Supplemento alle dispense di Sintassi

Luca Tesei

20 ottobre 2002

1 Formalizzazione

Lo scopo di questa sezione è quello di presentare in maniera formale e precisa le nozioni di Automa e di Grammatica Libera dal Contesto che abbiamo già visto nella prima parte del corso. Per far questo ci baseremo sui concetti che abbiamo già introdotto e preciseremo quelli che non sono stati formalizzati nelle dispense di Sintassi.

Sia A un insieme. Il *powerset* di A è l'insieme che contiene tutti i sottoinsiemi dell'insieme A. Lo indicheremo con $\mathcal{P}(A) = \{B \mid B \subseteq A\}$. Si noti che qualunque sia A, $\emptyset \in \mathcal{P}(A)$ e $A \in \mathcal{P}(A)$.

Definizione 1 (Prodotto Cartesiano) *Siano A e B due insiemi. Il* prodotto cartesiano *fra A e B si denota $A \times B$ ed è l'insieme di coppie ordinate di elementi di A e B.*

$$A \times B = \{(a, b) \mid a \in A, b \in B\}$$

Si noti che le coppie sono ordinate e quindi $(a, b) \neq (b, a)$.

La definizione di prodotto cartesiano si estende naturalmente a più insiemi:

$$A_1 \times A_2 \times \cdots \times A_n = \{(a_1, a_2, \ldots, a_n) \mid \forall i \in [1, n].a_i \in A_i\}$$

Gli elementi di $A_1 \times A_2 \times \cdots \times A_n$ si chiamano ennuple (n-ple).

1.1 Automi

Sia Λ un insieme finito di simboli che rappresenta l'alfabeto dell'automa.

Definizione 2 (Automa) *Un* Automa *su Λ è una tupla $\langle Q, \Lambda, q_0, \mathcal{F}, \mathcal{E}\rangle$ dove:*

- *Q è un insieme finito di* stati

- Λ *è l'alfabeto dei simboli*
- $q_0 \in Q$ *è lo* stato iniziale
- $\mathcal{F} \subseteq Q$ *è l'insieme degli* stati di accettazione *o* stati finali
- $\mathcal{E} \subseteq (Q \times \mathcal{P}(\Lambda) \times Q)$ *è l'insieme finito di* transizioni.

Ogni transizione di $\mathcal{E}$ è della forma (q, L, q') dove q è lo stato di partenza della transizione, q' è lo stato di arrivo e L è un insieme di simboli di Λ che etichettano la transizione.

Vogliamo ora esprimere la nozione di cammino etichettato e di stringa accettata.

Definizione 3 (Cammino etichettato) *Sia $A = \langle Q, \Lambda, q_0, \mathcal{F}, \mathcal{E} \rangle$ un automa. Un cammino etichettato di lunghezza $k \geq 0$ su A è una sequenza $q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_2} q_2 \xrightarrow{x_3} \cdots \xrightarrow{x_k} q_k$ dove:*

- q_0 *è lo stato iniziale*
- $\forall i \in [1, k].(q_{i-1}, L_i, q_i) \in \mathcal{E} \wedge x_i \in L_i$

La stringa $x_1 x_2 \cdots x_k$ si chiama stringa associata al cammino *ed è in generale una stringa di Λ^*. Se $k = 0$ allora il cammino è costituito solo dallo stato iniziale e la stringa associata è la stringa vuota ϵ.*

Si noti che un cammino di lunghezza k consiste di una sequenza di $k + 1$ stati dell'automa ed ha una stringa associata lunga k.

Definizione 4 (Stringa accettata) *Sia $A = \langle Q, \Lambda, q_0, \mathcal{F}, \mathcal{E} \rangle$ un automa. Una stringa $\alpha \in \Lambda^*$ è accettata dall'automa A* **se e solo se** *esiste un cammino etichettato $q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_2} q_2 \xrightarrow{x_3} \cdots \xrightarrow{x_k} q_k$ su A tale che α è la stringa associata al cammino e lo stato q_k è uno stato finale (in formule $q_k \in \mathcal{F} \wedge \alpha = x_1 x_2 \cdots x_k$). Se lo stato iniziale è di accettazione allora anche la stringa vuota ϵ è accettata dall'automa.*

Possiamo ora introdurre la nozione di linguaggio accettato dall'automa.

Definizione 5 (Linguaggio accettato) *Sia $A = \langle Q, \Lambda, q_0, \mathcal{F}, \mathcal{E} \rangle$ un automa. Il linguaggio accettato dall'automa è l'insieme*

$$L(A) = \{\alpha \in \Lambda^* \mid \alpha \text{ è accettata da } A\}$$

Si noti che le definizioni precedenti si possono applicare sia al caso di automi deterministici sia a quello di automi non deterministici. Questo perché la nozione di stringa accettata richiede semplicemente l'esistenza di un cammino etichettato e non fa distinzione fra il caso in cui tale cammino, se esiste, è unico (caso deterministico) e il caso in cui possono esserci diversi cammini con la stessa stringa associata (caso non deterministico).

La definizione di automa che abbiamo dato è quella più generale, cioè quella di automa non deterministico. Diamo ora una caratterizzazione degli automi deterministici in questo contesto formalizzato.

Definizione 6 (Automa deterministico) *Sia $A = \langle Q, \Lambda, q_0, \mathcal{F}, \mathcal{E} \rangle$ un automa. A è un* automa deterministico **se e solo se** *per ogni stato $q \in Q$ tutti gli insiemi di etichette L_i delle transizioni $(q, L_i, q_i') \in \mathcal{E}$ che hanno q come stato di partenza hanno intersezione vuota. In formule:*

$$\forall q \in Q. \quad \bigcap_{(q, L_i, q_i) \in \mathcal{E}} L_i \quad = \quad \emptyset$$

Esempio 1 *Consideriamo l'automa di figura 6 nelle dispense di Sintassi. Si ha:*

- $Q = \{0, 1, 2, 3\}$
- $\Lambda = \{a, b, c, \ldots, z\}$
- $q_0 = 0$
- $\mathcal{F} = \{3\}$
- $\mathcal{E} = \{(0, \Lambda, 0), (0, \{m\}, 1), (1, \{a\}, 2), (2, \{n\}, 3), (3, \Lambda, 3)\}$

L'automa è non deterministico poiché gli insiemi di etichette associati alle transizioni che escono dallo stato 0 hanno una intersezione non vuota: $\Lambda \cap \{m\} = \{m\}$. Ci sono due cammini etichettati per la stringa `command`*:*

1. $0 \xrightarrow{c} 0 \xrightarrow{o} 0 \xrightarrow{m} 0 \xrightarrow{m} 0 \xrightarrow{a} 0 \xrightarrow{n} 0 \xrightarrow{d} 0$
2. $0 \xrightarrow{c} 0 \xrightarrow{o} 0 \xrightarrow{m} 0 \xrightarrow{m} 1 \xrightarrow{a} 2 \xrightarrow{n} 3 \xrightarrow{d} 3$

La stringa `command` *è accettata poiché esiste un cammino che ce l'ha come stringa associata e termina in uno stato di accettazione (il cammino 2!).*

1.2 Grammatiche libere dal contesto

Diamo ora una definizione precisa di grammatica libera dal contesto.

Definizione 7 *Una* Grammatica libera dal contesto *è una tupla* $G = \langle \Lambda, V, S, P \rangle$ *dove:*

- Λ *è l'insieme finito dei simboli dell'alfabeto o* Simboli Terminali
- V *è l'insieme finito dei simboli che rappresentano* Categorie Sintattiche *(a volte vengono chiamati anche Simboli Non Terminali)*
- $S \in V$ *è il simbolo di Categoria Sintattica* Iniziale *o* Principale
- P *è l'insieme finito delle* Produzioni *della forma*

  $$Y \longrightarrow X_1 X_2 \cdots X_k$$

  dove:

  - $Y \in V$ *è la* testa *della produzione*
  - $\forall i \in [1, k].X_i \in (V \cup \Lambda)$. *La stringa* $X_1 X_2 \cdots X_k \in (V \cup \Lambda)^*$ *si chiama* corpo *della produzione.*

  Il corpo di una produzione può anche essere vuoto. In questo caso la produzione viene scritta $Y \longrightarrow \epsilon$.

Definiamo ora il concetto di *derivazione* di una stringa a partire da un simbolo di categoria sintattica. Questa nozione è un'alternativa a quella degli alberi di derivazione. Anche essa può essere usata come rappresentazione della "dimostrazione" che una certa stringa appartiene al linguaggio generato da un simbolo di categoria sintattica.

Sia $G = \langle \Lambda, V, S, P \rangle$ una grammatica libera dal contesto. Sia $\alpha \in (\Lambda \cup V)^*$ una stringa che contiene almeno un simbolo di Categoria Sintattica Y. Allora possiamo scrivere $\alpha = \delta Y \gamma$ con $\delta, \gamma \in (\Lambda \cup V)^*$. Possiamo applicare ad α un passo di derivazione utilizzando una produzione di P che ha Y come testa. Il passo consiste nel riscrivere α sostituendo al simbolo di categoria sintattica Y che abbiamo messo in evidenza il corpo della produzione scelta.

Definizione 8 (Passo di Derivazione) *Sia* $G = \langle \Lambda, V, S, P \rangle$ *una grammatica libera dal contesto e sia* $\alpha \in (\Lambda \cup V)^*$ *una stringa che contiene almeno un simbolo di categoria sintattica* Y. *Sia* $Y \longrightarrow X_1 X_2 \cdots X_k \in P$, *allora possiamo riscrivere*

$$\alpha = \delta Y \gamma \Longrightarrow_G \delta X_1 X_2 \cdots X_k \gamma$$

Il simbolo $\Longrightarrow_G$ *rappresenta un* passo di derivazione *per la grammatica* G.

Definizione 9 (Derivazione) *Sia $G = \langle \Lambda, V, S, P \rangle$ una grammatica libera dal contesto. Sia $Y \in V$ un simbolo di categoria sintattica di G. Una derivazione di una stringa $s \in \Lambda^*$ a partire da Y è una sequenza*

$$Y = \alpha_0 \Longrightarrow_G \alpha_1 \Longrightarrow_G \alpha_2 \cdots \Longrightarrow_G \alpha_n = s$$

dove $\forall i \in [0, n-1].\alpha_i \Longrightarrow_G \alpha_{i+1}$ è un passo di derivazione che riscrive un qualche simbolo di categoria sintattica in α_i. Si noti che se $i < n$ allora $\alpha_i \in (\Lambda \cup V)^*$ e invece $\alpha_n \in \Lambda^*$.

Una derivazione lunga 0 passi è la sequenza composta solo dal simbolo di categoria sintattica Y.

Per indicare che esiste una derivazione di zero o più passi da Y ad una certa stringa α scriviamo $Y \overset{*}{\Longrightarrow}_G \alpha$, mentre $Y \overset{+}{\Longrightarrow}_G \alpha$ indica che esiste una derivazione lunga almeno 1 passo da Y a α.

Non ci resta che definire allora il linguaggio generato da un simbolo di categoria sintattica e quindi quello generato da una grammatica.

Definizione 10 (Linguaggio generato) *Sia $G = \langle \Lambda, V, S, P \rangle$ una grammatica libera dal contesto. Sia $Y \in V$ un simbolo di categoria sintattica. Il linguaggio generato da Y, $L(Y)$, è l'insieme di tutte le stringhe derivabili da Y:*

$$L(Y) = \{s \in \Lambda^* \mid Y \overset{*}{\Longrightarrow}_G s\}$$

Il linguaggio generato dalla grammatica G, $L(G)$, corrisponde al linguaggio generato dal simbolo di categoria sintattica iniziale:

$$L(G) = L(S)$$

Si noti che la definizione precedente è equivalente a quella data usando gli alberi di derivazione, nel senso che l'insieme di stringhe definito con le derivazioni è lo stesso di quello definito con gli alberi di derivazione. Data una certa grammatica, ad ogni stringa che ha un albero di derivazione possono corrispondere più derivazioni. Questo indipendentemente dall'ambiguità della grammatica. In altre parole una grammatica può essere non ambigua e avere più derivazioni per la stessa stringa. Si può definire una regola di derivazione (per esempio la regola che dice di riscrivere sempre il simbolo di categoria sintattica più a destra in ogni α_i, oppure il simbolo più a sinistra) che associa ad ogni albero di derivazione una e una sola sequenza di derivazione. In questo modo solo se la grammatica è ambigua si avranno più sequenze di derivazione per la stessa stringa.

Esempio 2 *Consideriamo la grammatica non ambigua per le espressioni aritmetiche in Figura 33 e chiamiamola G. Si ha:*

- $\Lambda = \{0, 1, \ldots, 9, (,), +, *, -, /\}$
- $V = \{< E >, < T >, < F >, < N >, < C >\}$
- $S =< E >$
- $P = \{< E > \longrightarrow < E > + < T >, < E > \longrightarrow < E > - < T >, < E > \longrightarrow < T >, < T > \longrightarrow < T > * < F >, < T > \longrightarrow < T > / < F >, \ldots\}$

*Proviamo a derivare la stringa 1+2*3 il cui albero di derivazione è mostrato in Figura 35. Ad ogni passo la categoria sintattica scelta per essere riscritta è evidenziata in grassetto.*
$< \mathbf{E} > \Longrightarrow_G < \mathbf{E} > + < T > \Longrightarrow_G < \mathbf{T} > + < T > \Longrightarrow_G < \mathbf{F} > + < T > \Longrightarrow_G < \mathbf{N} > + < T > \Longrightarrow_G < \mathbf{C} > + < T > \Longrightarrow_G 1 + < \mathbf{T} > \Longrightarrow_G 1 + < \mathbf{T} > * < F > \Longrightarrow_G 1 + < \mathbf{F} > * < F > \Longrightarrow_G 1 + < \mathbf{N} > * < F > \Longrightarrow_G 1 + < \mathbf{C} > * < F > \Longrightarrow_G 1 + 2 * < \mathbf{F} > \Longrightarrow_G 1 + 2 * < \mathbf{N} > \Longrightarrow_G 1 + 2 * < \mathbf{C} > \Longrightarrow_G 1 + 2 * 3.$
Come si vede nella derivazione precedente si è seguita una regola ben precisa per scegliere quale simbolo di categoria sintattica riscrivere ad ogni passo: si è scelto sempre il primo simbolo non terminale della stringa, ovvero il simbolo di categoria sintattica più a sinistra. Seguendo questa regola la stringa $1 + 2 * 3$ *ha una unica derivazione che è quella che abbiamo visto. Tuttavia la definizione generale di derivazione non prescrive di seguire nessuna regola quindi ad esempio anche la seguente è una derivazione legale per* $1 + 2 * 3$*:*
$< \mathbf{E} > \Longrightarrow_G < E > + < \mathbf{T} > \Longrightarrow_G < E > + < T > \Longrightarrow_G < E > + < \mathbf{T} > * < F > \Longrightarrow_G < \mathbf{E} > + < F > * < F > \Longrightarrow_G < \mathbf{T} > + < F > * < F > \Longrightarrow_G < F > + < F > * < F > \overset{+}{\Longrightarrow}_G 1 + 2 * 3.$

2 Applicazioni degli automi e delle grammatiche libere

In questa sezione inquadreremo in maniera semplice e discorsiva il ruolo che possono avere gli automi e le grammatiche nelle applicazioni informatiche vere e proprie. Il tipo di applicazione che tipicamente usa i formalismi che abbiamo studiato è il *compilatore*. È da notare che tutti gli algoritmi che conosciamo sugli automi e sulle grammatiche libere sono stati scritti proprio per risolvere i problemi della compilazione.

In Figura 1 vediamo una schematizzazione semplice di un compilatore ovvero di una applicazione che prende in input un programma scritto in un certo linguaggio di programmazione e produce un programma equivalente

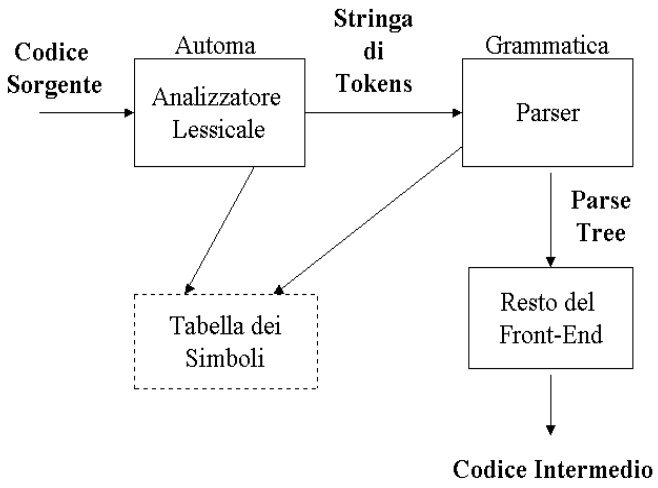

Figura 1: Struttura semplificata di un compilatore

scritto però in linguaggio macchina o in un linguaggio intermedio di livello basso (che può facilmente essere tradotto in linguaggio macchina o che possa essere interpretato efficientemente). Il primo modulo del compilatore si chiama Analizzatore Lessicale e vediamo che ha come input semplicemente il programma che si vuole compilare. Tipicamente il programma è una stringa di ASCII*, cioè una stringa di caratteri ASCII (quelli riproducibili da una tastiera). Il ruolo dell'analizzatore lessicale è quello di raggruppare questi simboli in gruppi e di fornire quindi una sequenza di gruppi di simboli chiamati *token*. Esempi di token sono gli identificatori (x, pippo, a23,...), i numeri (0, 123, -676.567, 456.987E-15, ...), gli operatori (<=, :=, +,...) e le parole riservate del linguaggio (ad esempio nel Pascal `if`, `then`, `procedure`,...). Per individuare i token a partire dal programma l'analizzatore lessicale usa un automa che tenta di riconoscere i token previsti dal linguaggio.

La sequenza di token prodotta dall'analizzatore lessicale viene inviata come input al secondo modulo del compilatore che si chiama *Parser*. Il parser si occupa di decidere se la sequenza di token è una sequenza corretta rispetto alla grammatica del linguaggio in questione. In altre parole il parser tenta di costruire l'albero di derivazione della sequenza di token a partire dalle produzioni della grammatica (libera) data per il linguaggio. Esistono tipi di parser diversi a seconda del tipo di grammatica libera che definisce il linguaggio. Alcune grammatiche libere possono essere facilmente analizzate Top-Down, cioè si cerca di costruire l'albero di derivazione a partire dalla ra-

dice e cercando di "indovinare" le produzioni da applicare via via guardando i token in input. Un altro modo è quello di procedere Bottom-Up: l'albero di derivazione è costruito dal basso allo stesso modo di come abbiamo visto nella sezione 3.1 delle dispense di Sintassi. In tutti casi l'analisi è problematica se la grammatica è ambigua. In alcuni casi si riesce a disambiguare la grammatica, in altri semplicemente se c'è un conflitto ad un certo stadio di costruzione dell'albero di derivazione si applica una regola generale per eliminare alcune scelte e costruire un solo albero di derivazione per la stringa data.

Alcuni vincoli sintattici dei linguaggi di programmazione non si possono esprimere con grammatiche libere dal contesto. Ad esempio si pensi al vincolo tipico che un identificatore non può essere usato se prima non è stato dichiarato. Questo tipo di vincoli non può essere espresso da grammatiche libere e per questo il compilatore utilizza delle strutture ausiliarie come la *tabella dei simboli* che viene arricchita via via che nuovi indentificatori appaiono nel programma e che viene poi utilizzata successivamente per controllare se un identificatore che si sta usando è stato dichiarato e/o inizializzato.

Se il programma dato in input al compilatore è sintatticamente corretto il parser riesce a costruire l'albero di derivazione che lo rappresenta. A questo punto l'albero può essere facilmente usato per generare un codice intermedio o il codice macchina (può seguire anche una fase di ottimizzazione). La semplicità della generazione del codice è dovuta al fatto che l'albero di derivazione riflette esattamente la struttura del programma e quindi semplicemente visitando l'albero in un certo ordine si genera un po' di codice per ogni nodo a seconda del costrutto che vi troviamo.

Esistono diversi tool che implementano gli algoritmi per l'analizzatore lessicale e per il parser. Un generatore di analizzatori lessicali famoso si chiama `lex` ed è in genere disponibile in ogni piattaforma Unix o Linux. Esso prende in input delle espressioni regolari (che sono un modo non grafico di rappresentare gli automi) e produce un programma in linguaggio C che si comporta come l'analizzatore lessicale sui token specificati con le espressioni regolari. Un generatore di parser è ad esempio `yacc` o `byacc`. Esso prende in input una grammatica libera e produce un programma in linguaggio C che analizza una stringa di token e costruisce, se esiste, un albero di derivazione per la stringa a seconda della grammatica fornita. Anche questo tool si trova tipicamente in tutte le piattaforme Unix o Linux. Oltre a queste cose fondamentatali questi tool permettono anche di fare altre cose utili in un contesto di traduzione/compilazione. Si rimanda alla documentazione dei tool per scoprirne tutte le potenzialità. Chiaramente se la grammatica data è ambigua o non rientra in certe classi di grammatiche libere potrebbero appalesarsi dei conflitti in qualche punto dell'analisi. I tool segnalano il

punto preciso dove si genera il conflitto e sono per questo molto utili anche in fase di progettazione della grammatica.

3 Un costrutto ambiguo

In questa sezione consideriamo un esempio classico di ambiguità e successiva disambiguazione. Il costrutto che produce il problema è uno dei più usati in tutti i linguaggi di programmazione: il condizionale.

L'ambiguità nasce dal fatto che un costrutto condizionale può avere un solo ramo oppure può essere completo e avere due rami. Prendiamo il seguente spezzone di grammatica del Pascal:

$< C >$	$\longrightarrow$	**if** $< E >$ **then** $< C >$
	|	**if** $< E >$ **then** $< C >$ **else** $< C >$
	|	**altri_comandi**

Consideriamo il comando
if $< E_1 >$ **then if** $< E_2 >$ **then** $< C_1 >$ **else** $< C_2 >$
dove supponiamo che i simboli di categoria sintattica corrispondono a costrutti corretti. Possiamo individuare due alberi di derivazione per il comando dato. Ciò è dovuto al fatto che la grammatica permette di specificare comandi condizionali con e senza **else** a piacimento.

Nelle Figure 2 e 3 sono mostrati i due alberi di derivazione diversi per la stringa data. Quindi la grammatica che abbiamo scritto è ambigua. Quello che si fa nel Pascal e anche in altri linguaggi di programmazione simili è quello di fissare una regola che permetta di interpretare le stringhe come quella che abbiamo considerato in una maniera univoca. La regola che si segue dice che ogni **else** deve essere associato al **then** non associato più vicino. Quindi l'albero di derivazione "giusto" è quello di Figura 2.

Questa regola di associazione può essere incorporata direttamente nella grammatica disambiguandola. L'idea è quella di dividere i comandi in due tipologie:

1. i comandi "matched", rappresentati dalla categoria sintattica < MC >, che sono i comandi in cui tutti i **then** hanno un **else** associato oppure non sono comandi condizionali (categoria sintattica < ALTRI >)

2. i comandi "unmatched", rappresentati dalla categoria sintattica < UC >, che sono tutti i comandi condizionali che hanno solo il ramo **then** seguito da qualsiasi comando oppure che hanno anche un ramo **else**, ma

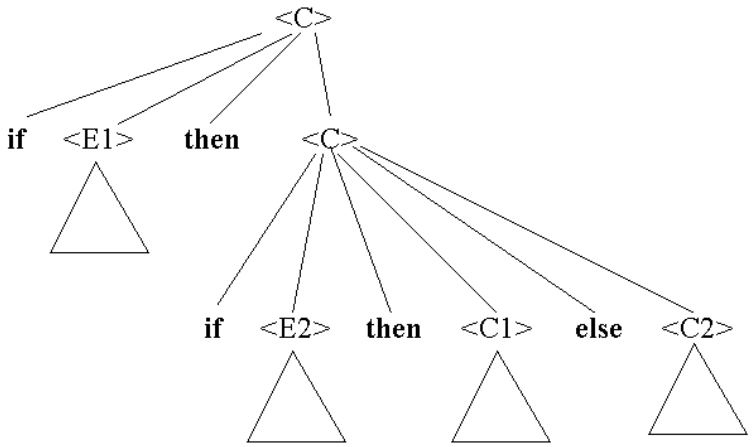

Figura 2: Un albero di derivazione per la stringa

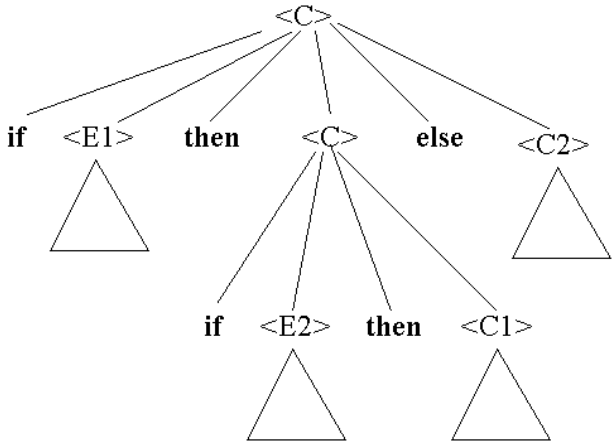

Figura 3: Un altro albero di derivazione per la stessa stringa

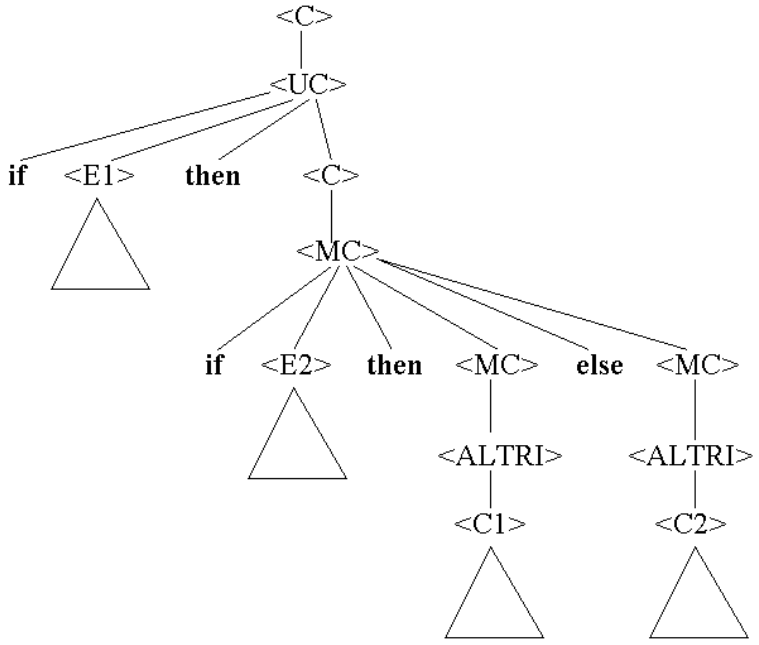

Figura 4: L'unico albero di derivazione per la stringa con la nuova grammatica

in questo caso il comando fra il **then** e l'**else** è un comando "matched" e il comando dopo l'**else** è "unmatched".

In questo modo ogni comando che appare tra un **then** ed un **else** non può finire con un **then** non associato seguito da un altro comando qualsiasi. Ecco la nuova grammatica:

$$
\begin{array}{rcl}
< C > & \longrightarrow & < MC > \mid < UC > \\
< MC > & \longrightarrow & \textbf{if} < E > \textbf{then} < MC > \textbf{else} < MC > \\
 & \mid & < \text{ALTRI} > \\
< UC > & \longrightarrow & \textbf{if} < E > \textbf{then} < C > \\
 & \mid & \textbf{if} < E > \textbf{then} < MC > \textbf{else} < UC >
\end{array}
$$

In Figura 4 è disegnato l'unico albero di derivazione per la stringa incriminata. Questa volta non si può costruire un albero simile a quello di Figura 3 perché fra il primo **then** e l'**else** non possiamo mettere un comando "unmatched" come un condizionale senza l'**else**.