LaTeX e grammatiche (la faccia triste dell'informatica)

Gianluca Pignalberi[*], *Enrico Bini*[†]

Sommario

Le grammatiche, i diagrammi sintattici, gli automi, sono strumenti fondamentali nell'informatica; è grazie a loro se i nostri computer sono in grado di fare quello che sanno. LaTeX deve essere in grado di comporli. In questo articolo vedremo quali strumenti abbiamo per aggiungere questi elementi ai nostri documenti.

Abstract

Grammars, syntax diagrams and automata are Computer Science's basic tools; thanks to them our computers can do what they do. LaTeX has to be able to typeset them. In this paper we'll overview which tools we can use to add those elements to our documents.

1 Introduzione

Contrariamente a quanto pensano gli studenti delle scuole di grado inferiore, la grammatica non è una materia arida e noiosa (lo è, semmai, il metodo usato per insegnarla). È alla base del linguaggio, ne fornisce le regole per la costruzione. In pratica, è il fondamento della comunicazione.

In informatica le grammatiche sono il fondamento dei linguaggi di programmazione e di meccanismi quali automi (Arbib *et al.*, 1981; Kfoury *et al.*, 1982; Moll *et al.*, 1988). Con un intuito felicissimo, Noam Chomsky ne traccia la gerarchia e le corrispondenze con meccanismi di generazione.

LaTeX, in quanto strumento d'elezione della comunità scientifica, non poteva non avere a disposizione degli strumenti adatti a scrivere e disegnare grammatiche, regole di riscrittura o produzioni, automi e altro. L'articolo vuole essere un compendio dei pacchetti esistenti e una guida, non breve né esaustiva, al loro uso. Riprendo qui, espandendoli, alcuni argomenti affini da me pubblicati alcuni anni fa su Dev (Pignalberi, 2004a,b), rivista del defunto e resuscitando Gruppo Editoriale Infomedia; sono certo di non infrangere alcun diritto d'autore.

2 Grammatiche BNF

Una grammatica G è una quadrupla (X, V, S, P) in cui X e V sono alfabeti finiti disgiunti e

1. X è l'alfabeto terminale della grammatica;
2. V è l'alfabeto non-terminale, o variabile, della grammatica;
3. S è il simbolo iniziale della grammatica;
4. P è l'insieme di produzioni della grammatica. P è un insieme di coppie (v, w) con v stringa di $(X \cup V)$ contenente almeno un simbolo non-terminale, mentre w è un elemento arbitrario di $(X \cup V)^*$. Un elemento (v, w) di P è di solito scritto $v \to w$, ed è definito regola di riscrittura, o produzione.

Data la precedente definizione, possiamo ora scrivere una grammatica; una semplice è quella delle parentesi corrispondenti: $G = (\{(,)\}, \{S\}, S, \{S \to (), S \to (S), S \to SS\})$. Siccome dal contesto è comprensibile quale simbolo appartenga a quale alfabeto, possiamo fornire la grammatica dando solo le sue produzioni.

Uno degli standard per denotare regole e produzioni grammaticali è il cosiddetto BNF: Backus-Naur Form. Gli strumenti per scrivere agevolmente dette regole sono forniti da syntax. Questo aggiunge a LaTeX anche altri comandi per altre finalità: formattazione di elementi sintattici, ridefinizione del carattere underscore (_), abbreviazione della sintassi del comando `\verb`. Mentre gli ultimi due comandi/caratteristiche possono essere considerate marginali per la composizione di grammatiche, non altrettanto deve essere considerata la formattazione di elementi sintattici.

All'interno di una serie di produzioni ci sono simboli terminali e non-terminali. I simboli terminali possono essere scritti tra apici oppure no. I comandi per ottenere le due rappresentazioni sono `\lit` e `\lit*`. I simboli non-terminali sono in genere rappresentati tra parentesi angolari. Il comando necessario introdotto da syntax è synt. I comandi `\synt{S}` e `\lit{()}` generano, rispettivamente, $\langle S \rangle$ e `()`.

Ora abbiamo in potenza tutti gli strumenti per scrivere una grammatica BNF; il problema, evidenziato più volte in Gregorio (2004), è la totale mancanza di chiarezza del codice: il sorgente di una grammatica scritta con synt è molto lungo e non agevole da seguire.

2.1 L'ambiente grammar

Per evitare sessioni di scrittura lunghe e passibili di errori (o di orrori), syntax fornisce l'ambiente grammar, realizzato proprio per rendere facile la

[*] Autore
[†] Presentatore

scrittura di grammatiche BNF. Scrivere una grammatica qualunque è ora estremamente semplice. Vediamo la forma BNF per la grammatica relativa al linguaggio delle parentesi corrispondenti:

$$\begin{array}{rcl} \langle S\rangle & ::= & () \\ & | & (\langle S\rangle) \\ & | & \langle S\rangle\langle S\rangle \end{array}$$

Queste produzioni dicono semplicemente: il simbolo non-terminale S può essere sostituito con una coppia di parentesi (simbolo terminale), oppure con un simbolo non-terminale S chiuso in parentesi, oppure con due simboli non-terminali S. Possiamo generare così qualunque stringa di parentesi corrispondenti, sia annidate che affiancate. Il codice della grammatica, pulitissimo, è:

```
\begin{grammar}
<S> ::= ()
\alt (<S>)
\alt <S><S>
\end{grammar}
```

Vediamo come ottenere la stringa (()(()())) a partire dalle produzioni grammaticali. S → (S) → (SS) → (()S) → (()(S)) → (()(SS)) → (()(()S)) →(()(()())).

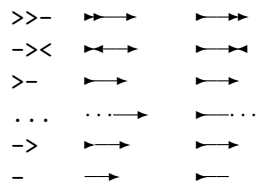
3 Diagrammi sintattici

Un diagramma sintattico è uno schematismo grafico teoricamente in grado di rendere semplice a chiunque la lettura di una grammatica: non tutti sono in grado di comprendere le produzioni di una grammatica BNF.

Il pacchetto syntax è dotato di un ambiente per formattare i diagrammi sintattici: syntdiag. Nella sua forma base syntdiag compone i diagrammi entro i limiti della pagina; nella versione *starred* (syntdiag*) siamo invece liberi di eccedere tali limiti, a meno di non fissare un limite, secondo la seguente sintassi:

```
\begin{syntdiag*}[dichiarazioni][ampiezza]
<testo>
\end{syntdiag*}
```

Le dichiarazioni a cui ci riferiamo riguardano il tipo di freccia sinistra e destra. Queste sono dichiarate dai comandi `\left{<tipo>}` e `\right{<tipo>}`, e i tipi ammessi sono riportati di seguito:



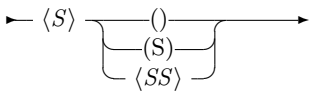
Vediamo finalmente cos'è un diagramma sintattico e cosa permette di rappresentare. Prendiamo sempre il caso della grammatica delle parentesi corrispondenti. Il diagramma sintattico rappresenta graficamente i simboli che formano la grammatica, concatenandoli tramite uno o più percorsi da seguire. Il simbolo non terminale di partenza viene rappresentato all'inizio del diagramma, mentre di seguito riportiamo le produzioni. Essendo quelle della grammatica in esame delle alternative, queste andranno rappresentate una sull'altra, con dei flussi di scelta, simulati dall'ambiente stack. Ogni sovrapposizione deve essere separata da \\. I flussi alternativi sono rivolti all'esterno, a significare che il flusso prosegue il normale percorso dopo aver scelto l'alternativa. Esiste anche la possibilità di rappresentare dei cicli, cioè dei flussi rientranti: l'ambiente necessario è rep.

Quando includiamo il pacchetto possiamo indicare come opzione la forma dei raccordi dei cicli e delle alternative. Scrivendo o no `rounded` otterremo delle linee arrotondate o squadrate.

Rappresentare la grammatica in esame come un diagramma sintattico è semplice, vista anche l'esiguità del sorgente:

```
\begin{syntdiag*}[][4cm]
<S>
\begin{stack}()\\(S)\\<SS>\end{stack}
\end{syntdiag*}
```

e questo è il risultato:



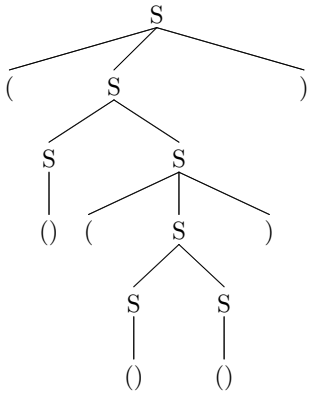
4 Alberi di derivazione

Un albero di derivazione è un formalismo grafico per visualizzare come una stringa di un linguaggio viene derivata dal simbolo non-terminale. Il pacchetto d'elezione per comporre alberi di derivazione è synttree. Il suo uso è molto semplice: l'unico comando `\synttree` prende come parametro la lista dei nomi dei nodi. Il primo nodo da passare è la radice dell'albero, e di seguito i figli. Ogni nodo e relativi figli devono essere racchiusi tra parentesi quadre. Possiamo anche dare, come parametro opzionale, la profondità massima dell'albero.

Come esempio, vediamo come derivare la stringa (()(()())) dal non-terminale. Il seguente codice, indentato per comodità di comprensione,

```
\synttree[S [()
            [S [S [()] ]
               [S [()
                  [S [S [()] ]
                     [S [()] ]
                  ]
                  [)]
               ]
            ]
            [)]
         ]
```

genera il seguente albero di derivazione:



Ora, leggiamo i simboli terminali (le foglie dell'albero) da sinistra a destra: otteniamo la stringa (()(()())), esattamente quella da derivare.

5 Automi

Se possiamo scrivere una grammatica, possiamo convertirla in automa. Il tipo più semplice di automa è il cosiddetto Automa a Stati Finiti (FSA – Finite State Automaton), corrispondente a un linguaggio regolare. Questo automa è del tutto simile ad un grafo con nodi (stati) e archi (transizioni). L'unico cambiamento rilevante è relativo agli stati iniziale e finale, evidenziati rispetto agli altri nodi. Possiamo fare un altro distinguo: in genere la grammatica viene considerata un generatore poiché dalle produzioni generiamo un linguaggio. Un automa viene invece considerato un accettore perché a partire da una stringa possiamo vedere se l'automa giunge allo stato finale (accetta la stringa come facente parte del linguaggio) o no.

Non sono molti i pacchetti a permettere di disegnare automi e grafi in LATEX. A dire la verità, degli unici tre trovati su CTAN uno solo è esattamente quanto sperato. Il primo pacchetto da vedere è finomaton, uno strumento Tcl per il disegno interattivo di automi, con successiva esportazione in METAPOST. Il secondo, gastex, è un vero e proprio pacchetto: fornisce comandi per comporre il grafo descritto. Purtroppo può generare solo immagini PostScript; un documento includente gastex dovrà essere compilato con LATEX, e non con PdfLATEX. Il terzo, dal promettente nome di automata, è disponibile solo per MiKTEX, ma non per TEX Live. Parleremo dei primi due.

5.1 Grafi con finomaton

Come già accennato, finomaton è un editor grafico, scritto in Tcl. Grazie al suo ambiente interattivo possiamo disegnare il grafo secondo il nostro gusto, per poi esportarlo in METAPOST ed includerlo nei nostri documenti. La cosa, naturalmente, comporta dei problemi: li vedremo presto.

Per prima cosa disegneremo un automa. Non sarà quello relativo alla grammatica delle parentesi

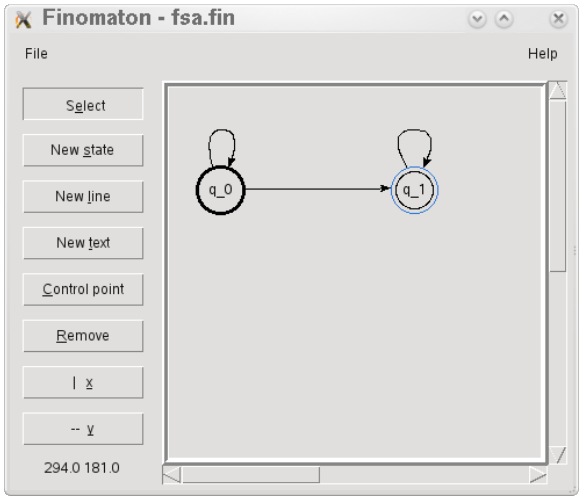


Figura 1: FSA del linguaggio delle parentesi corrispondenti disegnato con finomaton

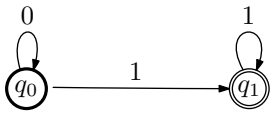


Figura 2: Il grafo convertito in METAPOST e poi in PDF

corrispondenti; questa è una grammatica *context-free* (libera dal contesto), riconosciuta da un *automa push-down*, ma non da un FSA. Potremmo anche disegnare un FSA in grado di riconoscere le stringhe generate dalla grammatica delle parentesi corrispondenti. Quest'automa riconoscerà anche stringhe non generate dalla grammatica, dunque non riconosce tutte e sole le stringhe di quel linguaggio, e quindi non è un accettore del linguaggio generato da quella grammatica. Allora disegneremo il grafo relativo alla seguente grammatica regolare, riconosciuta da un FSA:

$$\begin{aligned}\langle S\rangle ::= \; & 0\langle S\rangle 1\\ \mid \; & \langle S\rangle 1\\ \mid \; & 1\end{aligned}$$

Questa grammatica genera il linguaggio 0^*1^+. In figura 1 vediamo il corrispondente grafo nell'editor.

Una volta finito il disegno, possiamo esportare il file in METAPOST con l'opzione dedicata nel menù File. Otteniamo un file con estensione `.mp`. Per processarlo daremo `mpost <nome_file>.mp`, ottenendo un file PostScript con estensione `.1`. Se intendiamo includere quest'ultimo in un documento da processare con PdfLATEX, dobbiamo assicurarci di aver caricato il supporto a PDF (`\input supp-pdf.tex`), oppure di aver incluso graphics o graphicx, dopodiché possiamo convertire il file (richiede ConTEXt) inserendolo nel punto in cui lo includiamo: `\convertMPtoPDF{<file>}{scala X}{scala Y}`. La soluzione più semplice però consiste nel cambiare estensione al file, da `.1` a `.mps` e fare un'inclusione normale.

Il risultato è visibile in figura 2.

5.2 Grafi con gastex

L'altro pacchetto considerato è GasTeX(Graphs and Automata Simplified in TeX). Se la sezione relativa in TeX Live appare molto deficitaria, senza neanche un manuale (per fortuna il file di stile è ben commentato), il sito internet dedicato (Gastin, 2005) è molto ricco di esempi didattici, con un'ampia miscellanea di grafi e automi.

Disegniamo il solito automa con i comandi di gastex, tenendo presente questo: gli stati iniziale e finale sono qui denotati con archi rispettivamente entrante e uscente, non collegati ad altri nodi.

Attenzione: anche GasTeX non permette di processare i propri documenti con PdfLaTeX, poiché usa PostScript per disegnare ed eseguire calcoli. Il consiglio dell'autore è processare il documento con LaTeX, e poi convertirlo in PDF.

```
\unitlength=4pt
  \begin{picture}(25, 10)(0,-5)
  \gasset{Nw=5,Nh=5,Nmr=2.5,curvedepth=0}
  \thinlines
  \node[Nmarks=i,iangle=90]
       (A1)(0,0){$q_0$}
  \drawloop[loopangle=180](A1){$0$}
  \node[Nmarks=f,fangle=90]
       (A2)(25,0){$q_1$}
  \drawloop[loopangle=0](A2){$1$}
  \drawedge(A1,A2){$1$}
  \end{picture}
```

Il precedente codice è di semplice interpretazione, e contiene quanto basta per disegnare un grafo anche più complesso. Il comando `\node` specifica se il nodo da disegnare è iniziale o finale (*Nmarks=i* | *f*) e qual è la posizione angolare degli archi di inizio/fine (*i/fangle*): nel nostro caso (90°) sono nella parte superiore del nodo; qual è il nome, la posizione e l'etichetta. Il comando `\drawloop` serve a disegnare un autoarco, e specifica la posizione angolare dell'arco (nel nostro caso 0 e 180°, cioè uno alla sinistra e uno alla destra dei rispettivi nodi), il nodo a cui si applica e l'etichetta. Infine `\drawedge` disegna un arco tra due nodi, chiedendo solo i nomi dei nodi e l'etichetta dell'arco. Il comando `\gasset` si incarica di settare le dimensioni comuni: *Nw* e *Nh* sono larghezza e altezza di nodi ovali, *Nmr* è il raggio massimo di tali nodi (in genere Nw/2 o Nh/2); *curvedepth* indica la curvatura di un arco, cioè la distanza verticale tra il punto di massima curvatura dell'arco e la sua proiezione sulla retta unente i due estremi dell'arco; 0 indica un arco rettilineo.

Il risultato del codice è visibile in figura 3.

6 Formattazione di sorgenti

Un programma scritto in un qualsiasi linguaggio di programmazione funziona, in parole povere, grazie alla grammatica sottostante: questa ne assicura il funzionamento coerente. Anche evidenziare le parti del sorgente in base al significato implica evidenziarne le strutture grammaticali. Dunque possiamo vedere quei pacchetti in grado di comporre un sorgente ben formattato (qualcuno lo definirebbe abbellito) come degli strumenti concernenti la grammatica dei sorgenti da trattare.

Finora abbiamo parlato di programmi: sebbene non siano la stessa cosa, includeremo nel discorso anche gli algoritmi e lo pseudocodice.

Discuteremo i pacchetti algorithms, algorithmicx, program e listings.

6.1 Algoritmi e pseudocodice

Il primo strumento a venire in mente per formattare algoritmi è verbatim. Mi sembra ridondante parlarne ancora qui: l'ambiente è una delle prime cose descritte nei manuali, anche per principianti. Il suo compito è comporre il testo letteralmente, e ciò non equivale a formattare un algoritmo, visto anche l'impedimento di comporre eventuali formule in maniera adeguata. Anche usare alltt, o adottare altre soluzioni più casereccie (come inserire spazi arbitrari o parole bianche), non significa formattare un codice. Ricadremmo in uno dei casi non descritti, ma contemplati implicitamente, in Gregorio (2004).

6.1.1 Algoritmi con algorithms

Il pacchetto algorithms ci offre due validi ambienti per formattare pseudocodice. Entrambi fungono da pacchetto, quindi sono questi da includere nel preambolo. L'ambiente algorithmic permette la formattazione di algoritmi e pseudocodice grazie all'adozione di comandi specifici. Questi comandi servono a riprodurre i costrutti tipici dei linguaggi imperativi: affermazioni, cicli e scelte. In tabella 1 sono riassunti i comandi disponibili per l'ambiente citato, comprendenti anche i comandi per pre e post condizioni, commenti, restituzione di valori, valori true e false e stampa di messaggi.

In fase di apertura dell'ambiente possiamo anche passare l'opzione relativa alla frequenza di numerazione delle righe del sorgente. Ad esempio, il seguente sorgente

```
\begin{algorithmic}[2]
\IF{$S \neq \emptyset$}
  \STATE $S = S \setminus e$
\ELSE
  \STATE exit
\ENDIF
\caption{Ricorsione su insiemi}
```

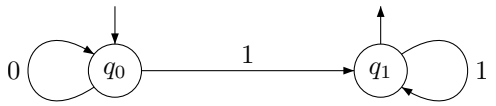


Figura 3: Il grafo di 0^*1^+ disegnato con GasTeX

Tabella 1: Comandi forniti dall'ambiente algorithmic per comporre pseudocodici

Costrutto	Comando LaTeX
Comando	`\STATE <testo>`

Ciclo	`\FOR{<condizione>} <testo> \ENDFOR`
	`\FORALL{<condizione>} <testo> \ENDFOR`
	`\WHILE{<condizione>} <testo> \ENDWHILE`
	`\REPEAT <testo> \UNTIL{<condizione>}`
	`\LOOP <testo> \ENDLOOP`

Scelta	`\IF{<condizione>} <testo> \ENDIF`
	`\IF{<condizione>} <testo> \ELSE <testo> \ENDIF`
	`\IF{<condizione>} <testo> \ELSIF <testo> \ELSE <testo> \ENDIF`

Precondizione	`\REQUIRE <testo>`
Postcondizione	`\ENSURE <testo>`
Commento	`\COMMENT{<testo>}`
Restituzione di valori	`\RETURN <testo>`
Vero	`\TRUE`
Falso	`\FALSE`
Stampa di messaggi	`\PRINT <testo>`

```
\label{alg:alg}
\end{algorithmic}
```

genera il seguente codice:

if $S \neq \emptyset$ **then**
  $S = S \setminus e$
else
  exit
end if

L'altro ambiente citato, algorithm, è il floating body aggiuntivo a quello appena visto. Come tutti i floating body fornisce il noto meccanismo di numerazione; il numero è reso visibile dal comando `\caption` e dal riferimento nel testo; l'etichetta è solo in inglese ma, un minimo di manualità ci permette di cambiarne la lingua:

```
\makeatletter
\renewcommand{ALG@name}{Algoritmo}
\makeatother
```

solo dopo aver incluso il pacchetto nel documento.

Possiamo anche decidere di aggiungere una cornice all'algoritmo con le opzioni *plain*, *boxed* e *ruled* (di default) passate al caricamento del pacchetto; ne vediamo l'esempio di seguito:

```
\usepackage[boxed]{algorithm}
\begin{algorithm}
\begin{algorithmic}
\IF{$S \neq \emptyset$}
  \STATE $S = S \setminus e$
\ELSE
  \STATE exit
\ENDIF
caption{Ricorsione su insiemi}
\label{alg:alg}
\end{algorithmic}
\end{algorithm}
```

if $S \neq \emptyset$ **then**
  $S = S \setminus e$
else
  exit
end if

Algorithm 1: Ricorsione su insiemi

6.1.2 Postilla agli algoritmi

Se avete notato delle minime differenze di composizione tra gli ultimi due algoritmi di esempio e il resto dell'articolo, avete senz'altro un occhio di falco, ma anche assolutamente ragione. In realtà gli algoritmi immersi nel testo sono due documenti PDF, compilati a parte a causa di un conflitto tra i pacchetti algorithmic e program, entrambi inclusi nel sorgente dell'articolo. Entrambi definiscono il comando `\COMMENT`. Quasi rimpiango la moda di anteporre al nome del comando una specie di *namespace*, secondo lo stile *nome_pacchetto.nome_comando*.

6.1.3 Algoritmi alternativi con algorithmicx

In alternativa (sarebbe più corretto dire "in aggiunta", come vedremo) ad algorithms possiamo usare algorithmicx. Questo pacchetto è stato scritto per sopperire ad alcune mancanze in algorithms riscontrate dall'autore del pacchetto in esame. Questo pacchetto non definisce alcun comando per gli algoritmi, ma fornisce delle macro per definirne; fornisce anche alcuni set di comandi da usare direttamente. Questi set (detti layout) sono: *algpseudocode*, molto simile nell'aspetto a quello definito in algorithmic, *algcompatible*, pienamente compatibile col pacchetto preesistente, *algpascal*, tendente a for-

mattare, con tanto di indentazione, un programma Pascal, e *algc*, incompleto, mirante a formattare un programma C. I layout vanno caricati nel preambolo come se fossero pacchetti.

Ogni algoritmo inizia con la dichiarazione `\begin{algorithmic}[lines]` e termina con `\end{algorithmic}`. Con il parametro *lines* definiamo la numerazione del codice: 0 non numera, 1 numera tutte le righe, n numera le righe multiple di n.

Una riga di testo è preceduta dal comando `\State`. Esiste anche la macro `\Statex`; questa definisce una riga non numerabile. I commenti sono definiti grazie a `\Comment`, senza le limitazioni imposte da algorithms. Ad esempio,

```
\begin{algorithmic}[0]
\State x++ \Comment{Incrementa x}
\end{algorithmic}
```

genera

x++ ▷ Incrementa x

ma nessuno ci vieta di ridefinire il commento:

```
\algrenewcommand{\algorithmiccomment}[1]
  {\hskip3em$\rightarrow$ #1}
```

e ottenere

x++ → Incrementa x

Non ci dilungheremo nella descrizione di tutte le caratteristiche del pacchetto: come già detto, ciò esula dai fini dell'articolo; vediamo piuttosto un esempio di algoritmo ottenuto con l'opzione *algpseudocode*: il calcolo ricorsivo del fattoriale.

```
\begin{algorithmic}[0]
\Function{fatt}{x}
\If{x < 0}
  \State exit(0);
\EndIf
\If{x = 0}
  \State return(1);
\Else
  \State return(x * fatt(x - 1));
\EndIf
\EndFunction
\end{algorithmic}
```

genera il seguente pseudocodice già formattato:

function FATT(x)
  if x < 0 **then**
    exit(0);
  end if
  if x = 0 **then**
    return(1);
  else
    return(x * fatt(x - 1));
  end if
end function

Vediamo un riassunto di tutti i costrutti possibili con algpseudocode in tabella 2.

Infine, possiamo usare algorithm del paragrafo 6.1.1 come contenitore per floating body.

6.1.4 Pseudocodici con program

Un altro pacchetto promettente riguardo la formattazione di pseudocodici è program. L'autore afferma di aver programmato la formattazione in base al proprio stile preferito, e non necessariamente deve essere lo stile preferito di altri. Lo stile dei comandi è molto simile a quello visto nel paragrafo 6.1.1, sebbene ridotto e con dei piccoli cambiamenti.

Anche il pacchetto program fornisce due ambienti: program e programbox. Questi sono simili nella sostanza, ma differenti nella forma: il secondo formatta il sorgente in una sola pagina.

In tabella 3 riassumiamo i comandi dell'ambiente in esame.

Dato il seguente pseudocodice ne vediamo la formattazione ad opera di program:

```
\FUNCT |fatt|(x) \BODY
\IF x < 0 \THEN |exit|(0);
\FI
\IF x = 0 \THEN |return|(1);
\ELSE |return|(x * |fatt|(x - 1));
\FI
\ENDFUNCT
```

funct $fatt(x)$ ≡
  if $x < 0$ **then** $exit(0)$;
  fi
  if $x = 0$ **then** $return(1)$;
    else $return(x * fatt(x - 1))$;
  fi
.

6.2 Sorgenti formattati

Il pacchetto listings ci aiuta a risolvere il problema di formattare, evidenziandone le parole chiave, un sorgente scritto in un linguaggio di programmazione. La lista delle opzioni disponibili al caricamento del pacchetto è veramente vasta, ma qui vedremo solo alcune di esse.

La prima cosa da fare quando dobbiamo inserire un codice sorgente in un documento è stabilire il linguaggio del sorgente: `\lstloadlanguages{C,csh}` stabilisce l'intenzione di formattare un sorgente contenente codice C e Cshell. La lista dei linguaggi riconosciuti ne conta circa 80. Il comando visto va inserito nel preambolo del documento.

Quando dobbiamo inserire il sorgente nel documento dobbiamo avere cura di selezionare il linguaggio giusto, nonché l'eventuale dialetto: `\lstset{language=[<dialetto>]<linguaggio>}`. Con lo stesso comando possiamo stabilire la dimensione della tabulazione (`tabsize=<numero>`), lo stile delle parole chiave (ne vedremo un esempio

Tabella 2: Costrutti disponibili con algpseudocode

Costrutto	Risultato

`\For{<testo>}`
`  <corpo>`
`\EndFor`
	for <testo> **do**
	  <corpo>
	end for

`\While{<testo>}`
`  <corpo>`
`\EndWhile`
	while <testo> **do**
	  <corpo>
	end while

`\Repeat`
`  <corpo>`
`\Until{<testo>}`
	repeat
	  <corpo>
	until <testo>

`\If{<testo>}`
`  <corpo>`
`[`
`\ElsIf{<testo>}`
`  <corpo>`
`...`
`]`
`[`
`\Else`
`  <corpo>`
`]`
`\EndIf`
	if <testo> **then**
	  <corpo>
	  [
	else if <testo> **then**
	  <corpo>
	  ...
	  ]
	  [
	else
	  <corpo>
	  ]
	end if

`\Procedure{<nome>}{<parametri>}`
`  <corpo>`
`\EndProcedure`
	procedure <NOME>(<parametri>)
	<corpo>
	end procedure

`\Function{<nome>}{<parametri>}`
`  <corpo>`
`\EndFunction`
	function <NOME>(<parametri>)
	<corpo>
	end function

`\Loop`
`  <body>`
`\EndLoop`
	loop
	  <body>
	end loop

`\Require <testo>`
`\Ensure <testo>`
`\Statex`
`\State <istruzione>`
`\Call{<nome>}{<parametro>}`
`\Comment <testo>`
	Require: <testo>
	Ensure: <testo>
	
	<istruzione>
	  <NOME>(<parametro>)
	  → <testo>

Tabella 3: Riassunto dei costrutti disponibili grazie al pacchetto program

Costrutto	Significato e uso
`\FUNCT |<nome>|(x) \BODY \ENDFUNCT`	Delimitatori di funzione: questa va scritta dopo `\BODY`
`\PROC |<nome>|(x) \BODY \ENDPROC`	Delimitatori di procedura
`\IF <condizione> \THEN <azione> \ELSE <azione> \FI`	Delimitatori di scelta; la sezione `\ELSE` è opzionale
`\WHILE <condizione> \DO <istruzioni> \OD`	Delimitatori di ciclo

a breve), la visibilità degli spazi e delle tabulazioni (`showspace=true` o `false`, `showtabs=true` o `false`, `tab=<comando>`), la numerazione delle righe. Demandiamo la visione di questa e delle altre caratteristiche modificabili alla lettura del manuale del pacchetto, esulando queste (e anche qualcuna di quelle descritte) dalle finalità dell'articolo.

Per formattare il documento useremo l'ambiente lstlisting. La sintassi è

```
\begin{lstlisting}[<opzioni>]{<parametro>}
  <sorgente>
\end{lstlisting}
```

Il parametro obbligatorio è un'etichetta: tutti i listati nel documento che hanno la stessa etichetta avranno lo stesso contatore di righe, e quindi manterranno la continuità della numerazione delle righe di codice.

Tra le opzioni abbiamo *caption* per le didascalie (etichetta solo in inglese; vale un'osservazione analoga fatta per algorithm, tesa però a modificare `\lst@UserCommand\lstlistingname{Listing}`); *captionpos* per posizionare le didascalie (valori *t* o *b*), *float* per rendere il listato un floating body.

È ora di vedere un esempio: il codice seguente

```
\lstset{language=[plain]TeX}
\begin{lstlisting}
\def\toclevel@title{0}
\let\@oldtoc\tableofcontents
\def\BlackBoxes{\overfullrule=5\p@}
\def\NoBlackBoxes{\overfullrule=\z@}
\def\ATname{\Ars}
\newcommand*\pkgname[1]{\textsf{#1}}
\newcommand*\clsname[1]{\textsf{#1}}
\newcommand*\optname[1]{\textsf{#1}}
\newcommand*\envname[1]{\textsf{#1}}
\DeclareRobustCommand\cmdname[1]
  {\texttt{\char`\\#1}}
\DeclareRobustCommand\meta[1]{%
  \ensuremath{\langle}\emph{#1}
  \ensuremath{\rangle}}
\def\@cfgfound{\ClassInfo{arstexnica}
  {Info loaded from \jobname.cfg.}}
\def\@cfgnotfound{\ClassWarning
  {arstexnica}
  {Config file not found.}}

\def\@usenempty#1{%
  \csname #1\endcsname
  \global\expandafter\let\csname
    #1\endcsname\@empty
}
\end{lstlisting}
```

fornisce questo output:

```
\def\toclevel@title{0}
\let\@oldtoc\tableofcontents
\def\BlackBoxes{\overfullrule=5\p@}
\def\NoBlackBoxes{\overfullrule=\z@}
\def\ATname{\Ars}
\newcommand*\pkgname[1]{\textsf{#1}}
\newcommand*\clsname[1]{\textsf{#1}}
\newcommand*\optname[1]{\textsf{#1}}
\newcommand*\envname[1]{\textsf{#1}}
\DeclareRobustCommand\cmdname[1]
  {\texttt{\char`\\#1}}
\DeclareRobustCommand\meta[1]{%
  \ensuremath{\langle}\emph{#1}
  \ensuremath{\rangle}}
\def\@cfgfound{\ClassInfo{arstexnica}
  {Info loaded from \jobname.cfg.}}
\def\@cfgnotfound{\ClassWarning
  {arstexnica}
  {Config file not found.}}

\def\@usenempty#1{%
  \csname #1\endcsname
  \global\expandafter\let\csname
    #1\endcsname\@empty
}
```

Supponiamo di voler cambiare aspetto della sola parola chiave *def*. Dovremo sempre passare uno o più parametri a `\lstset`. Nel nostro caso, volendo la parola chiave colorata, basterà specificare, prima del sorgente, lo stile da applicare, ad esempio:

```
\lstset{emph={def},
  emphstyle=\color{blue}}
```

Ecco lo stesso sorgente secondo il nuovo stile:

```
\def\toclevel@title{0}
\let\@oldtoc\tableofcontents
\def\BlackBoxes{\overfullrule=5\p@}
\def\NoBlackBoxes{\overfullrule=\z@}
\def\ATname{\Ars}
\newcommand*\pkgname[1]{\textsf{#1}}
\newcommand*\clsname[1]{\textsf{#1}}
\newcommand*\optname[1]{\textsf{#1}}
\newcommand*\envname[1]{\textsf{#1}}
\DeclareRobustCommand\cmdname[1]
  {\texttt{\char`\\#1}}
\DeclareRobustCommand\meta[1]{%
  \ensuremath{\langle}\emph{#1}
  \ensuremath{\rangle}}
\def\@cfgfound{\ClassInfo{arstexnica}
  {Info loaded from \jobname.cfg.}}
\def\@cfgnotfound{\ClassWarning
  {arstexnica}
  {Config file not found.}}

\def\@usenempty#1{%
  \csname #1\endcsname
  \global\expandafter\let\csname
    #1\endcsname\@empty
}
```

Il pacchetto ci permette altre due utili funzioni: la prima, `\lstinline`, rende possibile l'inserimento

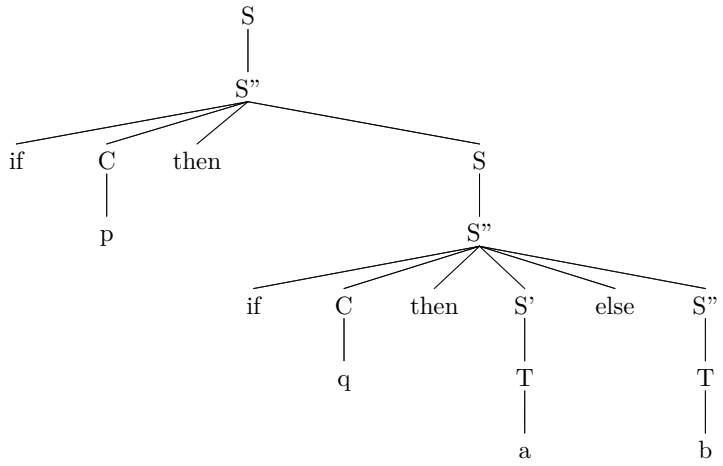


Figura 4: Albero di derivazione per if-then-else

inline di un comando, allo stesso modo di `\verb`; la seconda, `\lstinputlisting`, ci permette di inserire dei sorgenti nel documento senza copiarceli fisicamente.

6.3 Supplemento grammaticale

All'inizio del capitolo abbiamo fatto cadere dall'alto la similitudine "sorgente evidenziato – grammatica evidenziata". La cosa potrebbe sembrare strana a qualche persona; per ovviare a ciò proponiamo un esempio classico: if-then-else.

Si abbia la seguente grammatica:

$\langle S \rangle$ ::= $\langle S' \rangle$
| $\langle S'' \rangle$

$\langle S' \rangle$ ::= if $\langle C \rangle$ then $\langle S' \rangle$ else $\langle S'' \rangle$
| T

$\langle S'' \rangle$ ::= if $\langle C \rangle$ then $\langle S \rangle$
| if $\langle C \rangle$ then $\langle S' \rangle$ else $\langle S'' \rangle$
| T

$\langle C \rangle$ ::= p
| q

$\langle T \rangle$ ::= a
| b

con S, S', S'' e C simboli non-terminali (rispettivamente affermazione/Statement e condizione/Condition), a e b comandi terminali, p e q condizioni terminali, e *if*, *then* e *else* parole riservate terminali.

Questa grammatica genera una sola possibile derivazione della stringa `if p then if q then a else b`, come vediamo in figura 4.

Avere grammatiche non ambigue, cioè in grado di generare un solo albero di derivazione per istruzione, è essenziale per avere specificazioni sintattiche ben definite; al di là di questo fatto, pure importante nella teoria dei linguaggi di programmazione, possiamo rimarcare quanto segue: riuscire ad evidenziare un sorgente equivale ad effettuarne il parsing, e dunque a riconoscere il sorgente come scritto nel linguaggio scelto. Dunque la composizione di sorgenti non è fuori tema rispetto all'argomento trattato.

7 Conclusioni

Nel corso dell'articolo abbiamo introdotto alcuni argomenti ostici anche per molti informatici, ma fondamentali per inquadrare la portata dell'articolo stesso. Per ognuno degli argomenti (grammatiche, diagrammi sintattici, alberi di derivazione, automi a stati finiti, formattazione di algoritmi e sorgenti di programmi) abbiamo descritto almeno un pacchetto in grado di aggiungere a LaTeX la capacità di comporre gli elementi citati.

Semmai ce ne fosse bisogno, sapremmo da dove partire per i nostri futuri documenti su grammatiche e linguaggi formali.

8 Ringraziamenti

Un doveroso ringraziamento al Comitato Scientifico del GuIT e ai revisori: con i loro suggerimenti e osservazioni hanno notevolmente migliorato la qualità dell'articolo.

Un grazie sentito a Enrico Bini, senza il cui aiuto non sarei stato in grado di presentare il lavoro al meeting.

Oltre a dover ringraziare tutti gli autori dei pacchetti citati, ringrazio e mi scuso con Pallavicini e Conte (e Jannacci), inconsapevoli e incolpevoli ispiratori del titolo.

Riferimenti bibliografici

Arbib, M., Kfoury, A. e Moll, R. (1981). *A Basis for Theoretical Computer Science*. Springer, New York, N.Y.

Gastin, P. (2005). «GasTeX». URL http://www.lsv.ens-cachan.fr/~gastin/gastex/gastex.html.

Gregorio, E. (2004). «Orrori in LaTeX: come maltrattare LaTeX e rendere infelice un *copy editor*». In *Atti del GuIT meeting*. Pisa.

Kfoury, A., Moll, R. e Arbib, M. (1982). *A Programming Approach to Computability*. Springer, New York, N.Y.

Moll, R., Arbib, M. e Kfoury, A. (1988). *An Introduction to Formal Language Theory*. Springer, New York, N.Y.

Pignalberi, G. (2004a). «LaTeX, diagrammi sintattici e grammatiche». *Dev*, (122).

— (2004b). «LaTeX e la formattazione di pseudocodice e sorgenti». *Dev*, (123).

▷ Gianluca Pignalberi
Free Software Magazine
g dot pignalberi at freesoftwaremagazine dot com

▷ Enrico Bini
Scuola Superiore Sant'Anna
Pisa, Italia
e dot bini at sssup dot it