Riassuntissimo
Linguaggi e Modelli Computazionali LS

Silvia Cereda
silvia.cereda@studio.unibo.it

Indice

1 Grammatiche

1.1 Classificazione secondo Chomsky

Questa classificazione definisce le grammatiche in maniera gerarchica, ovvero una grammatica di tipo superiore ò anche un caso particolare della grammatica di tipo inferiore.

Tipo 0: non ci sono restrizioni, ovvero qualunque grammatica può essere di tipo 0. Sono ammesse anche produzioni che accorciano la forma della frase corrente.

Tipo 1: tutte le grammatiche che non ammettono la stringa vuota e che nelle loro produzioni non accordiano mai la lunghezza della frase.

Tipo 2: (Context-free) queste grammatiche, indipendentemente dal contesto, hanno produzioni di forma $A \to \alpha$; ovvero non esiste più:

$$S \to aBCD|aSBCD$$

$$aB \to ab$$

$$aBC \to abB$$

Non c'è più la possibilità di scegliere una volta incontrato un simbolo VN.

Una grammatica di tipo 2 ò detta ambigua se esiste almeno una frase del linguaggio che ammette più di una derivazione canonica sinistra (partendo dallo scopo, riscrivo sempre il simbolo non-terminale più a sinistra), oppure esistono più alberi sintattici distinti.

Tipo 3: (Grammatiche regolari) sono le grammatiche regolari a destra o a sinistra, ovvero nelle produzioni troviamo sempre una stringa e poi un simbolo VN all'estremo destro (rispettivamente estremo sinistro della frase); quindi avremo una catena aumentante che appende valori sempre in coda (rispettivamente in testa) alla produzione già scritta. Un caso particolare delle grammatiche regolari sono le grammatiche strettamente lineari, ovvero quelle che a sinistra (risp. destra) non hanno una stringa di caratteri, bensì un carattere solo.
Esempio $S \to cA, A \to iB, B \to aC, C \to o$

La differenza sostanziale tra grammatiche di tipo 2 e di tipo 3 (molto simili tra loro) sta nella presenza o meno di self-embedding, un metodo che introduce ricorsione in cui si aggiungono contemporaneamente simboli sia a

sinistra che a destra della produzione (es. le parentesi). Quindi una grammatica context-free che contiene il self-embedding ò sicuramente di tipo 2, ma se non lo contiene, allora ò di tipo 3. Bisogna però fare molta attenzione ai finti self-embedding, ovvero alle regole di self-embedding che, messe in OR assieme ad altre regole, di fatto lo disattivano.

Il fatto che la grammatica di tipo 1 ammetta la stringa vuota, mentre le grammatiche di tipo 2 e di tipo 3 la ammettono può sembrare fuorviante in quanto non ò possibile che un caso particolare di una grammatica senza stringa vuota possa ammetterla. Tuttavia si considera la stringa vuota inserita nelle grammatiche di tipo 2 e 3 come inserita di autorità, e in ogni caso, le produzioni di tipo 2 e 3 possono *sempre* essere riscritte in modo da evitare la stringa vuota ottenendola come regola dello scopo.

1.1.1 Eliminazione delle ε-Rules

Si deve cercare di eliminare la stringa vuota dalla produzione, cercando di portarla sempre nella produzione più alta, se non si riesce ad eliminare, fino ad averla solamente come produzione dello scopo.

1.2 Linguaggi

Un linguaggio può essere generato anche da più grammatiche; non ò detto che una grammatica di tipo 2 generi un linguaggio di tipo 2, ma potrebbe generarne uno più semplice; quindi dicendo che un linguaggio ò di un certo tipo, significa che la grammatica più semplice che lo può generare ò di quel tipo.

Linguaggi generati da grammatiche di tipo 1 o superiori si dicono riconoscibili, ovvero esiste un algoritmo che discrimina se una frase appartiene o meno a tale linguaggio.

1.3 Trasformazioni

Alcune volte, per facilitare la costruzione di riconoscitori, ò utile attuare alcune trasformazioni per rendere le produzioni più adatte allo scopo.

- **Sostituzione**: si espande il simbolo non terminale che compare nella parte destra della produzione (ad esempio per non mostrare il self-embedding)
  Esempio: $S \to Xa, B \to bQ|Sc|d \Longrightarrow S \to Xa, B \to bQ|Xac|d$

- **Fattorizzazione**: si isola il prefisso più lungo comune tra due o più produzioni e si crea una nuova produzione che contenga in OR i suffissi che non sono stati compattati.
  Esempio: $S \rightarrow aSb|aSc \Longrightarrow S \rightarrow aSX, X \rightarrow b|c$

- **Eliminazione della ricorsione sinistra**: questa trasformazione si articola in più fasi:

  - inizialmente si stabilisce una relazione d'ordine tra i simboli non terminali coinvolti nella ricorsione
  - si eliminano i cicli ricorsivi a sinistra
  - si eliminano le ricorsioni dirette.

2 Riconoscitori a Stati Finiti

I linguaggi di tipo 1, 2 o 3 sono riconoscibili (non lo sono quelli di tipo 0), quindi esiste un algoritmo che permette di decidere se una frase appartiene o meno ad un linguaggio. Per le grammatiche di tipo 3 si utilizza un automa a stati finiti, o meglio un riconoscitore a stati finiti, che è una specializzazione di ASF.

Un riconoscitore a stati finiti definisce l'evoluzione dell'automa a partire dallo stato iniziale in corrispondenza di una sequenza d'ingresso. Una frase accettata da un RSF è una frase che porta il riconoscitore dallo stato iniziale allo stato finale; se una frase è accettata da un RSF, allora è riconosciuta come appartenente al linguaggio.

Teoremi:

- Un linguaggio è vuoto se il RSF accetta una stringa di lunghezza minore al numero degli stati dell'automa

- Un linguaggio è infinito se il RSF accetta una stringa di lunghezza superiore al numero degli stati dell'automa, e minore al doppio di tale numero

Un automa riconoscitore può essere usato anche come automa generatore di un determinato linguaggio a partire dalla grammatica. Per arrivare a tali grammatiche partendo dal linguaggio si userà un sistema dipendente dal tipo di grammatica: Se la grammatica è regolare a destra, si avrà un sistema di tipo top-down:

1. si parte dallo scopo della grammatica come stato iniziale

2. per ogni produzione che ha solo simboli terminali, si raggiunge lo stato finale

3. per ogni produzione che ha anche simboli non terminali, si raggiunge un nuovo stato da cui proseguire

Se la grammatica è regolare a sinistra, si avrà un sistema di tipo bottom-up.

1. si parte dallo scopo della grammatica come stato finale

2. per ogni regola che ha solo simboli terminali, si fa partire la freccia dallo stato iniziale I verso lo stato analizzato

3. per ogni regola che ha un simbolo non terminale, si fa partire la freccia da uno stato precedente equivalente al simbolo non terminale.

Quindi, dato un automa riconoscitore,si possono trarre le relative grammatiche a seconda del tipo di analisi (top down/bottom up) che ne viene fatta.

2.1 Implementazione dei riconoscitori

Un RSF deterministico è facilmente realizzabile con un linguaggio imperativo.

- while + if annidati: riconoscitore cablato nel codice, estensibilità scarsa

- while + switch: idem

- while + tabella separata: automa non cablato nel codice, più estensibile.

Certe grammatiche però possono portare ad un automa non deterministico (= con la stessa iniziale ho più alternative). Se il linguaggio supporta il non-determinismo, fare il riconoscitore è molto semplice: ogni produzione diventa una regola (es. Prolog). Se il linguaggio non lo permette (es. linguaggi imperativi) il riconoscitore non deterministico è inefficiente: è necessario quindi ricondurre l'automa non deterministico ad un automa deterministico equivalente. L'automa deve essere tradotto in grammatica mediante tabella delle transizioni e tabella triangolare per rendere l'automa minimo, poi può essere implementato con un linguaggio dichiarativo o imperativo.

2.2 Espressioni regolari

L'insieme dei linguaggi riconosciuti da un ASF coincide con l'insieme di linguaggi regolari (cioè quelli descritti da espressioni regolari).
E' possibile passare dalla grammatica all'espressione regolare interpretando le produzioni come equazioni:

- i simboli terminali sono i simboli noti
- i simboli non terminali sono le incognite
- le alternative sono delle somme

2.2.1 Algoritmo per grammatiche lineari a destra

- Si riscrivono le produzioni della grammatica cambiando le alternative con le somme
- Si fa un raccoglimento dei simboli non terminali (che sono a destra) comuni
- Si eliminano le ricorsioni dirette ($X = uX \cup \delta \Rightarrow X = (u)^*\delta$)
- Si applicano le regole precedenti fino ad ottenere un sistema omogeneo

2.2.2 Algoritmo per grammatiche lineari a sinistra

- Si riscrivono le produzioni della grammatica cambiando le alternative con le somme
- Si fa un raccoglimento dei simboli non terminali (che sono a sinistra) comuni
- Si eliminano le ricorsioni dirette ($X = Xu \cup \delta \Rightarrow X = \delta(u)^*$)
- Si applicano le regole precedenti fino ad ottenere un sistema omogeneo

3 Riconoscitori per grammatiche Context-Free

4 Interpretazione

5 JavaScript

Javascript è un linguaggio di scripting interpretato.

Le variabili sono loosely typed, ovvero posso assegnare valori di tipo diverso alla stessa variabile. Lo scope è globale per le variabili definire fuori da funzioni e per quelle definite implicitamente dentro le funzioni, mentre è locale per le variabili definite con la parola chiave var.

Le procedure/funzioni sono introdotte dalla keyword *function(param1, ...)* e i parametri sono definiti senza dichiarazione di tipo. Le chiamate sono fatte fornendo la lista dei parametri: se ne vengono presentati troppi, gli eccedenti sono scartati, se ne vengono presentati pochi invece gli ultimi sono undefined. In Javascript le variabili possono riferirsi a funzioni; è quindi possibile passare funzioni come parametro di altre funzioni.

Un oggetto è una collezione di dati dotata di nome e per accedere alle proprietà si usa la dotNotation. Ogni oggetto è costruito mediante l'operatore new da un costruttore che stabilisce la struttura dell'oggetto (e le sue proprietà). Siccome non esistono classi, il nome del costruttore è scelto dall'utente (l'argomento di new non è il nome di una classe, ma il nome della funzione-costruttore). Ogni oggetto tiene traccia del proprio costruttore mediante la proprietà constructor.

E' possibile eliminare le proprietà mediante l'operatore delete, mentre per aggiungerne è sufficiente nominarle e usarle. Per aggiungere lo stesso metodo a più oggetti si può assegnare lo stesso metodo ai diversi oggetti (`p1.getX = p2.getX`) oppure basandosi sull'idea di prototipo.

Ogni funzione Javascript è un oggetto costruito dal costrutto function (`square=function(x){return x*x}`)o dal costruttore Function (`square = new Function(x,return x*x)`).

Ogni oggetto ha sempre un prototipo che ne specifica le proprietà iniziali. L'oggetto prototipo è referenziato dalla proprietà __proto__, presente internamente ad ogni oggetto. I costruttori inoltre hanno un prototipo di costruzione che specifica il prototipo da attribuire agli oggetti costruiti con tale costruttore, ed è specificato dalla proprietà prototype, presente in ogni costruttore. Le funzioni sono costruite dal costruttore predefinito Function; il prototipo è l'oggetto referenziato da Function.prototype. Gli oggetti sono definiti dall'utente mediante costruttori specifici; il loro prototipo è l'oggetto Object. Object.prototype è Object stesso. Perché tra i nuovi oggetti si possa instaurare una relazione personalizzata si può associare un apposito prototipo diverso da Object: gli oggetti creati avranno dall'inizio tutte le proprietà del loro prototipo __proto__.

- Prima si crea l'oggetto che farà da prototipo

- Poi lo si assegna alla proprietà prototype del costruttore

Gli oggetti creati prima di questa azione rimangono inalterati.

6 Reti di Petri

Una rete di Petri è una delle varie rappresentazioni matematiche di un sistema distribuito discreto.

Una rete di Petri Place/Transition consiste di posti, transizioni e archi diretti. I posti possono contenere un certo numero di token. Una distribuzione di token sull'insieme dei posti della rete è detta marcatura. Le transizioni agiscono sui token in ingresso secondo una regola, detta regola di scatto (in inglese firing). Una transizione è abilitata se può scattare, cioè se ci sono token in ogni posto di input. Quando una transizione scatta, essa consuma i token dai suoi posti di input, esegue dei task e posiziona un numero specificato di token in ognuno dei suoi posti di uscita. L'esecuzione delle reti di Petri è non deterministica. Ciò significa due cose:

1. se più transizioni sono abilitate nello stesso momento, una qualsiasi di esse può scattare

2. non è garantito che una transizione abilitata scatti; una transizione abilitata può scattare immediatamente, dopo un tempo di attesa qualsiasi (a patto che resti abilitata), o non scattare affatto.

Una rete di Petri può essere rappresentata a matrice: in questo modo un calcolatore comprende meglio struttura e funzionamento della rete. Una rete è pura se non esistono posti che sono contemporaneamente input e output per la stessa transazione.

Un ASF può essere descritto da una rete di Petri con le seguenti caratteristiche:

- ogni transazione ha un solo posto d'ingesso e un solo posto d'uscita

- tutti gli archi hanno peso unitario

- tutte le marcature prevedono un unico token, la cui posizione determina lo stato

6.1 Proprietà delle reti di Petri

- raggiungibilità : una marcatura M' è raggiungibile da M se esiste una sequenza finita di scatti che porta M a M'. L'insieme delle marcature

raggiungibili si chiama insieme di raggiungibilità ed è rappresentato da un grafo di raggiungibilità . Tale grafo è utile per determinare stati in/desiderati raggiungibili (es. deadlock)

- limitatezza: se per ogni marcatura dell'insieme di raggiungibilità il token in un posto è limitato, il posto è detto k-limitato. La rete è k-limitata se tutti i suoi posti sono k-limitati.

- vitalità : se dalla marcatura raggiunta è possibile far scattare qualunque transazione della rete. Ci sono 5 livelli di vitalità , dalla transizione morta a quella viva, in cui esiste sempre una sequenza di scatti che abilita la transazione.

Queste proprietà sono decidibili ma è indecidibile definire se l'insieme di raggiungibilità di una rete include quello di un'altra.

7 Riconoscitori LR(0)

Le grammatiche LR possono essere analizzate left-to-right ma utilizzando una Right-most-derivation. E' un'analisi meno naturale della LL, ma tuttavia più potente, ed essendo spesso complessa si adottano versioni semplificate di parsing LR quali Simple LR (SLR) oppure Look-Ahead LR (LALR).

La tecnica dell'analisi LR opera bottom-up, ovvero partendo dalla frase da riconoscere cerca di ricondursi alle produzioni della grammatica fino a risalire allo scopo. Per fare questo è necessario calcolare i contesti LR(k) della grammatica, verificare che non ci siano collisioni tra contesti di produzioni diversi e quindi utilizzare quel contesto per guidare l'analisi. Se non è necessario guardare avanti, la grammatica è LR(0), in caso contrario sarà LR(1), ...

Per definizione lo scopo delle grammatiche LR non deve mai comparire nella parte destra delle produzioni, pertanto se compare anche nella parte destra di almeno una produzione, si crea un nuovo scopo $Z \rightarrow S$ che quindi non compaia mai nella parte destra.

7.1 Calcolo dei contesti LR di una grammatica

Il contesto di una produzione è l'insieme di tutti i prefissi di una forma di frase che contenga all'ultimo passo di una derivazione canonica destra la produzione di cui si vuole calcolare il contesto. Quindi, il contesto di $A \rightarrow a$ sarà l'unione di tanti contesti sinistri che in generale precedono a e appunto

il simbolo terminale a. $LR(0)ctx(A \rightarrow a) = leftctx(A) * \{a\}$. Il contesto sinistro di un metasimbolo viene calcolato scorrendo tutte le produzioni della grammatica e ricordando che:

1. $leftctx(Z) = \{\varepsilon\}$
2. se io ho $B \rightarrow aAb$ allora $leftctx(A) \subseteq leftctx(B) * a$ (minore o uguale perché potrebbero esistere altre produzioni che contengono il metasimbolo A)
3. le produzioni che nella parte destra contengono solamente simboli terminali non sono considerate

Il contesto sinistro di ogni metasimbolo è poi ottenuto unendo tutti i sottoinsiemi delle diverse produzioni che contengono tale metasimbolo nella parte destra della produzione.
Esempio: $leftctx(Z) = \{\varepsilon\}$ $leftctx(S) = leftctx(Z)\{\varepsilon\} U leftctx(S)\{a\}$ $leftctx(A) = leftctx(S)\{aS\}$
Si calcola poi la grammatica regolare ausiliaria ponendo tutte quelle che prima erano unioni come degli OR (quindi separati da $+$) e, risolvendo le equazioni, fino ad ottenere le espressioni regolari dei vari contesti.

NB: nella riduzione ai contesti sinistri, si devono sostituire solamente i leftctx(), se conosciuti, per esempio dato $leftctx(S) = a^*$ allora $leftctx(A) = leftctx(S)aS$ diventa $leftctx(A) = a^*aS$.
Dopo aver calcolato i contesti sinistri si provvede a unirli alle singole produzioni della grammatica, quindi per ogni produzione si concatenerà il contesto sinistro del metasimbolo posto a sinistra con la produzione che si trova a destra.
Nell'esempio precedente $LR(0)ctx(S \rightarrow aSAB) = a^*aSAB$. Da qui è possibile costruire un ASF che riconosca l'unione di questi linguaggi e alterni le operazioni di shift (si usa un simbolo della forma di frase corrente per cambiare lo stato dell'automa) e reduce (passo di riduzione nel momento in cui si giunge ad uno stato finale) un per effettuare il parsing della frase. Ogni volta che si esegue uno shift si riparte dall'inizio con la nuova frase ottenuta <= si potrebbe ottimizzare mantenendo uno stack degli stati in cui impilare i vari stati attraversati e disimpilarne tanti quanti i sinboli coinvolti nelle riduzioni.

Una grammatica è analizzabile con l'analisi LR(0) se ogni stato di riduzione dell'ASF è etichettato da una produzione unica e non ha archi di uscita etichettati da simboli terminali.

E' possibile calcolare l'automa caratteristico in modo più meccanico, dato che il metodo presentato sopra è molto lungo e complesso. Per convenzione si

ricordi che ogni frase lecita è esplicitamente terminata dal simbolo \$ (quindi la prima produzione sarà $Z \rightarrow S\$$) e che il confine tra una parte già analizzata di una forma di frase e una non ancora analizzata è segnalato da un punto .

1. Inizialmente ho $Z \rightarrow .S\$$

2. Dato che a destra del cursore c'è solo S, per descrivere al meglio questo stato devo aggiungere tutte le produzioni di S inserendo il cursore all'inizio delle parti destre delle produzioni che aggiungo. Se in queste produzioni i primi simboli sono dei metasimboli, aggiungerà ancora le produzioni relative, e il cursore sarà anche qui a inizio della parte destra della produzione. Questo è il primo stato dell'automa caratteristico.

3. Ora bisogna investigare su tutte le possibile evoluzioni dello stato precedente, ovvero spostare verso destra il cursore di una posizione in tutti i modi possibili.

   - Se arrivo col cursore subito prima di \$, ho raggiunto uno stato finale di accettazione F
   - Se arrivo col cursore a leggere la fine di una produzione, ho raggiunto uno stato di riduzione Ri
   - Se mi ritrovo prima di un metasimbolo che ho già espanso, devo ri-aggiungere tutte le produzioni espandibili da quel metasimbolo, ponendo ancora il cursore all'inizio della parte destra della produzione

   Qui ogni arco corrisponde ad uno shift, e ogni nodo terminale corrisponde ad un reduce.

Per far meglio capire a una macchina come comportarsi, si costruisce una tabella di parsing scrivendo informazioni articolate spiegando se shiftare /consumare /ridurre /accettare l'input o la produzione e spostarsi in un determinato stato.

7.2 Analisi LR(1)

Le grammatiche LR(1) richiedono semplicemente i contesti LR(0) e il simbolo successivo per risolvere i conflitti. Generalmente l'analisi LR(k) è analoga a quella LR(0) ma tutte le riduzioni sono ritardate di k simboli.

Per calcolare l'automa caratteristico si procede come nel caso LR(0), ovvero si calcolano le espressioni regolari per i contesti LR(k) e le si usano per calcolare l'automa.

Calcolo dei contesti LR(1):

1. $leftctx(Z, \varepsilon) = \{\varepsilon\}$

2. Per gli altri casi di simboli non terminali, data la produzione $P \to \gamma Q \delta$ e considerando v come i possibili simboli terminali che possono seguire P e $u = head(L(\delta) \bullet v)$ con $L(\delta)$ il linguaggio generato da δ, si ha che $leftctx(Q, u) \supseteq leftctx(P, v) \bullet \{\gamma\}$

Esempio, data la grammatica

$$Z \to S\$ \quad S \to CcBA \quad A \to ab|Aab$$
$$B \to C|Db \qquad C \to a \qquad D \to a$$

Si avrà:
$leftctx(Z, \varepsilon) = \{\varepsilon\}$
$leftctx(S, \varepsilon) \supseteq leftctx(Z, \varepsilon) \bullet \{\varepsilon\}$
$leftctx(C, b) \supseteq leftctx(S, \varepsilon) \bullet \{\varepsilon\}$
$leftctx(B, a) \supseteq leftctx(S, \varepsilon) \bullet \{Cb\}$
$leftctx(A, \$) \supseteq leftctx(S, \varepsilon) \bullet \{CbB\}$
$leftctx(A, a) \supseteq leftctx(A, \$) \bullet \{CbB\}$
...