi comandi hanno una sintassi analoga ai precedenti solo che è possibile che compaia al parola chiave `toleft` (rappresentata dal simbolo terminale `C_TOLEFT`) seguita da una stringa. I comandi di questo tipo sono quelli che definiscono gli operatori, ad esempio:

```
Relazione "\geq": "maggiore uguale" toleft "minore uguale",
    "diseguaglianza complessa";
```

4.3.3 Funzioni

Abbiamo visto come le azioni semantiche associate a comandi di personalizzazione di TESI si limitino ad impostare alcune variabili globali ai valori desiderati dall'utente.

Invece, le azioni semantiche associate a istruzioni che definiscono i comandi TEX da riconoscere, chiamano la funzione `addtoken()` così definita:

```
static void addtoken (tokenval, name, desc, descleft, desclong)

int tokenval; /* Codice del token da restituire a yytex_parse() */
char *name;   /* Nome del token es. "+" o "\sum" */

/* Stringhe che indicano cosa leggere */
char *desc;
char *descleft, *desclong;
```

La funzione `addtoken` crea un nuovo entry nella tabella dei comandi TEX riconosciuti (la cui testa è puntata da `symtable`) inizializzandone opportunamente i campi in base ai valori dei suoi argomenti. Tali argomenti sono settati dalle azioni semantiche attivate dopo il riconoscimento di un comando nel file di configurazione e contengono il valore degli attibuti associati ai simboli `C_STRING` che seguono il comando in questione.

Facciamo un esempio. Supponiamo che il file di configurazione contenga il comando:

```
ParentesiAperta "(": "aperta tonda";
```

questa riga viene riconosciuta dalla regola:

```
statement:

            (...)

    | C_STR C_STRING ':' C_STRING {
            addtoken ($1, $2, $4, NULL, NULL);
            $$ = $1;
        }
```

in quanto il parser, individuata la stringa "`ParentesiAperta`", ritorna all'analizzatore sintattico il valore `C_STR` associando a questo simbolo terminale il valore `T_OPENING`; i due simboli `C_STRING` che compaiono nella regola avranno attributi inizializzati rispettivamente con le stringhe "`(`" e "`aperta tonda`". Quando la regola viene attivata essa invoca la funzione `addtoken` passandole i valori appena indicati, la funzione costruirà dunque un entry nella tabella dei comandi TEX riconosciuti per il simbolo "`(`" salvando queste informazioni.

Le altre due funzioni definite in questo file sono:

```
void freetable()
```

che si occupa di liberare la memoria allocata per la tabella dei comandi TEX riconosciuti.

```
int cfgparse (fname)

char fname[]; /* Nome del file di configurazione */
```

che si occupa di redirigere l'input sul file specificato e di eseguire alcune inizializzazioni prima di chiamare `yycfg_parse()`. Questa funzione costituisce l'intefaccia fra il parser del file di configurazione e le altre funzioni.

4.4 Il le `CFGSCAN.L`

Questo file contiene le specifiche `flex` per lo scanner del file di configurazione.

La sua struttura è piuttosto semplice essendo costituito in gran parte da una sequenza di espressioni regolari che identificano le parole chiave usate nei file di configurazione; abbiamo visto poco sopra un esempio di tali regole.

Altre regole specificate in questo file riguardano rispettivamente il riconoscimento dei numeri interi, che possono essere specificati come parametri di alcuni comandi:

```
[0-9]+ { /* Numero intero */

        /* Conversione stringa->numero intero */
        yycfg_lval.integer = atoi(yycfg_text);

        return C_INTEGER;
    }
```

ed il riconoscimento di parametri stringa:

```
 \"[^\"\n]*\" { /* Stringa fra doppi apici */
        int i;

        /* Riserva spazio in memoria per la stringa */
        yycfg_lval.str = (char *)my_malloc( yycfg_leng - 1);

        /* Copia la stringa eliminando le virgolette */
        for (i=1; i<yycfg_leng-1; i++)
            yycfg_lval.str[i-1] = yycfg_text[i];
        yycfg_lval.str[i-1] = '\0';

        return C_STRING;
    }
```

Un'ultima cosa degna di nota riguarda la gestione del meccanismo di inclusione file:

```
"Input"[ \t]+ {
        BEGIN (Include);
    }
```

Vediamo che non appena viene incontrato il comando `Input`, che specifica l'inclusione di un file, lo scanner viene fatto entrare nella modalità `Include` (tramite la macro

BEGIN()) in cui si aspetta di trovare una stringa che specifica il nome del file di input.

```
<Include>[^ \t\n]+ { /* Legge il filename */

        char s[FNAMELEN+80];

        if ( stack_ptr >= MAX_INCLUDE_DEPTH ) {

            (...) /* Errore: stack overflow */

        } else {

            if ( (yycfg_in = fopen (yycfg_text, "r")) == NULL ) {

                (...) /* Errore: file non trovato */

            } else {

                /* Salva il contesto attuale sullo stack */
                strcpy (include_stack[stack_ptr].name, cfgfname);
                include_stack[stack_ptr].line = cfgfline;
                cfgfline = 1;
                include_stack[stack_ptr].buffer = YY_CURRENT_BUFFER;
                ++stack_ptr;

                /* Ridirige l'input sul nuovo file */
                strcpy (cfgfname, yycfg_text);
                yycfg__switch_to_buffer (
                    yycfg__create_buffer (yycfg_in, YY_BUF_SIZE)
                );
            }
        }

        /* Ritorna alla normale modalita' di funzionamento */
        BEGIN (INITIAL);
    }
```

Quando viene riconosciuto il nome del nuovo file da aprire lo scanner salva nella struttura `include_stack[]` la situazione relativa al file corrente di input. Tale situazione comprende: il buffer associato al file di input rappresentato dalla macro `YY_CURRENT_BUFFER`, messa a disposizione da `flex`, il nome del file (`cfgname`) ed il numero della linea attualmente in esame (`cfgline`). Una volta che la situazione corrente è stata salvata, lo scanner apre il nuovo file e vi ridirige l'input mediante la chiamata ad alcune funzioni rese disponibili da `flex`.

Quando incontra la fine del file di input lo scanner esamina il contenuto dello stack: se non è vuoto significa che deve essere terminato l'esame di alcuni file che erano stati accantonati sullo stack a seguito di una direttiva `Input`, il contesto viene quindi

ripristinato e l'esame dei vecchi file prosegue. Invece, se lo stack è vuoto, viene segnalata la fine dell'input al parser mediante chiamata alla funzione `yyterminate()`.

```
<<EOF>> {

        if ( --stack_ptr < 0 ) {

            /* Stack vuoto: fine dell'input */
            stack_ptr = 0;
            yyterminate();

        } else {

            /* Altrimenti ripristina il contesto precedente */
            yycfg__delete_buffer (YY_CURRENT_BUFFER);
            yycfg__switch_to_buffer (include_stack[stack_ptr].buffer);
            strcpy (cfgfname, include_stack[stack_ptr].name);
            cfgfline = include_stack[stack_ptr].line;
        }
    }
```

Capitolo 5

Elaborazione del file TEX in input

Questo modulo esegue l'analisi del file TEX in input e costruisce gli alberi sintattici delle espressioni matematiche ivi contenute. Contemporaneamente il file TEX viene salvato in memoria per poter essere visualizzato nella successiva fase di esplorazione delle formule.

5.1 File sorgenti

Ecco un elenco dei file che compongono questo modulo:

- `TEXDEF.H` — File con la dichiarazione dei tipi di dati usati in questo modulo, nonché delle funzioni e delle variabili globali.
- `TEXSCAN.L` — Sorgente `flex` per lo scanner del file TEX.
- `TEXSCAN.C` — File generato da `flex` che contiene l'analizzatore lessicale del file TEX.

  Lo scanner vero e proprio è costituito dalla funzione `yytex_scan()`.
- `TEXPARSE.Y` — Il file sorgente per Bison che contiene la grammatica per le espressioni matematiche scritte in TEX.
- `TEXPARSE.C` — Il file generato da Bison che contiene l'analizzatore sintattico per il file TEX.

  Il parser vero e proprio è costituito dalla funzione `yytex_parse()`.
- `TEXPARSE.H` — Anche questo file è generato da Bison e contiene i valori associati ai simboli terminali della grammatica.
- `CRIPPLE.C` — Questo file contiene un insieme di funzioni che eseguono una riorganizzazione degli alberi sintattici, tale riorganizzazione ha lo scopo di renderne la struttura più facilmente esplorabile da parte dell'utente.
- `CRIPPLE.H` — Header per il file `CRIPPLE.C`.

5.2 Il le `TEXDEF.H`

Questo file definisce i tipi di dati utilizzati in questo modulo per creare due strutture dati distinte: l'albero sintattico delle formule matematiche ed una lista che contiene le righe del file TEX in input.

La struttura `TNODE` descrive un nodo dell'albero sintattico:

```
typedef struct tnode {
    int token;
    TNODEUNION nodeunion;
    struct tnode *prec;
    struct tnode **operand;
    int allocated;
    struct tnode *apice, *pedice;
    struct tnode *base;
    int complexity;
    int currop;
    int visitapice, visitpedice;
    int toread;
    int first_line, last_line, first_column, last_column;
} TNODE;
```

Ecco una descrizione dei vari campi della struttura:

- `token` — Questo campo contiene un valore che identifica la categoria lessicale cui appartiene il simbolo contento nel nodo.
- `nodeunion` — Una struttura che contiene informazioni diverse a seconda del token contenuto nel nodo:

  ```
  typedef union tnodeunion {
      char *number;
      char *letter;
      TTOKEN *desc;
  } TNODEUNION;
  ```

  - `number` — Se il nodo descrive un numero questa stringa ne contiene la descrizione.
  - `letter` — Come il campo precedente nel caso però il nodo descriva una lettera.
  - `desc` — In tutti gli altri casi questo campo punta ad un entry della tabella dei comandi TEX riconosciuti che contiene il comando descritto dal nodo corrente.

  Non sempre il campo `nodeunion` punta ad una struttura `TNODEUNION`: per alcuni tipi di token tale struttura è inutile e `tnodeunion` vale `NULL`.
- `prec` — Un puntatore che punta al nodo padre del nodo corrente (il nodo di cui il nodo corrente è operando).
- `operand` — Questo campo punta ad un array di puntatori ai nodi figli del nodo corrente (i nodi operandi del nodo corrente).
- `allocated` — Il numero di operandi del nodo.

- `apice` — Se il nodo possiede un apice questo campo punta alla radice del suo albero sintattico.
- `pedice` — Come il campo precedente ma per il pedice del nodo.
- `base` — Se il nodo fa parte di un sottoalbero che descrive l'apice od il pedice di un nodo, questo campo punta al nodo "base".
- `complexity` — Questo campo contiene la complessità del sottoalbero di cui il nodo è radice.

  La complessità di una formula coincide all'incirca col numero di elementi (variabili, operatori, funzioni, etc.) che la compongono.
- `currop` — Durante la visita dell'albero sintattico di una formula questo campo contiene l'indice dell'operando su cui è posizionato l'utente.

  Questa informazione è utilizzata per tenere traccia del percorso effettuato dall'utente all'interno dell'albero sintattico ed è utilizzata, ad esempio, in modalità Storia.
- `visitapice` e `visitpedice` — Due flag che indicano se l'utente sta visitando rispettivamente l'apice od il pedice del nodo.
- `toread` — Un flag usato durante la fase di sintesi vocale della formula.
- `first_line`, `last_line`, `first_column`, `last_column` — Questi quattro campi contengono la posizione, all'interno del file TEX, del pezzo di formula descritto dal sottoalbero di cui il nodo corrente è radice.

Il restante tipo di dati definito serve per costruire la lista che contiene il file TEX in input, si tratta di una lista doppia in cui ogni elemento contiene una riga del file:

```
typedef struct t_riga {
    char *testo;
    int line;
    struct t_riga *prec, *succ;
} TRIGA;
```

- `testo` — Una stringa che contiene il testo della riga.
- `line` — Il numero della riga.
- `prec`, `succ` — Puntatori agli elementi precedenti e seguenti nella lista.

5.3 Il le `TEXPARSE.Y`

Questo file contiene la grammatica per il file TEX in input nonché una serie di funzioni che effettuano la costruzione dell'albero sintattico delle formule.

5.3.1 Variabili globali

In questo file vengono definite alcune variabili globali usate per contenere gli alberi sintattici delle formule:

```
/*
Array che contiene la radice dell'albero sintattico
delle singole formule
*/
extern TNODE* parsetree[];

/*
Numero di elementi contenuti in parsetree[]
*/
extern int treenum;
```

5.3.2 Grammatica per il le T EX in input

I simboli della grammatica possiedono tutti attributi di tipo `TNODE` che, come abbiamo visto, definisce una struttura che descrive un nodo dell'albero sintattico. Quello che avviene in pratica è che ogni token è associato ad un nodo dell'albero sintattico, durante l'analisi sintattica, mano a mano che i token vengono riconosciuti, le azioni semantiche chiamano le funzioni che provvedono a costruire l'albero sintattico. Quando una formula viene riconosciuta interamente l'albero sintattico relativo è stato costruito e la sua radice viene salvata in `parsetree[]`.

Iniziamo a descrivere la grammatica elencando i simboli terminali.

```
%token T_MMBEGIN T_MMEND
%token T_DMBEGIN T_DMEND
```

Valori associati a token che segnalano rispettivamente inizio e fine del TEX math mode e del TEX display math mode. Questi valori non sono ritornati dallo scanner ma servono per classificare i token contenuti nella tabella dei comandi TEX riconosciuti.

Quando lo scanner incontra un token di questo tipo controlla la variabile globale `readallformula` la quale costituisce un flag, impostato dai comandi `(No)ReadAllFormula`, che indica se vanno lette anche le formule in TEX math mode. Se il token indica una formula scritta in una modalità valida, lo scanner restituisce uno dei seguenti valori:

```
%token T_BEGIN T_END
```

per indicare rispettivamente inizio e fine di una formula.

Altri valori che descrivono il tipo di comandi TEX riconosciuti e che vengono utilizzati nei relativi entry della tabella sono:

```
/* Simbolo da scartare (comando Garbage) */
%token T_GARBAGE

/* Lettera (comando Lettera) */
%token T_LETTERA

/* Simbolo (comando Simbolo) */
```

```
%token T_SIMBOLO

/*
Comando che introduce un blocco di testo
(comando Testo)
*/
%token T_TEXTCMD

/*
Token che rappresentano parentesi aperte e chiuse
(comandi ParentesiAperta e ParentesiChiusa)
*/
%token T_OPENING T_CLOSING

/*
Token che segnalano apici e pedici
(comando Apice e Pedice)
*/
%token T_APICE T_PEDICE

/*
Token che descrive un comando di accentazione
(comando Accento)
*/
%token T_ACCENTO

/* Radice quadrata (comando Radice) */
%token T_RADICE

/*
Token che rappresentano funzioni
(comando Funzione)
*/
%token T_FUNZIONE

/*
Token che descrive un operatore
di tipo sommatoria o produttoria
(comando LargeOperator)
*/
%token T_LARGEOP

/* Token definiti col comando Frazione */
%token T_OVER

/* Token definiti col comando Atop */
%token T_ATOP
```

Altri token riguardano gli operatori:

```
%left  T_SEPARATORE
%left  T_RELAZIONE
%left  T_OPSOMMA T_OPOR
%left  T_OPPROD
%left  T_NULLOP /*Operatore implicito di moltiplicazione */
%right T_OPNOT
%left  T_OPFACT
%right T_OPEXP
```

Questi valori non sono i soli a comparire nel campo `token` di un nodo: alcuni nodi sono introdotti nella struttura dell'albero sintattico per evidenziarne le diverse componenti e non sempre rappresentano token che compaiono nella formula. Questi nodi sono marcati coi seguenti valori:

```
/* Numero */
%token T_NUMERO

/* Il nodo contiene un blocco di testo */
%token T_TESTO

/*  IL nodo contiene un'espressione fra parentesi */
%token T_PARENTESI

/*
Quando viene incontrato un errore sintattico
all'interno di una formula viene creato un nodo
con questo token
*/
%token T_ERROR
```

Un discorso a parte merita il token:

```
%token T_CLUSTER
```

questo valore marca alcuni nodi particolari generati dall'operazione di "potatura" degli alberi sintattici successiva all'analisi sintattica, ritorneremo su questo tipo di nodi più avanti.

Passiamo ora a descrivere le regole della grammatica; il loro elenco, privato dei commenti si trova nell'appendice B.

`input` $\rightarrow \epsilon \mid$ `input formula ';'`
L'input viene definito come una sequenza (eventualmente vuota) di formule.

`formula` $\rightarrow$ `T_BEGIN espressione T_END`
`formula` $\rightarrow$ `T_BEGIN corpo_frazione T_END`
`formula` $\rightarrow$ `T_BEGIN corpo_atop T_END`
`formula` $\rightarrow$ `T_BEGIN error T_END`
Ogni formula è racchiusa fra token `T_BEGIN` e `T_END` che, come detto sopra, delimitano le espressioni matematiche all'interno del file TEX. L'ultima regola implementa un

estremo tentativo di recupero dell'errore: se TeSI non riesce a confinare un errore ai livelli più interni della formula, il parser lo gestisce a questo livello rendendo possibile la risincronizzazione e l'esame delle formule seguenti nel file di input.

```
espressione → espressione T_RELAZIONE espressione
espressione → espressione T_SEPARATORE espressione
espressione → espressione T_OPSOMMA espressione
espressione → espressione T_OPOR espressione
espressione → espressione T_OPPROD espressione
espressione → espressione espressione2
espressione → T_OPNOT espressione
espressione → espressione1
espressione1 → T_OPSOMMA espressione1
espressione1 → espressione2
espressione2 → espressione2 T_OPFACT
espressione2 → espressione2 T_OPEXP espressione2
espressione2 → espressione3
```

Queste regole definiscono una `espressione` e riconoscono le formule che contengono degli operatori.

La priorità degli operatori in parte dipende da come questi sono stati definiti: abbiamo visto[1] che la dichiarazione di `T_SEPARATORE` precede qulla di `T_RELAZIONE` quindi `T_RELAZIONE` possiede una priorità minore di `T_SEPARATORE`. La priorità deglio operatori dipende anche da come le regole grammaticali sono state formulate: per riconoscere un'`espressione` il parser dovrà sempre riconoscere prima `espressione2` quindi l'operatore `T_OPEXP` sarà sempre applicato prima di, ad esempio, `T_OPPROD` e quindi avrà priorità maggiore.

Il simbolo `espressione3` rappresenta un livello elementare di espressione corrispondente a termini atomici quali numeri, variabili, funzioni, etc.

```
espressione3 → simbolo
espressione3 → elemento2
espressione3 → large_operator
espressione3 → testo
espressione3 → ACCENTO BEGIN espressione END
espressione3 → ACCENTO BEGIN error END
espressione3 → BEGIN error END
```

Meritano a questo punto una spiegazione i simboli `BEGIN` e `END` che implementano dei kludge[2]. Come abbiamo visto lo scanner deve operare in due modalità diverse: dopo aver incontrato un comando che indica un apice, un pedice od accentazione TeX si aspetta di trovare un singolo carattere che sarà trattato appunto come apice, pedice o testo da accentare, se anziché un singolo carattere si desidera utilizzare un testo più lungo questo va racchiuso fra {...}. Accade dunque che una stessa stringa di caratteri debba essere trattata in due modi diversi a seconda che segua uno di questi comandi o meno[3]. Per implementare queste due diverse metodologie di analisi lessicale TeSI usa le seguenti regole:

[1] Cfr. pag. 64.

[2] Vedi pag. 44

[3] La stringa `12` dev'essere trattata come un'unica sequenza numerica nell'espressione `$ x+12 $` ma va spezzata in due nella formula `$ x^12 $` che genera l'espressione: "x^12".

token_tex → token_tex_numerico
token_tex → token_tex_non_numerico
Dove token_tex indica o un singolo carattere od un'espressione fra parentesi graffe.

token_tex_numerico → TeXtrue tokenn TeXfalse
token_tex_numerico → TeXtrue BEGIN token_tex_numerico END TeXfalse
tokenn → T_NUMERO
tokenn → BEGIN T_NUMERO END
token_tex_numerico rappresenta una singola cifra od un numero fra parentesi graffe.

token_tex_non_numerico → TeXtrue token1 TeXfalse
token_tex_non_numerico → TeXtrue BEGIN token_tex_non_numerico END
TeXfalse
token1 → T_SIMBOLO
token1 → lettera
token1 → frazione
token1 → costrutto_atop
token1 → BEGIN espressione END
token1 → BEGIN error END
token_tex_non_numerico rappresenta invece una qualsiasi altra espressione costituita da un singolo carattere o da più caratteri fra graffe.

Le parentesi graffe sono gestite dalle seguenti regole:

BEGIN → '{' TeXfalse
END → '}'

Il diverso trattamento dell'input è determinato dal seguente pezzo di codice Bison:

```
TeXtrue
    : {
            expectTeXtoken = TRUE;
        }
;
TeXfalse
    : {
            expectTeXtoken = FALSE;
        }
;
```

Come si vede i simboli TeXtrue e TeXfalse in realtà sono definiti con le regole:

TeXtrue → ϵ
TeXfalse → ϵ
che possono sempre essere applicate quando una regola contenente questi simboli viene attivata. Il risultato netto è che ogni volta che il parser tenta di riconoscere il simbolo token_tex viene chiamata una regola che utilizza i simboli TeXtrue e TeXfalse i quali settano opportunamente la variabile globale expectTeXtoken; lo scanner, ogni volta che riconosce un token, controlla questa variabile e decide se restituire al parser l'intero token o solo il primo carattere.

Infine la regola:

`ACCENTO → T_ACCENTO TeXtrue`
assicura che il trattamento riservato per apici e pedici sia applicato anche ai comandi di accentazione.

Vediamo ora quali sono gli altri elementi che compongono le espressioni e la cui definizione è rimasta in sospeso.

`testo → T_TEXTCMD T_TESTO`
Regola per i comandi che introducono del testo nelle formule (es. `\mbox{}`).

```
simbolo → T_SIMBOLO
simbolo → simbolo T_APICE token_tex
simbolo → simbolo T_PEDICE token_tex
simbolo → ACCENTO T_SIMBOLO TeXfalse
simbolo → ACCENTO BEGIN simbolo END
```
Simboli (definiti col comando `Simbolo`) eventualmente con apici, pedici ed accenti.

```
lettera → T_LETTERA
esp_letterale → lettera
esp_letterale → esp_letterale T_APICE token_tex
esp_letterale → esp_letterale T_PEDICE token_tex
esp_letterale → ACCENTO lettera TeXfalse
esp_letterale → ACCENTO BEGIN esp_letterale END
```
`esp_letterale` indica una lettera (definita col comando `Lettera` nel file di configurazione) eventualmente con, apici, pedici ed accenti.

```
esp_numerica → T_NUMERO
esp_numerica → BEGIN T_NUMERO T_OVER T_NUMERO END
esp_numerica → T_RADICE token_tex_numerico
esp_numerica → esp_numerica T_APICE token_tex
esp_numerica → esp_numerica T_PEDICE token_tex
esp_numerica → ACCENTO token_tex_numerico
esp_numerica → ACCENTO BEGIN esp_numerica END
```
`esp_numerica` rappresenta un numero con apici e pedici. Anche una frazione con numeratore e denominatore puramente numerici e la radice di un numero rientrano in questa categoria, come vedremo questo rende più agevole definire gli argomenti di funzione.

```
esp_parentesi → T_OPENING espressione T_CLOSING
esp_parentesi → esp_parentesi T_APICE token_tex
esp_parentesi → esp_parentesi T_PEDICE token_tex
esp_parentesi → ACCENTO BEGIN esp_parentesi END
esp_parentesi → T_OPENING error T_CLOSING
```
Qui vengono definite le espressioni fra parentesi. L'ultima regola permette la gestione degli errori che si verificano all'interno di parentesi, in questo caso l'espressione fra parentesi non è riconosciuta, ma non è compromesso il riconoscimento della formula in cui essa è inserita.

corpo_frazione → espressione T_OVER espressione
corpo_frazione → T_NUMERO T_OVER espressione
frazione → BEGIN corpo_frazione END
frazione → frazione T_APICE token_tex
frazione → frazione T_PEDICE token_tex
frazione → ACCENTO BEGIN corpo_frazione END
Definizione di frazione.

corpo_atop → espressione T_ATOP espressione
costrutto_atop → BEGIN corpo_atop END
costrutto_atop → costrutto_atop T_APICE token_tex
costrutto_atop → costrutto_atop T_PEDICE token_tex
costrutto_atop → ACCENTO BEGIN corpo_atop END
Idem per i costrutti di tipo \atop.

radice → T_RADICE token_tex_non_numerico
radice → ACCENTO BEGIN radice END
Radici quadrate.

funzione → corpo_di_funzione argomento_di_funzione
corpo_di_funzione → T_FUNZIONE
corpo_di_funzione → corpo_di_funzione T_APICE token_tex
corpo_di_funzione → corpo_di_funzione T_PEDICE token_tex
corpo_di_funzione → ACCENTO T_FUNZIONE TeXfalse
corpo_di_funzione → ACCENTO BEGIN corpo_di_funzione END
argomento_di_funzione → sequenza1
argomento_di_funzione → elemento1
argomento_di_funzione → sequenza1 elemento1
argomento_di_funzione → funzione
Qui viene definita la sintassi per le funzioni. Il simbolo sequenza1 indica una categoria di termini che possono comparire più di una volta in un argomento di funzione:

elemento_sequenza1 → esp_numerica
elemento_sequenza1 → esp_letterale
sequenza1 → elemento_sequenza1
sequenza1 → sequenza1 elemento_sequenza1
ad esempio l'espressione:

$$\sin \frac{1}{2}\alpha$$

può venire riconosciuta da queste regole (si noti che questo è reso possibile anche dal fatto che si è dichiarato che esp_numerica può essere costituita da una frazione con numeratore e denominatore numerici).

elemento1 → esp_parentesi
elemento1 → frazione
elemento1 → costrutto_atop
elemento1 → radice

`elemento1` invece rappresenta token che possono comparire una sola volta come argomento di funzione, pena un'espressione ambigua che sarebbe meglio esplicitare con l'uso di parentesi.

`elemento2` → `elemento_sequenza1`
`elemento2` → `elemento1`
`elemento2` → `funzione`
`sequenza2` → `elemento2`
`sequenza2` → `sequenza2 elemento2`
`sequenza2` ed `elemento2` ripropongono quanto appena detto per i "large operator", ossia espressioni come sommatorie o produttorie.

`large_operator` → `corpo_di_operatore argomento_di_operatore`
`corpo_di_operatore` → `T_LARGEOP`
`corpo_di_operatore` → `corpo_di_operatore T_APICE token_tex`
`corpo_di_operatore` → `corpo_di_operatore T_PEDICE token_tex`
`corpo_di_operatore` → `ACCENTO T_LARGEOP TeXfalse`
`corpo_di_operatore` → `ACCENTO BEGIN large_operator END`
`argomento_di_operatore` → `sequenza2`
La definizione di "large operator" chiude questa sezione sulla grammatiche del file TEX.

5.3.3 Funzioni

La maggior parte delle funzioni definite nel file `TEXPARSE.Y` si occupa della costruzione degli alberi sintattici delle formule: queste funzioni sono chiamate dalle azioni semantiche associate alle regole viste sopra. La loro struttura è piuttosto simile, quindi, a titolo di esempio, forniremo una descrizione dettagliata solo della prima rimandando il lettore al listato del programma per una descrizione completa delle altre funzioni.

```
static TNODE *insertbinop (binop, operand1, operand2)

TNODE *binop; /* Nodo che descrive un operatore binario */
TNODE *operand1, *operand2; /* Nodi per i due operandi */

{
    /* Riserva memoria per i puntatori agli operandi del nodo binop */
    binop->operand = (TNODE**)my_malloc( sizeof(TNODE*) * 2 );

    /* Lega gli operandi al nodo */
    binop->allocated = 2;
    binop->operand[0] = operand1;
    operand1->prec = binop;
    binop->operand[1] = operand2;
    operand2->prec = binop;

    /* Aggiorna la complessità del nodo */
    binop->complexity += (operand1->complexity + operand2->complexity);

    /*
    Tiene traccia della posizione della subformula individuata
    dal nodo binop all'interno del file TeX
    */
    binop->first_line = min3 (binop->first_line, operand1->first_line,
                              operand2->first_line);
    binop->first_column = min3 (binop->first_column,
                                operand1->first_column,
                                operand2->first_column);
    binop->last_line = max3 (binop->last_line, operand1->last_line,
                             operand2->last_line);
    binop->last_column = max3 (binop->last_column,
                               operand1->last_column,
                               operand2->last_column);

    /* Ritorna un puntatore al nodo così creato */
    return binop;
}
```

Questa funzione lega l'operatore `binop` ai nodi `operand1` ed `operand2` che costituiscono i suoi operandi. Ecco un esempio di regola sintattica che fa uso di questa funzione:

```
espressione

    (...)

    | espressione T_OPSOMMA espressione {
            $$ = insertbinop ($2, $1, $3);
        }
```

In pratica, quando una regola individua un operatore binario, chiama la funzione

`insertbinop()` indicando l'operatore ed i suoi due operandi, la funzione si incarica di inizializzare correttamente i puntatori contenuti all'interno delle strutture `TNODE` che costituiscono i nodi dell'albero sintattico associati a questi token. Tali strutture, come vedremo, vengono create dallo scanner non appena i token in questione sono stati riconosciuti.

```
static TNODE *addnullop (operand1, operand2)

TNODE *operand1, *operand2; /* Nodi per i due operandi */
```

Questa funzione è analoga alla precedente ma viene invocata per l'operatore implicito di moltiplicazione (come nell'espressione "$2x$" o "xy"). A differenza della funzione precedente qui deve venire riservata memoria per il nodo che costituisce l'operatore implicito.

```
TNODE *insertuniop (uniop, operand)

TNODE *uniop; /* Nodo che si riferisce all'operatore unario */
TNODE *operand; /* Nodo che descrive l'operando */
```

Analoga alle funzioni precedenti ma per gli operatori unari (come nelle espressioni "-1" o "$x!$").

```
static TNODE* addnum (number, line, column)

char *number; /* Stringa che descrive il numero */
int line, column; /* Posizione del token nel file TeX */
```

Questa funzione crea un nodo che descrive un numero a partire da una stringa che ne contiene la descrizione.

```
static TNODE* addparenthesis (opening, closing, formula)

TNODE *opening, *closing; /* Nodi per le parentesi aperte e chiuse */
TNODE *formula; /* Nodo con l'espressione fra parentesi */
```

Quando viene riconosciuta un'espressione fra parentesi, viene creato un nodo di tipo `T_PARENTESI` che ha come figli il nodo che descrive la parentesi aperta, il nodo radice del sottoalbero con l'espressione fra parentesi ed il nodo che descrive la parentesi chiusa. Questa organizzazione dell'albero facilita la successiva lettura dell'espressione.

```
static TNODE *insertapici (node, apice, pedice)
TNODE *node, *apice, *pedice;

static void setbase (node, base)
TNODE *node, *base;
```

Queste due funzioni sono usate per la costruzione di apici e pedici di un nodo: la funzione `insertapici` inserisce i nodi `apice` e `pedice` rispettivamente come apici e pedici di `node`; la funzione `setbase` invece visita l'intero sottoalbero puntato da `node`

e imposta il campo `base` di ogni nodo al valore `base`. In questo modo, quando si sta visitando un nodo del sottoalbero `node` e si preme il tasto **Home**, la visita passa al nodo puntato da `base`, che costituisce appunto il termine di cui il sottoalbero è apice o pedice. `setbase` viene chiamata da `insertapici`.

```
static TNODE *adderror (delimiter1, delimiter2)

TNODE *delimiter1, *delimiter2;
```

Quando viene riscontrato un errore durante l'analisi sintattica viene invocata questa funzione la quale crea un nodo di tipo `T_ERROR` che viene inserito nell'albero sintattico della formula per rappresentarne la parte che ha generato l'errore. `delimiter1` rappresenta l'ultimo token accettato dall'analizzatore sintattico prima del verificarsi dell'errore, `delimiter2` è il primo token riconosciuto con successo dopo che il parser ha recuperato la condizione di errore. Questi token saranno utilizzati per delimitare la porzione del testo che ha generato l'errore sintattico all'interno del file TEX.

```
texparse (finname)

char *finname; /* Nome del file TeX */
```

Questa funzione, prima di chiamare il parser, si preoccupa di inizializzare alcune variabili e di reindirizzare l'input sul file TEX scelto dall'utente: essa costituisce l'interfaccia fra la funzione `yytex_parse()`, che costituisce il parser vero e proprio, ed il resto del programma.

5.4 Il le `TEXSCAN.L`

Questo file costituisce il sorgente `flex` dello scanner per il file TEX in input. La struttura di questo scanner è piuttosto complessa, date le diverse modalità con cui viene gestito il file di input. Prima di esaminarla vediamo le variabili globali definite in questo file.

Innanzitutto ci sono le variabili che saranno utilizzate per memorizzare il file TEX in input:

```
/*
Puntatore alla testa della lista che contiene il file TeX
*/
extern TRIGA* inputtext;

/*
Numero di righe contenute nella lista
*/
extern int inputrows;
```

Seguono alcune variabili utilizzate durante l'analisi lessicale del file:

```
/*
Puntatore all'ultima riga inserita nella lista
che contiene il testo del file TeX
*/
```

```
static TRIGA* lastrow = NULL;

/*
Colonna e riga attuale all'interno del file TeX
*/
static int filecolumn = 1;
static int fileline = 1;

/*
"Buffer di riga" che contiene la riga corrente
*/
static char *buffer = NULL;
static unsigned buffdim = 0; /* Dimensione totale del buffer */
static unsigned buffused = 0; /* Caratteri gia' nel buffer */
```

Merita un discorso a parte questo buffer, che chiameremo "buffer di riga", dove viene salvata la porzione della linea corrente già sottoposta all'analisi lessicale. A mano a mano che il parser riconosce dei token all'interno di una riga questi vengono salvati in `buffer`. Per rendere più veloce questa operazione, quando è necessario ingrandire `buffer`, questo viene ridimensionato di misura leggermente superiore al necessario in modo da poter coprire successive richieste senza bisogno di riservare nuova memoria. `buffdim` contiene la dimensione totale del buffer mentre `buffused` contiene il numero di caratteri salvati al suo interno.

La funzione `salvastringa()` si occupa di inserire una stringa nel buffer di riga:

```
static void salvastringa (s)

char *s; /* Stringa da mettere nel buffer di riga */

{
    int slen;
    char *tmp;

    slen = strlen (s);
    if ( (buffer==NULL) || (buffused+slen > buffdim) ) {

        /* Riserva nuova memoria */
        tmp = (char*)my_malloc( sizeof(char) * (buffused+slen+20) );
        buffdim = buffused + slen + 10;

        (...)

    }

    /* Vi copia la stringa */
    strcat (buffer, s);
    buffused += slen;
}
```

Quando viene incontrata la fine di una riga, quest'ultima viene salvata all'interno della lista che contiene il file TEX in input mediante chiamata alla funzione:

```
static void salvariga ()
```

Abbiamo detto che il salvataggio del testo nel buffer deve avvenire ogni qualvolta viene riconosciuto un token; per automatizzare questa operazione basta definire la macro:

```
#define YY_USER_ACTION parse();
```

In questo modo, ogni volta che una espressione regolare viene riconosciuta, prima di eseguire le azioni semantiche ad essa associate `flex` invoca la funzione `parse()`. Questa funzione si occupa di salvare il testo del token nel buffer di riga; inoltre aggiorna la struttura `yylloc`[4] che contiene la posizione del token all'interno del file di input. Questa informazione, salvata dal parser negli opportuni campi della struttura `TNODE` associata al token, è utilizzata nella successiva fase di esplorazione della formula.

```
static void parse ()

{
    /* Salva il testo del token nel buffer di riga */
    salvastringa (yytex_text);

    /*
    Tiene traccia della posizione del token
    all'interno del file TeX
    */
    yylloc.first_line = yylloc.last_line = fileline;
    yylloc.first_column = filecolumn;
    filecolumn += strlen (yytex_text);
    yylloc.last_column = filecolumn - 1;
}
```

Segue la parte di definizioni ove sono dichiarate alcune espressioni regolari di ovvio significato:

```
LETTERA     [a-zA-Z]
NON_LETTERA [^a-zA-Z]
COMANDO_TEX \\({LETTERA}+)|\\{NON_LETTERA}
CIFRA       [0-9]
INTERO      {CIFRA}+
REALE       {INTERO}|{CIFRA}*"."{INTERO}
```

Infine inizia la parte che contiene le espressioni regolari con le azioni semantiche loro associate. Questa sezione è appesantita da una serie di regole necessarie per individuare commenti all'interno del file TEX: in questa sede eviteremo di esaminare queste regole, cercando di dare invece un'idea generale sul funzionamento dello scanner.

L'analizzatore lessicale può trovarsi in diverse modalità di funzionamento, le principali sono:

1. `INITIAL` — Questa è la modalità di default per lo scanner: in questa modalità esso esamina la parte di file TEX che non è relativa ad alcuna formula.

[4] Vedi pag. 45.

2. `MathMode` — In questa modalità lo scanner sta esaminando il testo di una formula.

3. `TextMode` — Lo scanner si trova in questa modalità quando incontra del testo all'interno di una formula. Questo avviene quando si trovano dei comandi (come `\mbox{}`) utilizzati per inserire commenti nelle formule.

Nella modalità `INITIAL` lo scanner individua i comandi TEX; se incontra un comando di inizio TEX (display) math mode, passa alla modalità `MathMode`. Le azioni semantiche fanno uso della routine `checkcmd()` la quale ritorna il tipo di un token dopo aver esaminato gli entry della tabella dei comandi TEX riconosciuti:

```
static int checkcmd (cmd)

char *cmd; /* Token da riconoscere */

{
     TSYMTAB *ptr;
     TNODE *nodeptr;
     char s[80];

    /* Scorre la tabella dei comandi TeX riconosciuti */
    for (ptr = symtable;
        (ptr != NULL) && (strcmp(ptr->entry->name, cmd) != 0);
        ptr = ptr->next)
    ;

    if (ptr != NULL) { /* Trovato il comando nella symtable */

        /*
        Riserva un nodo per il token e
        lo inizializza opportunamente
        */
        nodeptr = allocatenode();

        (...)

        yytex_lval = nodeptr;
        return nodeptr->token;

    } else { /* Comando non trovato */

        if ( strlen (cmd) == 1) {

            /*
            Il token estratto e' un singolo carattere:
            ne restituisce il codice ASCII
            */
            yytex_lval = NULL;
            return ( cmd[0] );

        }  else {

            /* Token non riconosciuto */
            yytex_lval = NULL;
            return T_ERROR;
        }
    }
}
```

Ecco le principali regole utilizzate in modalità `INITIAL`:

```
\$\$|\$ {
    TNODE *nodeptr;

    /*
    Lo scanner controlla che sia dichiarato l'inizio del TeX
    display math mode o che l'utente abbia usato il comando
    ReadAllFormula nel file di configurazione
    */
    if ( (strcmp(yytex_text, "$$") == 0) || readallformula ) {

       BEGIN (MathMode);

       (...)

    }
}

{COMANDO_TEX} {
    int token;

    /* Trova il tipo di token associato a COMANDO_TEX */
    token = checkcmd (yytex_text);

    /* Come sopra */
    if ( (token == T_DMBEGIN) ||
         ( (token == T_MMBEGIN) && readallformula)
       ) {

        BEGIN (MathMode);
        return T_BEGIN;

    } else

        (...)

}

\n {
    /*
    Alla fine di ogni riga salva il contenuto del buffer
    nella lista col testo del file TeX in input
    */
    salvariga();
}

. { /* Testo normale */ }
```

Bisogna notare che lo scanner non segnala nessun token all'analizzatore sintattico (ad eccezione dei comandi di inizio TEX (display) math mode), nella modalità `MathMode`

invece lo scanner, avvalendosi dei valori ritornati da `checkcmd()`, costruisce un nodo per ogni token individuato e ritorna al parser il tipo di token riconosciuto:

```
<MathMode>[ \t] {/* Salta gli spazi*/}

<MathMode>\n { /* Nuova riga */
    salvariga();
}

<MathMode>{REALE}|{REALE}[eE]{INTERO}|{REALE}[eE][+-]{INTERO} {
    /* Numero */

    TNODE *tmp;

    /*
    Se è stato individuato un comando di apice, pedice
    od accentazione allora viene restituito solo il primo
    carattere del numero
    */
    if ( (expectTeXtoken) && (yytex_text[0] == '.') ) {

        /*
        "Rigetta" parte del testo riconosciuto: il testo
        verrà riprocessato alla prossima chiamata dello scanner
        */
        cancellastringa (yytex_leng-1);
        yyless (1);

        /* In questo caso il numero iniziava con "." (es. .10) */
        return ( checkcmd (".") );

    } else {

        /* Costruisce un nodo per il numero */
        tmp = allocatenode();

        (...)

    }

    /* Ritorna il token al parser */
    yytex_lval = tmp;
    return T_NUMERO;
}
```

Questa regola fa uso delle funzioni `cancellastringa()` e `yyless()`: lo scopo di queste funzioni è di ritornare parte del testo già processato al buffer di input, questa azione si rende necessaria quando lo scanner processa un testo la cui lunghezza eccede quella del token effettivamente riconosciuto. In questo caso viene processata una stringa numerica di cui però solo il primo carattere va passato al parser (come già detto questa

situazione si verifica quando si incontrano comandi che introducono apici o pedici); la restante parte della stringa viene quindi "rigettata" e sarà riprocessata alla successiva chiamata dello scanner.

La funzione `cancellastringa()` toglie dal buffer di riga la parte di testo che va riprocessato mentre la funzione `yyless()`, fornita da `flex`, rimette questo testo nel buffer di input.

Le restanti regole si preoccupano di analizzare i comandi TEX incontrati nelle formule:

```
<MathMode>\$\$|\$ { /* Fine della formula */
    BEGIN (INITIAL);


    (...)
}

<MathMode>{COMANDO_TEX}|. {
    int tmp;
    char s[80];

    switch ( tmp = checkcmd (yytex_text) ) {

        case T_GARBAGE: /* Token da ignorare */
            freenode (yytex_lval);
            break;

        case T_DMEND: /* Fine della formula */
        case T_MMEND:
            BEGIN(INITIAL);
            return T_END;
            break;

        case T_TEXTCMD: /* Testo nella formula */
            BEGIN(InizioTextMode);
            return T_TEXTCMD;
            break;

        /* In tutti gli altri casi segnala il token */
        default:
            return tmp;
            break;
    }
}
```

La modalità `TextMode` non è particolarmente interessante: semplicemente il testo racchiuso fra parentesi graffe, che costituisce il parametro del comando di inserimento testo (es. `\mbox{Commento}`), viene salvato in una stringa, dopodiché lo scanner torna nella modalità `MathMode`.

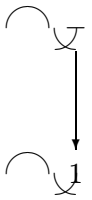

Figura 5.1: Albero sintattico per l'espressione "-1".

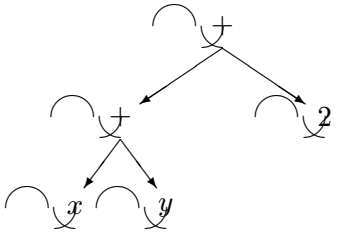

Figura 5.2: Albero sintattico per l'espressione "$x + y + 2$".

5.5 Il file `CRIPPLE.C`

Questo file contiene alcune routine che eseguono una riorganizzazione dell'albero sintattico al fine di rendere la successiva fase di esplorazione più semplice.

Il lavoro di riorganizzazione attuato è essenzialmente di due tipi:

1. Fusione di operatori unari.

   Espressioni che contengono un operatore unario, come "-1" oppure "$x!$", generano un albero sintattico costituito da due nodi: uno per l'operatore ed uno per l'operando (vedi fig. 5.1). Quando l'operando è di bassa complessità, ad esempio un singolo numero o variabile, questa struttura rallenta l'esplorazione della formula senza aggiungere nulla in chiarezza quindi, durante questa fase, i due nodi vengono fusi in un unico nodo che contiene l'intera espressione.

2. Fusione di operatori simili.

   Consideriamo un'espressione come:

   $$x + y + 2$$

   il cui albero sintattico è indicato in figura 5.2.

   Di fronte ad un'espressione di questo tipo l'utente si aspetterebbe di poter scorrere i termini della somma semplicemente usando i tasti $\square$ e $\boxed{\rightarrow}$ ma questo non è possibile, data la struttura dell'albero sintattico: quando l'utente sta esplorando la radice i tasti cursore gli permettono semplicemente di muoversi fra i nodi che rappresentano rispettivamente le espressioni "$x + y$" e "2".

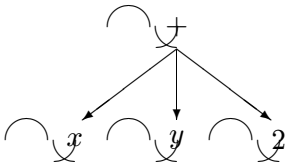

Figura 5.3: Albero sintattico per l'espressione "$x + y + 2$" dopo la fase di riorganizzazione.

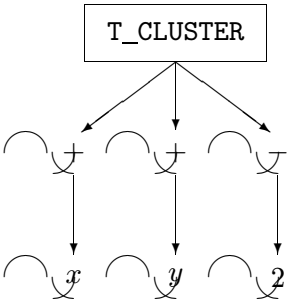

Figura 5.4: Nodo `T_CLUSTER` per l'espressione "$x + y - 2$".

Per permettere all'utente di scorrere facilmente i termini di espressioni di questo tipo TESI riorganizza l'albero sintattico della formula in una forma simile a quella mostrata in figura 5.3.

Questo tipo di riorganizzazione è attuata anche per operatori che non sono uguali ma che hanno la stessa priorità: le formule

$$x + y - 2 \qquad 0 < x \leq y$$

ad esempio, vengono trattate allo stesso modo.

Il file `CRIPPLE.C` contiene le funzioni che attuano questa riorganizzazione degli alberi sintattici: non ci dilungheremo oltre sugli algoritmi usati, diciamo solo che nodi complessi quali l'operatore "+" che compare nella figura 5.3 sono rappresentati da un nodo di tipo `T_CLUSTER` il quale possiede come figli un nodo per ogni operando: tale nodo contiene la descrizione dell'operatore che lega l'operando al resto della formula e possiede come nodo figlio l'operando vero e proprio. La figura 5.4 esemplifica questa struttura.

Capitolo 6

Interfaccia con l'utente

In questo capitolo descriveremo il modulo di interfaccia con l'utente, concentrando la nostra attenzione sulle routine che permettono la visita degli alberi sintattici delle formule.

6.1 File sorgenti

Il modulo si compone di due soli file:

- `VISIT.C` — File che contiene le funzioni di questo modulo.
- `VISIT.H` — Header per il file `VISIT.C`.

6.2 Il le `VISIT.C`

La funzione `visitparsetree()` costituisce l'interfaccia fra le funzioni di questo modulo ed il resto del programma, questa funzione chiama la routine:

```
static int visittree ()
```

la quale implementa la modalità Esplora. Questa funzione si compone di un loop all'interno del quale vengono compiute ciclicamente tre azioni:

1. Viene letto il contenuto del sottoalbero sulla cui radice è posizionato in un dato momento l'utente, contemporaneamente viene visualizzata nella finestra TEX il pezzo di formula corrispondente.

   Questo compito viene svolto dalle funzioni `showtext()`[1] e `readformula()`.

2. Viene letto il tasto premuto dall'utente.

3. Un lungo costrutto `switch` interpreta il tasto premuto ed esegue i comandi relativi.

   In pratica `visittree()` tiene traccia del nodo su cui si trova l'utente e, ogni volta che l'utente preme un tasto opportuno, si sposta su un nuovo nodo dell'albero sintattico.

   Alcuni tasti possono portare TESI in una modalità diversa dalla modalità Esplora: in questo caso verranno chiamate altre funzioni che implementano la nuova modalità di funzionamento.

[1] Vedi par. 7.2.

La lettura della "formula corrente" (quella sulla cui radice è posizionato l'utente) viene fatta dalla funzione:

```
static void readformula (node, force)

TNODE *node; /* Radice della formula */

/*
Flag: se TRUE la formula viene letta interamente,
indipendentemente dalla sua complessita'
*/
int force;

{
    cleantree (node);

    /*
    Confronta la complessità della formula con il
    parametro maxcomplexity, settato dall'utente
    nel file di configurazione
    */
    if ( (node->complexity > maxcomplexity) && (!force) )
        simplify (node, node->complexity);

    speaktree (node, 1);

    (...)
}
```

La funzione `readformula()` chiama innanzitutto la routine `cleantree()` la quale si preoccupa di mettere al valore `TRUE` il campo `toread`[2] di tutti i nodi del sottoalbero puntato da `node`. La funzione `simplify()` esplora ricorsivamente il sottoalbero, se un nodo è troppo complesso per essere letto interamente (il valore del suo campo `complexity`[3] supera il valore della variabile `maxcomplexity`) alcuni dei suoi operandi, ad iniziare dai più complessi, saranno "mascherati" marcando il campo `toread` dei loro nodi col valore `FALSE`:

[2] Vedi pag. 61.
[3] Vedi pag. 61.

```
static int simplify (node, actualcpx)

TNODE *node; /* Nodo corrente */
int actualcpx; /* Complessita' attuale dell'intera formula */

{
    int j, maxcpx, tmpcpx;
    TNODE *tosim; /* Operando da semplificare */

    /* Nodo abbastanza semplice: fine */
    if (actualcpx <= maxcomplexity)
        return actualcpx;

    tmpcpx = actualcpx;

    /* I nodi T_NULLOP vanno semplificati per intero */
    if (node->token != T_NULLOP)
        do {

            /* Cerca l'operando più complesso del nodo */
            tosim = NULL;
            maxcpx = 0;
            for (j=0; j < node->allocated; j++)
                if ( (node->operand[j]->complexity > maxcpx)
                     && (node->operand[j]->toread)
                   ) {
                        tosim = node->operand[j];
                        maxcpx = tosim->complexity;
                }

            (...)

            /* Semplifica l'operando */
            if (tosim != NULL) tmpcpx = simplify (tosim, tmpcpx);

        /* Se la complessità è stata sufficientemente ridotta esce */
        } while ( (tmpcpx > maxcomplexity) && (tosim != NULL) );

    if (tmpcpx > maxcomplexity) {

        /* Se la formula resta comunque troppo complessa
        semplifica il nodo in blocco */
        node->toread = FALSE;
        return (actualcpx - node->complexity + 1);

    } else return tmpcpx;
}
```

Una volta che la complessità della formula è stata sufficientemente ridotta il controllo passa alla funzione `speaktree()` che esegue la lettura vera e propria: questa

funzione legge il nodo del sottoalbero sfruttando le informazioni contenute nel campo `nodeunion`[4] che permette di recuperare, direttamente nel caso di numeri e lettere o passando attraverso la tabella dei comandi TEX riconosciuti per gli altri token, la stringa da inviare al sintetizzatore in corrispondenza ad un dato comando TEX. La funzione prosegue poi ricorsivamente la lettura passando agli operandi del nodo.

Se l'utente desidera entrare in modalità Storia la funzione `visittree()` passa il controllo alla routine:

```
void history (node)

TNODE *node; /* Nodo corrente */
```

che possiede una analoga struttura: sintesi della formula, lettura del tasto premuto, sua interpretazione, eccetto per il fatto di avere un set più ridotto di comandi.

Le funzioni che implementano le modalità Input file, TEX ed Editor non sono contenute in questo file in quanto fanno uso di alcune routine di input/output machine-dependent e quindi sono state relegate nel file `OUTPUT.C`[5] che contiene tutte e sole le routine di questo tipo.

[4] Vedi pag. 60.
[5] Vedi par. 7.2.

Capitolo 7

Funzioni ausiliarie

7.1 File sorgenti

- `MAIN.C` — In questo file è contenuta la funzione `main()` che si occupa di processare i parametri della linea di comando, di inizializzare l'hardware e di chiamare le funzioni che processano il file di configurazione ed il file TEX in input.
- **Funzioni di interfaccia con l'hardware** — Queste funzioni fanno da ponte fra l'hardware della macchina e le funzioni ad alto livello descritte nei capitoli precedenti. Questi file contengono le routine machine-dependent che fanno parte di TESI.
  - `KEYBOARD.C` — Questo file contiene la funzione

    ```
    unsigned getkey()
    ```

    che, sfruttando alcune routine del BIOS del computer, riesce a riconoscere i tasti speciali della tastiera (come **Home**, **Esc** o **F1**). `getkey()` ritorna un valore che rispecchia la combinazione di tasti premuta dall'utente.
  - `KEYBOARD.H` — Header file per `KEYBOARD.C`, contiene la definizione dei valori che corrispondono alle combinazioni di tasti riconosciute da `getkey()`.
  - `OUTPUT.C` — Funzioni che eseguono l'output a video. Queste funzioni sono anche responsabili dell'interfacciamento con il sintetizzatore vocale dato che questo avviene tramite uno screen reader.
  - `OUTPUT.H` — Header file per `OUTPUT.C`.
  - `TIMER.C` — Questo file contiene alcune funzioni che riprogrammano il timer del PC e che gestiscono gli interrupt ad esso associati.

    Queste funzioni implementano un cronometro che possiede una risoluzione di un millisecondo, utilizzato per la generazione del suono.
  - `TIMER.H` — Header file per `TIMER.C`.
- **Funzioni di gestione della memoria** — Queste funzioni si occupano di allocare e liberare la memoria necessaria alla creazione delle strutture dati utilizzate da TESI.
  - `MYMEMORY.C` — Qui sono contenute alcune funzioni che si sovrappongono alle funzioni di libreria `malloc()` e `free()` e che ne ricalcano i compiti, queste

funzioni però tengono traccia dei blocchi allocati facilitando il debug del programma.

  - `MYMEMORY.H` — Header file per `MYMEMORY.C`.
  - `NODEMEM.C` — Alcune funzioni che gestiscono la creazione e la distruzione di strutture `TNODE` per la costruzione degli alberi sintattici.
  - `NODEMEM.H` — Header file per `NODEMEM.C`.

- `DEBUG.C` — Alcune funzioni che gestiscono il log file, i messaggi di errore e le informazioni per il debug del programma.
- `DEBUG.H` — Header file per `DEBUG.C`.

7.2 Il le `OUTPUT.C`

Qui sono contenute le funzioni di input/output che dipendono direttamente dall'hardware.

Ogni volta che si è dovuto interfacciarsi con l'hardware della macchina si è fatto ricorso a funzioni contenute nelle librerie fornite con il MS C/C++ 7.0 che si è utilizzato nello sviluppo di TESI, queste funzioni sono facilmente riconoscibili nel listato del programma perché il loro nome inizia con un carattere "underscore".

Le funzioni utilizzate sono essenzialmente di tre tipi:

1. Funzioni di interfacciamento col video — utilizzate per la creazione di finestre sullo schermo. Funzioni di questo tipo, per ragioni di efficienza, scavalcano il BIOS del computer e quindi anche lo screen-reader; pertanto, tutto l'output effettuato tramite queste funzioni è trasparente all'utente non vedente.

   Un simile comportamento presenta alcuni vantaggi: la gestione delle finestre a video è semplificata, messaggi di errore e segnalazioni visive dirette ad un eventuale istruttore vedente (ad esempio il testo evidenziato nella finestra TEX), non interferiscono con il lavoro dell'utente, ma c'è pure l'inconveniente di costringere il programmatore ad interfacciarsi direttamente con l'hardware per ottenere l'output su sintetizzatore.

2. Funzioni di interfacciamento con i dischi — utilizzate in modalità Input per leggere i nomi dei file nelle directory, per cambiare la directory di lavoro etc.

3. Funzioni di interfacciamento col BIOS — Queste funzioni permettono al programmatore di chiamare direttamente il BIOS e di passare parametri tramite opportune strutture dati.

   Si è fatto ricorso a queste chiamate quando le routine di libreria non fornivano le necessarie funzionalità (es. programmazione dei timer del PC, output a video tramite BIOS etc.).

Utilizzando funzioni di questo tipo sono state costruite funzioni che fanno da cuscinetto con gli altri moduli del programma, pertanto tutto il codice machine-dependent è stato relegato in questo file.

Passiamo ora a vedere le funzioni contenute in questo file:

7.2.1 Gestione dello schermo

In questa categoria sono comprese alcune funzioni che, all'entrata ed all'uscita dal programma, impostano la modalità video:

```
void initvideo ()

/* Inizializza il video in modalita' testo 80x25 */

{
    /* Salva la modalita' video attuale */
    if (initialized == 0) {
        _getvideoconfig( &oldmode );
        initialized = 1;
    }

    _setvideomode ( _TEXTC80 );
}

void resetvideo ()

/* Riporta la modalita' video alle condizioni originarie */

{
    if (initialized == 1)
        _setvideomode ( oldmode.mode );

    _clearscreen (_GCLEARSCREEN);
    initialized = 0;
}
```

Inoltre è definita la funzione

```
void drawmask ()
```

che stampa a video le varie finestre utilizzate dal programma.

7.2.2 Output al sintetizzatore vocale

Esistono due funzioni che eseguono l'output di messaggi in finestra Sintetizzatore e verso il sintetizzatore vero e proprio:

```
void synthoutput (s)

char *s;

/*
Fa l'output di una stringa che deve essere
letta dal sintetizzatore vocale.
Contemporaneamente la mostra a video nella
finestra Sintetizzatore
*/

void silentoutput (s)

char *s;

/*
Mostra una stringa nella finestra Sintetizzatore senza leggerla
*/
```

Entrambe queste funzioni basano il loro funzionamento sulla routine:

```
static void output (char *s, int col, int silent)

/*
Fa l'output della stringa s nella finestra Sintetizzatore
usando il colore col.
silent e' un flag, se vale TRUE testo viene stampato ma non letto
*/

{
    /* Riga e colonna dell'ultimo output */
    static struct _rccoord currpos = {1, 1};

    int i;

    /* Imposta la finestra */
    ( ... )

    if (!silent) {

        for (i=0; s[i] != '\0'; ++i) {

            ( ... )

            /*
            Esegue l'output di ogni singolo carattere usando
            chiamate al BIOS
            */
            bios_outchar (s[i], col, SYNTHWBCOL);

            /* Aggiorna posizione nella finestra */
            if ( ++currpos.col > SYNTHWW ) {
                currpos.col = 1;
                ++currpos.row;
                space = 1;
            }
        }
        /* Scroll della finestra se necessario */
        if (currpos.row > SYNTHWH) {
            _scrolltextwindow (1);
            currpos.row = SYNTHWH;
        }

    } else { /* Silent */

        /* Stampa in finestra utilizzando le routine di libreria */
        ( ... )
   }
}
```

7.2.3 Modalità TEX

La modalità TEX viene implementata dalla funzione:

```
int texmode (line, column)

int line, column;

/*
Mostra nell'apposita finestra il testo TeX e permette di esplorarlo.
Il cursore viene posto inizialmente alla posizione all'interno del file
specificata da line e column.

La funzione ritorna il valore TEX_EDITFILE se l'utente ha chiesto di
modificare il file di input durante la visita dell'albero sintattico,
in caso contrario ritorna l'indice della formula su cui si trova il
cursore uscendo da questa modalita' oppure il valore
TEXMODE_NOFORMULA se il cursore non si trova in una formula.
*/
```

Non ci soffermeremo ulteriormente su questa funzione in quanto la sua struttura è piuttosto semplice e ricalca lo schema utilizzato per implementare le altre modalità: ciclo in cui viene letto il tasto premuto dall'utente e spostato il cursore di conseguenza.

Se la funzione ritorna il valore `TEX_EDITFILE` allora il chiamate dovrà preoccuparsi di riprocessare il file TEX in input che potrebbe essere stato modificato dall'utente.

7.2.4 Modalità input le

La funzione:

```
void readfname (retname, mask)

char retname[];
char *mask;
```

implementa la modalità Input File, questa routine, oltre a presentare il solito ciclo che interpreta i tasti premuti dall'utente si preoccupa, mediante chiamata di opportune funzioni del BIOS, di leggere i nomi dei file presenti nella directory di lavoro corrispondenti alla maschera scelta dall'utente (che di default è `*.TEX`). Questi nomi verranno utilizzati per creare il menù che viene presentato all'utente in questa modalità.

7.3 Il le `MYMEMORY.C`

Questo file contiene le funzioni `my_malloc()` e `my_free()` che vengono usate nel programma in luogo delle funzioni di libreria `malloc()` e `free()`. Il compito svolto da queste funzioni è analogo solo che, in più, viene tenuta traccia del numero di blocchi allocati e vengono stampati alcuni messaggi nel file di log al fine di aiutare il programmatore nella fase di debugging del programma.

Ecco a titolo di esempio il listato delle funzione `my_malloc()`:

```
void *my_malloc (size_t size)

{
    char s[80];
    void *tmp;

    /* Alloca la memoria */
    if ( (tmp = malloc(size)) == NULL)

       /* Se manca memoria segnala l'errore */
       criticalerrormsg ("my_malloc(): memoria esaurita");

    /* Tiene traccia dei blocchi allocati */
    ++allocated;

    /* Stampa un messaggio nel log file */
    sprintf (s, "malloc(): %p (%u) tot. %d", tmp, size, allocated);
    debugmsg (s);

    return tmp;
}
```

7.4 Il le `DEBUG.C`

In questo file sono contenute alcune routine preposte alla stampa di messaggi di errore e di debug.

```
void debugmsg (char *msg)
```

Stampa messaggi di debug nel log file, se questo è stato aperto dall'utente usando l'opzione `-L` sulla riga di comando[1].

```
void criticalerrormsg (char *s)
```

Stampa messaggi che segnalano un errore critico che causa la terminazione del programma.

Questa funzione chiama la routine

```
void my_exit (n)

int n;
```

che ripristina le condizioni del computer quali erano all'avvio di TESI: in particolare vengono resettati i timer e ripristinata la modalità grafica di avvio del programma.

[1] Vedi par. 1.2.

Conclusioni

Col presente lavoro si è inteso fornire un ausilio per permettere a persone che non hanno accesso a testi stampati di trattare la matematica a livello universitario, cercando di colmare un vuoto che è causa di abbandono scolastico, soprattutto nelle facoltà tecnico-scientifiche, da parte di studenti ipovedenti o non vedenti.

La matematica è probabilmente l'unico campo dell'attività umana in cui il lavoro viene svolto solamente in forma scritta: naturalmente questo è un grosso problema per i non vedenti; inoltre, essendo la matematica alla base di molte altre discipline, il problema diviene ancor più rilevante. È quindi naturale che l'accesso alla matematica da parte dei non vedenti sia un problema molto sentito, le attività svolte finora in questo campo sono state condotte in ambiti ristretti e, pur avendo portato a risultati interessanti, si sono spesso fermate a livello di ricerca, senza portare ad applicazioni valide di ampia diffusione. Questo genere di problemi è ad esempio emerso all'International Workshop on Access to Mathematics, tenutosi ad Amsterdam il 3–4 febbraio 1993, dove si è cercato di coordinare la ricerca in questo campo dotandola di uno sbocco più ampio a livello internazionale.

Il programma sviluppato consente di fornire al disabile le informazioni "spaziali" sulla formula che viene trattata consentendogli di costruire una rappresentazione mentale della struttura dell'espressione. In particolare, la navigazione attraverso l'albero sintattico delle espressioni, stimola l'utente a costruire una associazione fra il tasto cursore premuto (e quindi la direzione corrispondente) ed il testo sintetizzato: in questo modo viene stabilita una struttura spaziale-gerarchica analoga a quella che la persona vedente costruisce (spesso a livello inconscio) osservando una formula stampata in un testo.

Il principale vantaggio derivante da questo tipo di approccio risiede nella sua semplicità d'impiego: per fornire queste informazioni sulla struttura della formula non è necessario un controllo sulla prosodia del testo sintetizzato né è richiesto l'ausilio di un dispositivo di interfaccia con l'utente particolare (barra braille, touch-tablet[2], suono stereofonico[3], etc.), tutto quello che viene richiesto è la presenza di uno screen reader, strumento di cui praticamente tutte le postazioni per non vedenti sono necessariamente fornite; inoltre, TESI è pensato per lavorare con qualsiasi tipo di screen reader, senza costringere all'acquisto di materiale dedicato o tecnologicamente sofisticato.

Tutto questo si traduce essenzialmente in un vantaggio economico per l'utente. Questo aspetto, che pur trascende il lato puramente tecnico del lavoro, non è da trascurare: i disabili devono infatti far fronte a spese che l'utente usuale non è costretto ad affrontare (come appunto l'acquisto dello screen reader). Costringere l'utente all'acquisto di materiale specifico per permettergli di manipolare espressioni matematiche (ad esempio uno screen reader che permetta il controllo sulla prosodia o fornisca effetti sonori

[2] Vedi [6] riguardo l'impiego di simili dispositivi.

[3] Vedi [3] sulla possibilità di utilizzare suoni stereofonici con gli screen reader. Vedi [4] riguardo all'impiego di suoni non verbali per facilitare la comprensione del testo sintetizzato.

stereofonici, magari non compatibile con i programmi che l'utente già usa) può essere economicamente proibitivo e comunque sicuramente scomodo.

L'economicità (il programma è distribuito gratuitamente per via telematica) e l'universalità di impiego sono dunque due punti forti in cui TESI batte altri prodotti analoghi, anche commerciali.

Un altro punto su cui si ritiene di aver raggiunto dei validi risultati è la flessibilità del programma: non solo è possibile impostare varie opzioni per rendere TESI più consono alle proprie esigenze, ma è possibile istruire completamente il programma sul tipo dei comandi da riconoscere e sul modo in cui vanno presentati all'utente. Questo permette di coprire un largo bacino di utenza: tramite una opportuna configurazione TESI può trattare la matematica in diversi campi: il programma può leggere formule in cui il medesimo simbolo ha significati diversi a seconda della materia in cui viene impiegato (ad es. il simbolo "+" indica normalmente l'addizione ma, nell'algebra booleana, rappresenta l'operazione di OR logico). Inoltre, una certa flessibilità nell'interpretazione delle formule consente a TESI di costruire l'albero sintattico di espressioni potenzialmente ambigue permettendo poi all'utente di risolvere l'ambiguità (ad es. TESI interpreta correttamente un'espressione del tipo: $P(x, y)$ indipendentemente dal fatto che P rappresenti un punto geometrico od una funzione).

Infine, ricordiamo che il programma, durante il suo funzionamento, fornisce una serie di informazioni dirette ad un eventuale istruttore vedente presente alla sessione di lavoro. Queste informazioni sono presentate in modo trasparente per l'utente non vedente e quindi non rallentano il suo lavoro costringendolo a prestare attenzione ad informazioni ridondanti.

A questo punto ritengo importante sottolineare la diversa ottica di programmazione che lo scrivente ha dovuto affrontare: solitamente il programmatore viene incoraggiato a fornire informazioni sullo stato di funzionamento del programma. Nella programmazione per utenti non vedenti invece, dato che le informazioni passano tutte per lo screen reader, le informazioni "di stato" vanno filtrate, magari ridotte ad un semplice cicalino la cui frequenza e la cui durata definiscono il tipo di informazione da trasmettere. In definitiva, durante la stesura di TESI, mi sono trovato a dover trattare con un nuovo tipo di utenza e quindi con un nuovo modo di pensare l'interfaccia utente. Credo che la possibilità di una utenza che si discosta da quella classica venga ignorata troppo spesso dai programmatori: è successo finora con il software e sta già succedendo con le pagine WWW. Sarebbe auspicabile una maggiore sensibilità da parte degli sviluppatori in questo senso: basta un piccolo sforzo per rendere le interfacce utente fruibili anche ai disabili aumentando in questo modo il campo di utenza dei propri prodotti e, in definitiva, l'utilità degli stessi.

TESI basa il suo funzionamento su un input costituito da file TEX, questa può essere vista come una limitazione ma ritengo che non lo sia, o meglio, che lo sia solo in parte. Indubbiamente usare TEX richiede un certo periodo di apprendimento per familiarizzarsi con i suoi comandi, ma il tempo investito in tal senso è ampiamente ripagato dai risultati: TEX costituisce probabilmente il più valido strumento disponibile per l'utente non vedente per la creazione di stampati di alta qualità estetica. Inoltre, la presenza di comandi che specificano la struttura logica del documento (suddivisione in capitoli, paragrafi, e via di seguito, l'indicazione di componenti del testo quali liste, campi con indirizzi, formule matematiche etc.), rendono il formato TEX potenzialmente più accessibile ai non vedenti che altri formati utilizzati nei programmi di word processing. Come evidenziato in [1] la stesura di documenti strutturati è il primo necessario passo per abbattere le naturali barriere che impediscono un normale accesso ai documenti

da parte di chi non può leggere, in tal senso un editor per non vedenti orientato ai documenti TEX sarebbe un utile ausilio.

La vera limitazione di TESI è invece costituita dal fatto che la modifica delle formule avviene solo editando il file TEX: un miglioramento da introdurre in versioni future potrebbe essere la capacità di cambiare direttamente l'albero sintattico delle espressioni, magari mediante un'interfaccia a menù; sarebbe utile fornire all'utente la possibilità di eseguire operazioni di "taglia e incolla" sui rami dell'albero e di salvare in formato TEX i risultati.

Così com'è TESI può costituire comunque un valido ausilio per gli studenti non vedenti che utilizzano la matematica alle scuole superiori od in ambito universitario. Il fatto che il programma preveda informazioni per un istruttore vedente potrebbe suggerire un'impiego di TESI già ai primi livelli del sistema scolastico, naturalmente previa applicazione delle modifiche sopra menzionate. Un aiuto in tal senso potrebbe essere fornito dalla possibilità di interfacciare opzionalmente il programma con una barra braille dove vengano "stampate" le parti di formula di cui si sta svolgendo la sintesi vocale.

Come ricordato sopra, uno sviluppo futuro di questo lavoro potrebbe essere la creazione di un editor di file TEX, pensato per gli utenti non vedenti, in grado di trattare anche testo normale oltre alle formule matematiche le quali sarebbero interpretate da un modulo molto simile all'attuale versione di TESI. Tale editor, con qualche lieve modifica, potrebbe facilmente essere impiegato anche per la costruzione di pagine WWW.

Appendice A

File di configurazione

A.1 Il le DEFAULT.CFG

Questo file viene letto per primo all'avvio di TESI. Esso richiama gli altri file che definiscono i comandi plain TEX e LATEX.

```
% File di configurazione di default per il programma

% Lettura altri file di default
Input texplain.cfg
Input latex209.cfg


% Impostazione parametri di personalizzazione del programma
PausaBreve "";
PausaLunga "";
MaxComplexity 5;
EditCommand "edit #f";
NoLeftDiscrim;
ErrorBell;
NoReadAllFormula;
```

A.2 Il le TEXPLAIN.CFG

Questo file di configurazione definisce i comandi per il plain TEX.

```
% File di configurazione comandi plain TeX


% Comandi di formattazione che devono venire ignorati
Garbage "\textstyle";       Garbage "\displaystyle";
Garbage "\scriptstyle";     Garbage "\scriptscriptstyle";
Garbage "\mathstrut";       Garbage "\mathsurround";
Garbage "\delimiterfactor"; Garbage "\delimitershortfall";
Garbage "\relpenality";     Garbage "\binoppenality";
Garbage "\nobreak";
Garbage "\limits";          Garbage "\nolimits";
```

```
Garbage "\ ";               Garbage "\,";
Garbage "\>";               Garbage "\;";
Garbage "\!";
Garbage "\quad";            Garbage "\qquad";


% Lettere minuscole e maiuscole
Lettera "a" : "a"; Lettera "b" : "b"; Lettera "c" : "c";
Lettera "d" : "d"; Lettera "e" : "e";
Lettera "i" : "i"; Lettera "j" : "j"; Lettera "k" : "k";
Lettera "l" : "l"; Lettera "m" : "m"; Lettera "n" : "n";
Lettera "o" : "o"; Lettera "p" : "p"; Lettera "q" : "q";
Lettera "r" : "r"; Lettera "s" : "s"; Lettera "t" : "t";
Lettera "u" : "u"; Lettera "v" : "v"; Lettera "w" : "w";
Lettera "x" : "x"; Lettera "y" : "y"; Lettera "z" : "z";
Lettera "A" : "A"; Lettera "B" : "B"; Lettera "C" : "C";
Lettera "D" : "D"; Lettera "E" : "E"; Lettera "F" : "F";
Lettera "G" : "G"; Lettera "H" : "H"; Lettera "I" : "I";
Lettera "J" : "J"; Lettera "K" : "K"; Lettera "L" : "L";
Lettera "M" : "M"; Lettera "N" : "N"; Lettera "O" : "O";
Lettera "P" : "P"; Lettera "Q" : "Q"; Lettera "R" : "R";
Lettera "S" : "S"; Lettera "T" : "T"; Lettera "U" : "U";
Lettera "V" : "V"; Lettera "W" : "W"; Lettera "X" : "X";
Lettera "Y" : "Y"; Lettera "Z" : "Z";


% Accenti
Accento "\overline" : "sopralineato";
Accento "\underline" : "sottolineato";
Accento "\hat" : "cappello";
Accento "\widehat" : "cappello";
Accento "\check" : "accento check";
Accento "\tilde" : "tilde";
Accento "\widetilde" : "tilde";
Accento "\acute" : "accento acuto";
Accento "\grave" : "accento grave";
Accento "\dot" : "accento punto";
Accento "\ddot" : "accento doppio punto";
Accento "\breve" : "accento breve";
Accento "\bar" : "barrato";
Accento "\vec" : "vettore";


% Lettere greche
Lettera "\Gamma" : "gamma maiuscola";
Lettera "\Delta" : "delta maiuscola";
Lettera "\Theta" : "theta maiuscola";
Lettera "\Lambda" : "lambda maiuscola";
Lettera "\Xi" : "xi maiuscola";
```

```
Lettera "\Pi" : "pi greco maiuscolo";
Lettera "\Sigma" : "sigma maiuscola";
Lettera "\Upsilon" : "upsilon maiuscola";
Lettera "\Phi" : "fi maiuscola";
Lettera "\Psi" : "psi maiuscola";
Lettera "\Omega" : "omega maiuscolo";
Lettera "\alpha" : "alfa";    Lettera "\beta" : "beta";
Lettera "\gamma" : "gamma";   Lettera "\delta" : "delta";
Lettera "\epsilon" : "epsilon";
Lettera "\varepsilon" : "epsilon corsivo";
Lettera "\zeta" : "zeta";     Lettera "\eta" : "eta";
Lettera "\theta" : "theta";   Lettera "\vartheta" : "theta corsivo";
Lettera "\iota" : "iota";     Lettera "\kappa" : "kappa";
Lettera "\lambda" : "lambda"; Lettera "\mu" : "mu";
Lettera "\nu" : "nu";         Lettera "\xi" : "xi";
Lettera "\pi" : "pi greco";   Lettera "\varpi" : "\varpi";
Lettera "\rho" : "rho";       Lettera "\varrho" : "rho corsivo";
Lettera "\sigma" : "sigma";   Lettera "\varsigma" : "varsigma";
Lettera "\tau" : "tau";       Lettera "\upsilon" : "upsilon";
Lettera "\phi" : "phi";       Lettera "\varphi" : "phi corsivo";
Lettera "\chi" : "chi";       Lettera "\psi" : "psi";
Lettera "\omega" : "omega";


% Lettere speciali
Lettera "\imath" : "i";
Lettera "\jmath" : "j";
Lettera "\ell" : "l corsivo";


% Simboli
Simbolo "\Re" : "reale";        Simbolo "\Im" : "immaginario";
Simbolo "\partial" : "differenziale";
Simbolo "\infty" : "infinito";  Simbolo "'" : "primo";
Simbolo "\prime" : "primo";     Simbolo "\nabla" : "gradiente";
Simbolo "\emptyset" : "insieme vuoto";
Simbolo "\forall" : "per ogni"; Simbolo "\exist" : "esiste";
Simbolo "\angle" : "fase";      Simbolo "\nabla" : "nabla";
Simbolo "\dots" : "puntini";    Simbolo "\ldots" : "puntini";
Simbolo "\cdots" : "puntini centrati";
Simbolo "\ddots" : "puntini in diagonale";


% Operatori binari
OperatoreSomma "+" : "più", "somma complessa";
OperatoreSomma "-" : "meno", "sottrazione complessa";
OperatoreSomma "\pm" : "più e meno";
OperatoreSomma "\mp" : "meno e più";
OperatoreSomma "\setminus" : "sottratto all'insieme",
```

```
    "sottrazione complessa fra insiemi";
OperatoreSomma "\cup" : "unito", "unione complessa";

OperatoreProdotto "*" : "per", "moltiplicazione complessa";
OperatoreProdotto "\ast" : "per", "moltiplicazione complessa";
OperatoreProdotto "\cdot" : "punto", "moltiplicazione complessa";
OperatoreProdotto "\times" : "segno di per",
    "moltiplicazione complessa";
OperatoreProdotto "/" : "diviso" toleft "che divide",
    "divisione complessa";
OperatoreProdotto "\cap" : "intersecato", "intersezione complessa";

OperatoreOR "\vee" : "OR", "operazione di OR complessa";
OperatoreAND "\vedge" : "AND", "operazione di AND complessa";
OperatoreNOT "\neg" : "NOT", "negazione complessa";

OperatoreFattoriale "!" : "fattoriale", "fattoriale complesso";

Relazione "=" : "uguale", "uguaglianza complessa";
Relazione "<" : "minore" toleft "maggiore",
    "diseguaglianza complessa";
Relazione ">" : "maggiore" toleft "minore",
    "diseguaglianza complessa";
Relazione "\ll" : "molto minore" toleft "molto maggiore",
    "diseguaglianza complessa";
Relazione "\gg" : "molto maggiore" toleft "molto minore",
    "diseguaglianza complessa";
Relazione "\leq" : "minore-uguale" toleft "maggiore-uguale",
    "diseguaglianza complessa";
Relazione "\geq" : "maggiore-uguale" toleft "minore-uguale",
    "diseguaglianza complessa";
Relazione "\equiv" : "coincode con", "uguaglianza complessa";
Relazione "\sim" : "asintotico";
Relazione "\simeq" : "circa uguale";
Relazione "\subset" : "contenuto" toleft "contiene";
Relazione "\superset" : "contiene" toleft "contenuto";
Relazione "\subseteq" : "contenuto-uguale"
    toleft "contiene-uguale";
Relazione "\superseteq" : "contiene-uguale"
    toleft "contenuto-uguale";
Relazione "\propto" : "proporzonale";
Relazione "\mid" : "tale che";
Relazione "\doteq" : "uguale puntato";
Relazione "\parallel" : "parallelo";
Relazione "\perp" : "perpendicolare";
Relazione "\mapsto" : "tende a";
Relazione "\approx" : "circa";

Separatore ":" : "due-punti", "espressioni separate da due punti";
```

```
Separatore "\colon" : "due-punti", "espressioni separate da due punti";
Separatore "," : "virgola", "espressioni separate da virgole";
Separatore ";" : "punto e virgola",
    "espressioni separate da punto e virgola";


% Specificatori di dimensione per le parentesi: vanno eliminati
Garbage "\big";  Garbage "\Big";  Garbage "\bigg";  Garbage "\Bigg";
Garbage "\bigl"; Garbage "\Bigl"; Garbage "\biggl"; Garbage "\Biggl";
Garbage "\bigr"; Garbage "\Bigr"; Garbage "\biggr"; Garbage "\Biggr";
Garbage "\bigm"; Garbage "\Bigm"; Garbage "\biggm"; Garbage "\Biggm";


% I \left e \right per accoppiare parentesi vanno tolti
Garbage "\left";  Garbage "\right";
Garbage "\left."; Garbage "\right.";


% Parentesi
ParentesiAperta "(" : "aperta tonda";
ParentesiChiusa ")" : "chiusa tonda";
ParentesiAperta "[" : "aperta quadra";
ParentesiChiusa "]" : "chiusa quadra";
ParentesiAperta "\lbrack" : "aperta quadra";
ParentesiChiusa "\rbrack" : "chiusa quadra";
ParentesiAperta "\{" : "aperta graffa";
ParentesiChiusa "\}" : "chiusa graffa";
ParentesiAperta "\lbrace" : "aperta graffa";
ParentesiChiusa "\rbrace" : "chiusa graffa";
ParentesiAperta "\lfloor" : "aperta quadra inferiore";
ParentesiChiusa "\rflorr" : "chiusa quadra inferiore";
ParentesiAperta "\lceil" : "aperta quadra superiore";
ParentesiChiusa "\rceil" : "chiusa quadra superiore";
ParentesiAperta "\langle" : "aperta acuta";
ParentesiChiusa "\rangle" : "chiusa acuta";


% Apici e pedici
Apice "^";  Apice "\sp";
Pedice "_"; Pedice "\sb";


% Funzioni
Funzione "f" : "f"; Funzione "g" : "g"; Funzione "h" : "h";
Funzione "\arccos" : "arcocoseno";
Funzione "\arcsin" : "arcoseno";
Funzione "\arctan" : "arcotangente";
Funzione "\cos" : "coseno";
Funzione "\cosh" : "coseno iperbolico";
```

```
Funzione "\cot" : "cotangente";
Funzione "\coth" : "cotangente iperbolica";
Funzione "\ker" : "nucleo";
Funzione "\lim" : "limite";
Funzione "\liminf" : "limite inferiore";
Funzione "\limsup" : "limite superiore";
Funzione "\ln" : "logaritmo naturale";
Funzione "\log" : "logaritmo";
Funzione "\max" : "massimo";
Funzione "\min" : "minimo";
Funzione "\sin" : "seno";
Funzione "\sinh" : "seno iperbolico";
Funzione "\tan" : "tangente";
Funzione "\tanh" : "tangente iperbolica";


% Large operators
OperatoreEsteso "\sum" : "sommatoria", "sommatoria complessa";
OperatoreEsteso "\prod" : "produttoria",
    "produttoria complessa";
OperatoreEsteso "\sum" : "sommatoria", "sommatoria complessa";
OperatoreEsteso "\int" : "integrale", "integrale complesso";
OperatoreEsteso "\smallint" : "integrale", "integrale complesso";
OperatoreEsteso "\oint" : "integrale circuitale",
    "integrale complesso";
OperatoreEsteso "\bigcap" : "intersezione", "intersezione complessa";
OperatoreEsteso "\bigcup" : "unione", "unione complessa";


% \over e comandi analoghi
Frazione "\over";


% \atop e comandi analoghi
Atop "\atop" : "";
Atop "\choose" : "coefficiente binomiale";


% \sqrt e comandi analoghi
Radice "\sqrt" : "radice quadrata di",
    "radice quadrata di un'espressione complessa";
```

A.3 Il le LATEX209.CFG

Qui sono definiti i comandi per il LaTeX nella versione 2.09.

```
% File di configurazione comandi LaTeX 2.09

%Apertura e chiusura (display) math mode
InizioMathMode "\("; FineMathMode "\)";
InizioDisplayMathMode "\["; FineDisplayMathMode "\]";

Testo "\hbox";
```

Appendice B

Grammatiche

B.1 La grammatica per il le di congurazione

```
input → ϵ | input statement ';'
input → error ';'
statement → C_LFT_DISCR | C_NO_LFT_DISCR
statement → C_ERROR_BELL | C_NO_ERROR_BELL
statement → C_READALL | C_NO_READALL
statement → C_MAXCPX C_INTEGER
statement → C_SYNTHPAUSE C_INTEGER
statement → C_EDITCMD C_STRING
statement → (C_PAUSAB | PAUSA_L) C_STRING
statement → C_NULL C_STRING
statement → C_STR C_STRING ':' C_STRING
statement → C_STR_OPTSTR C_STRING ':' C_STRING [',' C_STRING]
statement → C_STR_LFTSTR_OPTSTR C_STRING ':' C_STRING [C_TOLEFT ','
C_STRING] [',' C_STRING]
```

B.2 La grammatica per il le TEX in input

```
input → ϵ | input formula ';'

formula → T_BEGIN espressione T_END
formula → T_BEGIN corpo_frazione T_END
formula → T_BEGIN corpo_atop T_END
formula → T_BEGIN error T_END
```

espressione → espressione T_RELAZIONE espressione
espressione → espressione T_SEPARATORE espressione
espressione → espressione T_OPSOMMA espressione
espressione → espressione T_OPOR espressione
espressione → espressione T_OPPROD espressione
espressione → espressione espressione2
espressione → T_OPNOT espressione
espressione → espressione1
espressione1 → T_OPSOMMA espressione1
espressione1 → espressione2
espressione2 → espressione2 T_OPFACT
espressione2 → espressione2 T_OPEXP espressione2
espressione2 → espressione3

espressione3 → simbolo
espressione3 → elemento2
espressione3 → large_operator
espressione3 → testo
espressione3 → ACCENTO BEGIN espressione END
espressione3 → ACCENTO BEGIN error END
espressione3 → BEGIN error END

ACCENTO → T_ACCENTO TeXtrue

token_tex → token_tex_numerico
token_tex → token_tex_non_numerico

token_tex_numerico → TeXtrue tokenn TeXfalse
token_tex_numerico → TeXtrue BEGIN token_tex_numerico END TeXfalse
tokenn → T_NUMERO
tokenn → BEGIN T_NUMERO END

token_tex_non_numerico → TeXtrue token1 TeXfalse
token_tex_non_numerico → TeXtrue BEGIN token_tex_non_numerico END
TeXfalse
token1 → T_SIMBOLO
token1 → lettera
token1 → frazione
token1 → costrutto_atop
token1 → BEGIN espressione END
token1 → BEGIN error END

BEGIN → '{' TeXfalse
END → '}'

TeXtrue → ϵ
TeXfalse → ϵ

testo → T_TEXTCMD T_TESTO

```
simbolo → T_SIMBOLO
simbolo → simbolo T_APICE token_tex
simbolo → simbolo T_PEDICE token_tex
simbolo → ACCENTO T_SIMBOLO TeXfalse
simbolo → ACCENTO BEGIN simbolo END

lettera → T_LETTERA
esp_letterale → lettera
esp_letterale → esp_letterale T_APICE token_tex
esp_letterale → esp_letterale T_PEDICE token_tex
esp_letterale → ACCENTO lettera TeXfalse
esp_letterale → ACCENTO BEGIN esp_letterale END

esp_numerica → T_NUMERO
esp_numerica → BEGIN T_NUMERO T_OVER T_NUMERO END
esp_numerica → T_RADICE token_tex_numerico
esp_numerica → esp_numerica T_APICE token_tex
esp_numerica → esp_numerica T_PEDICE token_tex
esp_numerica → ACCENTO token_tex_numerico
esp_numerica → ACCENTO BEGIN esp_numerica END

esp_parentesi → T_OPENING espressione T_CLOSING
esp_parentesi → esp_parentesi T_APICE token_tex
esp_parentesi → esp_parentesi T_PEDICE token_tex
esp_parentesi → ACCENTO BEGIN esp_parentesi END
esp_parentesi → T_OPENING error T_CLOSING

corpo_frazione → espressione T_OVER espressione
corpo_frazione → T_NUMERO T_OVER espressione
frazione → BEGIN corpo_frazione END
frazione → frazione T_APICE token_tex
frazione → frazione T_PEDICE token_tex
frazione → ACCENTO BEGIN corpo_frazione END

corpo_atop → espressione T_ATOP espressione
costrutto_atop → BEGIN corpo_atop END
costrutto_atop → costrutto_atop T_APICE token_tex
costrutto_atop → costrutto_atop T_PEDICE token_tex
costrutto_atop → ACCENTO BEGIN corpo_atop END

radice → T_RADICE token_tex_non_numerico
radice → ACCENTO BEGIN radice END
```

```
funzione → corpo_di_funzione argomento_di_funzione
corpo_di_funzione → T_FUNZIONE
corpo_di_funzione → corpo_di_funzione T_APICE token_tex
corpo_di_funzione → corpo_di_funzione T_PEDICE token_tex
corpo_di_funzione → ACCENTO T_FUNZIONE TeXfalse
corpo_di_funzione → ACCENTO BEGIN corpo_di_funzione END
argomento_di_funzione → sequenza1
argomento_di_funzione → elemento1
argomento_di_funzione → sequenza1 elemento1
argomento_di_funzione → funzione

elemento_sequenza1 → esp_numerica
elemento_sequenza1 → esp_letterale
sequenza1 → elemento_sequenza1
sequenza1 → sequenza1 elemento_sequenza1

elemento1 → esp_parentesi
elemento1 → frazione
elemento1 → costrutto_atop
elemento1 → radice

elemento2 → elemento_sequenza1
elemento2 → elemento1
elemento2 → funzione
sequenza2 → elemento2
sequenza2 → sequenza2 elemento2

large_operator → corpo_di_operatore argomento_di_operatore
corpo_di_operatore → T_LARGEOP
corpo_di_operatore → corpo_di_operatore T_APICE token_tex
corpo_di_operatore → corpo_di_operatore T_PEDICE token_tex
corpo_di_operatore → ACCENTO T_LARGEOP TeXfalse
corpo_di_operatore → ACCENTO BEGIN large_operator END
argomento_di_operatore → sequenza2
```

Bibliografia

[1] B. Bauwens, J. Engelen, F. Evenepoel "Structuring Documents: the Key to Increasing Access to Information for the Print Disabled" in ICCHP '94 Proceedings. Proceedings Lecture Notes in Computer Science series, volume 860, Vienna: W. L. Zagler, G. Busby, R. Wagner editori, 1994, pag. 214. ISBN 3-540-58476-5

[2] G. Bruno, Linguaggi formali e compilatori, Torino: UTET Libreria, 1992, capitoli 1–5.

[3] K. Crispien ed altri "Using Spatial Audio for the Enhanced Presentation of Synthesised Speech within Screen-Readers for Blind Computer Users" in ICCHP '94 Proceedings. Proceedings Lecture Notes in Computer Science series, volume 860, Vienna: W. L. Zagler, G. Busby, R. Wagner editori, 1994, pag. 144. ISBN 3-540-58476-5

[4] A. Darvishi, V. Guggiana, E. Munteanu, H. Schauer "Synthesizing Non-Speech Sounds to Support Blind and Visually Impaired Computer Users" in ICCHP '94 Proceedings. Proceedings Lecture Notes in Computer Science series, volume 860, Vienna: W. L. Zagler, G. Busby, R. Wagner editori, 1994, pag. 385. ISBN 3-540-58476-5

[5] C. Donnelly, R. Stallman, Bison, the YACC-compatible parser generator, Bison version 1.25, Free Software Foundation, Novembre 1995.

[6] J. Fricke, H. Baehring "Displaying Laterally Moving Tactile Information" in ICCHP '94 Proceedings. Proceedings Lecture Notes in Computer Science series, volume 860, Vienna: W. L. Zagler, G. Busby, R. Wagner editori, 1994, pag. 461. ISBN 3-540-58476-5

[7] M. Goossens, F. Mittelbach, A. Samarin, The LaTeX companion, Addison-Wesley, 1994.

[8] J. E. Hopcroft, J. D. Ullman, Introduction to automata theory, languages and computation, Addison-Wesley, 1979, capitolo 10.

[9] D. E. Knuth, The TeXbook, Addison-Wesley.

[10] L. Lamport, LaTeX user guide & reference manual, Addison-Wesley.

[11] L. F. Ludwig, N. Pincever, M. Cohen "Extending the notion of a window system to audio" IEEE Computer, pag. 66–72, Agosto 1990.

[12] M. H. O'Malley, D. R. Kloker, B. Dara-Abrams "Recovering parentheses from spoken algebraic expression" IEEE transactions on audio and electroacoustics, vol. AU-21, n. 3, pag. 217–220, Giugno 1973.

[13] V. Paxon, Flex, A fast scanner generator, Edition 2.5, The Regents of the University of California, Marzo 1995.

[14] M. D. Spivak, The joy of TEX: a gourmet guide to typesetting with the AMS-TEX macro package, Providence, Rhode Island: American mathematical society, seconda edizione, 1990.

[15] R. Stevens, A. Edwards, "Mathtalk: The Design of an Interface for Reading Algebra Using Speech" in ICCHP '94 Proceedings. Proceedings Lecture Notes in Computer Science series, volume 860, Vienna: W. L. Zagler, G. Busby, R. Wagner editori, 1994, pag. 313. ISBN 3-540-58476-5