Riconoscimento e parsing di linguaggi context–free

27 aprile 2004

1 Algoritmi di riconoscimento e parsing di linguaggi context–free

Studieremo qui i problemi del riconoscimento e del parsing per i linguaggi context–free. Dato un linguaggio context–free L, il problema del riconoscimento per L consiste nel rispondere alla domanda "$x \in L$?" dove x è una stringa in input. Nel caso del parsing, data una grammatica context–free che genera L, si richiede una risposta al problema del riconoscimento e, in caso di risposta affermativa, un "testimone" di tale risposta, cioè un albero di derivazione della stringa x o, equivalentemente, una sua derivazione leftmost nella grammatica G (si noti che nel caso di grammatiche ambigue il problema può essere formulato chiedendo anche *tutti* gli alberi di derivazione della stringa).

È noto che la classe dei linguaggi context–free coincide con la classe dei linguaggi accettati da automi a pila. Pertanto un primo algoritmo di riconoscimento potrebbe basarsi sulla simulazione di automi a pila. Tuttavia, il modello che occorre considerare è non deterministico. Infatti, la classe dei linguaggi accettati da automi a pila deterministici, detta anche classe dei *linguaggi context–free deterministici*, è contenuta *strettamente* in quella dei linguaggi accettati da automi a pila non deterministici. La simulazione diretta di un automa a pila non deterministico può essere estremamente inefficiente. Consideriamo ad esempio l'automa che riconosce il linguaggio formato dalle parole palindrome: ad ogni lettura di un simbolo in input, è necessario effettuare una scelta non deterministica. D'altra parte, è facile scrivere un programma che riconosce le stringhe di tale linguaggio, con una semplice analisi dell'input in tempo lineare.

Sono stati proposti in letteratura diversi algoritmi per il riconoscimento dei linguaggi context–free. Indichiamo brevemente i principali qui di seguito:

- Algoritmo di CYK [Yo67]
  Questo algoritmo è stato scoperto nella seconda metà degli anni sessanta, indipendentemente da Cocke, Younger e Kasami. L'algoritmo richiede che la grammatica del linguaggio sia *in Forma Normale di Chomsky* ed è basato su tecniche di programmazione dinamica. Il tempo

di lavoro dell'algoritmo è $O(n^3)$ dove n è la lunghezza della stringa in input. Mediante la struttura dati costruita dall'algoritmo (una tabella), una volta effettuato il riconoscimento è possibile, sempre negli stessi limiti di complessità in tempo, effettuare anche il parsing.

L'algoritmo è piuttosto semplice. Tuttavia la restizione di dovere lavorare con grammatiche in forma normale di Chomsky lo rende poco utile alle applicazioni nell'ambito dei compilatori. Ogni grammatica context–free può essere infatti trasformata in forma normale di Chomsky. Tuttavia, gli alberi di parsing della grammatica ottenuta riflettono poco la struttura iniziale delle produzioni e, dunque, mal si prestano alle fasi successive del processo di compilazione.

- Algoritmo di Earley [Ea70]
  Proposto da Earley nel 1970, questo algoritmo è in grado di trattare *qualsiasi* grammatica context–free, superando quindi lo svantaggio principale dell'algoritmo precedente. Anche in questo caso il tempo di lavoro è $O(n^3)$ dove n è la lunghezza della stringa in input e, anche in questo caso, nello stesso tempo è possibile effettuare anche il parsing. Presenteremo in dettaglio l'algoritmo più sotto.

- Algoritmo di Valiant [Va75]
  Questo algoritmo, presentato nel 1975, è di interesse principalmente teorico, in quanto mostra che la complessità del problema del riconoscimento per i linguaggi context–free non è a $O(n^3)$, come i due algoritmi precedenti potrebbero fare supporre, ma è inferiore. In particolare l'algoritmo consiste di una ingegnosa e veloce riduzione del problema del riconoscimento al problema del calcolo del prodotto di matrici booleane che può essere risolto con l'algoritmo di Strassen in tempo $O(n^{2.81})$ (e in tempo inferiore con algoritmi per il prodotto di matrici che hanno migliorato quello di Strassen). Pertanto, essendo il costo della riduzione trascurabile, si ottiene un tempo subcubico anche per il problema del riconoscimento.

  Si noti che è tuttora aperto il problema dell'esistenza di algoritmi quadratici di riconoscimento per i linguaggi context–free.[1]

2 L'algoritmo di Earley

Presentiamo ora più in dettaglio l'algoritmo di riconoscimento e di parsing proposto da Earley. Analizzeremo prima gli aspetti relativi al riconoscimento, indicando poi come effettuare, con alcune semplici modifiche, anche il parsing.

Nel seguito denoteremo una grammatica come una quadrupla $G = (V, \Sigma, P, S)$ dove V è l'insieme delle *variabili* o *simboli non terminali*, Σ è l'insieme del *simboli terminali*, P è l'insieme delle *produzioni* e $S \in V$ è il *simbolo iniziale*. Indicheremo con $L(G)$ o, più brevemente, con L, il linguaggio generato dalla grammatica.

Esempio di riferimento
Nel seguito, per gli esempi che presenteremo ci riferiremo alla seguente grammatica $G = (V, \Sigma, P, S)$:

- $V = \{E\}$
- $\Sigma = \{a, +, *, (,)\}$
- Le produzioni sono:
  $E \rightarrow E + E$
  $E \rightarrow E * E$
  $E \rightarrow (E)$
  $E \rightarrow a$

[1] Per particolari sottoclassi dei linguaggi context–free si conoscono algoritmi migliori. Ad esempio, lo stesso algoritmo di Earley, nel caso di grammatiche non ambigue lavora in tempo quadratico.

- Il simbolo iniziale è E.

È facile vedere che il linguaggio generato dalla grammatica è costituito da semplici espressioni aritmetiche in cui a indica gli operandi.

Le operazioni base dell'algoritmo

Una grammatica descrive un linguaggio in maniera "generativa", fornendo cioè uno strumento per generare Il meccanismo di generazione, partendo dal simbolo iniziale della grammatica, effettua sostituzioni successive, in base alle regole di produzione, sino a generare una stringa composta da soli simboli terminali. In particolare, nel caso delle grammatiche libere dal contesto, ogni sostituzione riscrive una variabile A della forma sentenziale considerata, con una stringa α (eventualmente vuota) di terminali e non terminali, applicando cioè la produzione $A \to \alpha$.

Come noto, senza perdita di generalità possiamo limitarci a considerare *derivazioni leftmost* (più a sinistra) cioè derivazioni nelle quali ad ogni passo si applica una sostituzione al simbolo più a sinistra della forma sentenziale considerata. Questo significa che se la forma sentenziale ottenuta è del tipo $wA\beta$ con $w \in \Sigma^*$, $A \in V$ e $\beta \in (V \cup \Sigma)^*$, la prossima sostituzione dovrà essere applicata alla variabile A.

Il procedimento di generazione è (salvo casi banali) non deterministico, in quanto ad ogni passo sono possibili più scelte: vi potrebbero essere più produzioni con A sulla sinistra, pertanto da $wA\beta$ si potrebbero ottenere in un passo differenti forme sentenziali. Volendo generare tutte le stringhe del linguaggio occorrerebbe esaminare tutte le possibili derivazioni leftmost (che possono essere infinite).

Consideriamo la grammatica dell'esempio di riferimento. Partendo dal simbolo iniziale E sono possibili 4 sostituzioni, corrispondenti alle 4 produzioni della grammatica. Pertanto, dopo il primo passo di derivazione, vi sono 4 forme sentenziali possibili, e cioè $E + E$, $E * E$, (E), ed a. Se vogliamo utilizzare il procedimento di derivazione, per verificare se una data stringa x in input appartiene al linguaggio, sulla base del primo simbolo di x possiamo escludere alcune di queste forme sentenziali. Ad esempio, se il primo simbolo in input è a, la forma (E) può essere scartata.

L'idea dell'algoritmo di Earley è quella di costruire progressivamente tutte le possibili derivazioni leftmost "compatibili" con la stringa x in input. Le derivazioni non vengono rappresentate direttamente, ma mediante un sistema di puntatori. Durante il procedimento si analizza la stringa di input da sinistra verso destra scartando via via le derivazioni in cui non vi sia corrispondenza tra i terminali derivati e i corrispondenti simboli nella stringa in input.

Data in input una stringa $x = \sigma_1\sigma_2 \ldots \sigma_n$, con $\sigma_j \in \Sigma$, l'algoritmo esamina x da sinistra verso destra costruendo, in corrispondenza di ogni simbolo σ_j un insieme di "stati" S_j, dove uno *stato* è definito da:

- una *dotted rule*, cioè una produzione della grammatica sul cui lato destro sia stato inserito un punto, per marcarne una posizione,
- un *puntatore*, cioè un intero i utilizzato per indicare la posizione dell'input dopo la quale è iniziato l'esame della produzione contenuta nella dotted rule.

In altre parole, uno stato ha la forma $(A \to \alpha \cdot \beta, i)$, dove $A \to \alpha\beta$ è una produzione della grammatica e i è un intero. In particolare se lo stato $(A \to \alpha \cdot \beta, i)$ appartiene all'insieme S_j, l'intero i deve soddisfare $0 \leq i \leq j$. Ciò, per come è costruito l'algoritmo, significa che:

- si è iniziato l'esame della produzione $A \to \alpha\beta$ a partire dalla posizione $i + 1$ dell'input,
- si è esaminata la parte che precede il punto, cioè α,
- si è verificato che α genera il fattore $\sigma_{i+1} \ldots \sigma_j$ di x.

L'algoritmo di Earley esamina via via gli stati e in base alla loro struttura sceglie una tra le tre operazioni *scanner*, *predictor* e *completer* che ora descriviamo. In particolare, supponiamo che l'algoritmo debba esaminare uno stato $(A \to \alpha \cdot \beta, i) \in S_j$:

- **Scanner**
  Se β *inizia con un terminale* a, cioè, $\beta = a\beta'$, con $a \in \Sigma$, $\beta' \in (V \cup \Sigma)^+$:

  $$\texttt{se } a = \sigma_{j+1} \texttt{ allora aggiungi } (A \to \alpha a \cdot \beta', i) \texttt{ a } S_{j+1}$$

  Questa operazione verifica la corrispondenza tra il successivo simbolo nella produzione (cioè quello a destra del punto) e il successivo simbolo in input. In caso affermativo la parte di input riconosciuta può essere aumentata, spostando il punto a destra di una prosizione.

- **Predictor**
  Se β *inizia con una variabile* B, cioè, $\beta = B\beta'$, con $B \in V$, $\beta' \in (V \cup \Sigma)^+$:

  $$\texttt{per ogni produzione } B \to \gamma \texttt{ aggiungi } (B \to \cdot\gamma, j) \texttt{ a } S_j$$

  Questa operazione "predice" come verrà espansa la variabile B: le predizioni vengono fatte espandendo B con tutti i lati destri delle relative produzioni (l'indice j nel nuovo stato tiene traccia di dove è stata effettuata la predizione).

- **Completer**
  Se β *è la parola vuota:*

  $$\texttt{per ogni stato } (C \to \eta \cdot A\delta, h) \in S_i \texttt{ aggiungi } (C \to \eta A \cdot \delta, h) \texttt{ a } S_j$$

  Questa operazione "completa" l'operazione di "predictor": si individuano nell'insieme S_i tutti gli stati che avevano predetto la produzione $A \to \alpha$. Poiché in S_j la predizione si è verificata corretta, si può "allungare" la parte riconosciuta. In altre parole, $(C \to \eta \cdot A\delta, h) \in S_i$ implica che $\eta \stackrel{*}{\Rightarrow} \sigma_{h+1} \dots \sigma_i$ e $(A \to \alpha\cdot, i) \in S_j$ implica che $A \Rightarrow \alpha \stackrel{*}{\Rightarrow} \sigma_{i+1} \dots \sigma_j$. Da ciò segue che $\eta A \stackrel{*}{\Rightarrow} \sigma_{h+1} \dots \sigma_i$; di conseguenza, si aggiunge all'insieme S_j lo stato $(C \to \eta A \cdot \delta, h)$.

L'algoritmo

Per semplicità, conviene aggiungere un nuovo terminale, indicato con \$, utilizzato per delimitare la fine dell'input, e la produzione $\phi \to S\$$, dove ϕ è una nuova variabile e S il simbolo iniziale della grammatica G considerata.

L'algoritmo, costruisce ed elabora, uno dopo l'altro, gli insiemi S_j di stati:

```
input x = σ_1 ... σ_n
σ_{n+1} := $
S_0 := {(φ → ·S$, 0)}
for j := 0 to n + 1 do
  elabora S_j uno stato alla volta, applicando
  una delle 3 operazioni scanner, predictor, completer
if S_{n+1} = {(φ → S$·, 0)}
  then accetta
  else rifiuta
```

Esempio
Mostriamo ora in dettaglio i principali passi dell'esecuzione dell'algoritmo di Earley su una stringa x molto semplice e cioè la stringa (a).

Abbiamo $n = 3$ e, dopo l'aggiunta del marcatore di fine stringa, $\sigma_1 = (, \sigma_2 = a, \sigma_3 =)$ e $\sigma_4 = \$$.

Verranno dunque costruiti ed elaborati 5 insiemi di stati $S_0, S_1, \ldots, S_4$. Inizialmente tali insiemi sono vuoti, con l'eccezione di S_0 che contiene, come unico stato, $(\phi \to \cdot E\$, 0)$.

La fase di elaborazione, esamina ogni insieme di stati, partendo da S_0 (secondo ciclo for).

- Esame dell'insieme S_0.
  Come abbiamo detto, all'inizio questo insieme contiene solo $(\phi \to \cdot S\$, 0)$, pertanto si esaminerà questo stato:

  - Esame dello stato $(\phi \to \cdot E\$, 0)$.
    Il primo simbolo a destra del punto è la variabile E. Verrà dunque effettuata l'operazione predictor, con la quale si aggiungono all'insieme S_0 nuovi stati corrispondenti a tutte le produzioni della grammatica con la variabile E sul lato sinistro. Il punto viene collocato all'inizio del lato destro per indicare che l'esame del lato destro non è ancora iniziato, il puntatore viene inizializzato a 0, per indicare la posizione in cui è stata fatta la predizione, cioè quella attualmente considerata. Dopo queste aggiunte l'insieme S_0 sarà dunque costituito dagli stati:

    $$(\phi \to \cdot S\$, 0)$$
    $$(E \to \cdot E + E, 0)$$
    $$(E \to \cdot E * E, 0)$$
    $$(E \to \cdot (E), 0)$$
    $$(E \to \cdot a, 0)$$

    A questo punto l'elaborazione prosegue con l'elemento successivo presente ora in S_0.
  - Esame dello stato $(E \to \cdot E + E, 0)$.
    Come nel caso precedente, il simbolo a destra del punto è la variabile E. Dunque, verrà effettuata l'operazione di predictor che, in questo caso, non aggiunge nulla di nuovo all'insieme S_0, poiché esso contiene già tutti gli stati che l'operazione potrebbe aggiungere.
  - Esame dello stato $(E \to \cdot E * E, 0)$.
    Analogamente al punto precedente, gli insiemi di stati non vengono modificati.
  - Esame dello stato $(E \to \cdot (E), 0)$
    In questo caso, il simbolo a destra del punto è il terminale (. Viene pertanto effettuata l'operazione chiamata scanner, con la quale tale terminale viene confrontato con il successivo in input, cioè σ_1. Poiché i due caratteri coincidono, l'effetto è quello di aggiungere all'insieme S_1 (attualmente vuoto) lo stato ottenuto da quello in esame, spostando il punto a destra del terminale, e cioè lo stato $(E \to (\cdot E), 0)$.
  - Esame dello stato $(E \to \cdot a, 0)$.
    Anche in questo caso, essendo il simbolo a destra del punto un terminale, si effettua l'operazione di scanner. Poichè il terminale a è differente dal simbolo di input in posizione 1, l'operazione non aggiunge alcuno stato agli insiemi.

  A questo punto tutto gli stati dell'insieme S_0 sono stati esaminati. Si passa dunque all'insieme di stati successivo.

- Esame dell'insieme S_1.
  L'insieme contiene solo lo stato $(E \to (\cdot E), 0)$, inserito in uno dei passi precedenti.

- Esame dello stato $(E \to (\cdot E), 0)$.
  Predictor: vengono aggiunti a S_1 i seguenti stati:

$$(E \to \cdot E + E, 1)$$
$$(E \to \cdot E * E, 1)$$
$$(E \to \cdot (E), 1)$$
$$(E \to \cdot a, 1)$$

- Esame dello stato $(E \to \cdot E + E, 1)$.
  Esame dello stato $(E \to \cdot E * E, 1)$.
  Predictor: in entrambi i casi non vi sono aggiunte di nuovi stati a S_1.
- Esame dello stato $(E \to \cdot (E), 1)$.
  Scanner: il terminale a destra del punto, cioè la parentesi aperta è diverso dal successivo in input, cioè il carattere a. L'operazione pertanto non modifica gli insiemi di stati.
- Esame dello stato $(E \to \cdot a, 1)$.
  Scanner: il terminale a destra del punto coincide con il carattere nella posizione successiva dell'input. Pertanto si aggiunge all'insieme S_2 lo stato $(E \to a\cdot, 1)$.

A questo punto non vi sono ulteriori stati da esaminare in S_1. Per comodità se ne riporta il contenuto:

$$(E \to (\cdot E), 0)$$
$$(E \to \cdot E + E, 1)$$
$$(E \to \cdot E * E, 1)$$
$$(E \to \cdot (E), 1)$$
$$(E \to \cdot a, 1)$$

- Esame dell'insieme S_2.
  L'insieme contiene solo lo stato $(E \to a\cdot, 1)$, inserito all'ultimo dei passi precedenti.

  - Esame dello stato $(E \to a\cdot, 1)$.
    Non vi sono simboli a destra del punto. Pertanto si effettua l'operazione completer: essa stabilisce che vadano aggiunti nuovi stati a S_2 (l'insieme in esame); tali stati si ottengono considerando gli stati di S_1 (l'insieme indicato dal puntatore presente nello stato) in cui E appare a destra del punto, e spostando il punto a destra di E. Ad esempio, dal primo stato $(E \to (\cdot E), 0)$ di S_1, si ottiene il nuovo stato di S_2 $(E \to (E\cdot), 0)$. Analogamente, si aggiungono i due stati $(E \to E \cdot +E, 1)$ e $(E \to E \cdot *E, 1)$.
  - Esame dello stato $(E \to (E\cdot), 0)$.
    Scanner: il simbolo a destra del punto coincide con la posizione successiva in input. Pertanto all'insieme S_3, ora vuoto, viene aggiunto lo stato $(E \to (E)\cdot, 0)$.
  - Esame dello stato $(E \to E \cdot +E, 1)$
    Esame dello stato $(E \to E \cdot *E, 1)$
    Scanner: il simbolo a destra del punto è diverso dal simbolo presente nella posizione successiva dell'input. Pertanto non si aggiungono nuovi stati.

  Termina così l'esame di S_2. Gli stati presenti sono:

$$(E \to \cdot a, 1)$$

$$(E \to (E\cdot), 0)$$
$$(E \to E \cdot + E, 1)$$
$$(E \to E \cdot * E, 1)$$

- Esame dell'insieme S_3.
  L'insieme contiene solo lo stato $(E \to (E)\cdot, 0)$.
  - Esame dello stato $(E \to (E)\cdot, 0)$.
    Completer: vengono aggiunti all'insieme gli stati $(E \to E \cdot \$, 0)$, $(E \to E \cdot + E, 0)$ e $(E \to E \cdot * E, 0)$.
  - Esame dello stato $(\phi \to E \cdot \$, 0)$.
    Scanner: il terminale a destra del punto coincide con il simbolo in posizione 4 dell'input (marcatore di fine input). Pertanto si aggiunge a S_4 lo stato $(\phi \to E\$\cdot, 0)$.
  - Esame dello stato $(E \to E \cdot + E, 0)$.
    Esame dello stato $(E \to E \cdot * E, 0)$.
    Scanner: il simbolo a destra del punto è diverso dal simbolo in posizione 4 dell'input. Pertanto non si aggiunge alcuno stato.

  Alla fine dell'esame di S_3, gli stati presenti sono:

$$(E \to (E)\cdot, 0)$$
$$(\phi \to E \cdot \$, 0)$$
$$(E \to E + \cdot E, 0)$$
$$(E \to E * \cdot E, 0)$$

A questo punto sono stati esaminati tutti gli insiemi di stati da S_0 a S_3. Non resta che stabilire l'appartenenza della stringa x di input al linguaggio, controllando il contenuto di S_4. Poiché $S_4 = \{(\phi \to E\$\cdot, 0)\}$ la risposta è positiva.

Come ottenere un parser

Al fine di ottenere, al termine del processo di riconoscimento, un albero di derivazione della stringa in input, è utile memorizzare alcune informazioni aggiuntive, oltre agli insiemi degli stati.

In particolare, effettuando l'operazione di completer relativamente a uno stato $(A \to \alpha\cdot, i) \in S_j$, si memorizza un puntatore da ogni stato $(C \to \eta A \cdot \delta, h)$ che viene aggiunto a S_j allo stato $(A \to \alpha\cdot, i) \in S_j$. L'esame degli insiemi di stati (dall'ultimo verso il primo) e l'uso di questi puntatori, permette di ricostruire la derivazione della stringa.

Consideriamo la stringa $x = (a)$ dell'esempio precedente. Durante il riconoscimento verranno aggiunti i seguenti puntatori:

- nell'insieme S_3: da tutti gli altri stati verso lo stato $(E \to (E)\cdot, 0)$,
- nell'insieme S_2: da tutti gli altri stati verso lo stato $(E \to a\cdot, 1)$.

Partiamo dall'insieme S_4 in cui troviamo lo stato $(\phi \to E\$\cdot, 0)\}$. Muovendoci a ritroso, ritroviamo in S_3 lo stato $(\phi \to E \cdot \$, 0)\}$ da cui esso è stato generato mediante un'operazione di scanner. Seguiamo quindi il puntatore che ci porta allo stato $(E \to (E)\cdot, 0)$ di S_3. Ritroviamo in S_2 lo stato $(E \to (E\cdot), 0)$ da cui esso proviene, seguendo il puntatore raggiungiamo lo stato $(E \to a\cdot, 1)$ di S_2. Tralasciando la produzione aggiunta inizialemte, abbiamo dunque la seguente derivazione $E \Rightarrow (E) \Rightarrow (a)$.

Per motivi di brevità, l'esempio presentato è estremamente semplice. È opportuno simulare l'algoritmo su esempi più articolati, sia di stringhe generate dalla grammatica, che di stringhe non

generate. In particolare, si suggerisce di effettuare la simulazione con una stringa come $a + a * a$, per la quale esistono due alberi di derivazione. Con il procedimento indicato, è possibile ricavarli entrambi.

Riferimenti bibliografici

[Ea70] J. Earley. An efficient conext–free parsing algorithm. *Comm. ACM* **13** (1970), 94–102.

[Va75] L.G. Valiant. General conext–free recognition in less than cubic time. *J. Comp. Syst. Sciences* **10** (1975), 308–315.

[Yo67] D.H. Younger. Recognition and parsing of context–free languages in time n^3. *Information and Computation* **10**(1967), 189–208.