Riconoscere i linguaggi context-free: automi a pila

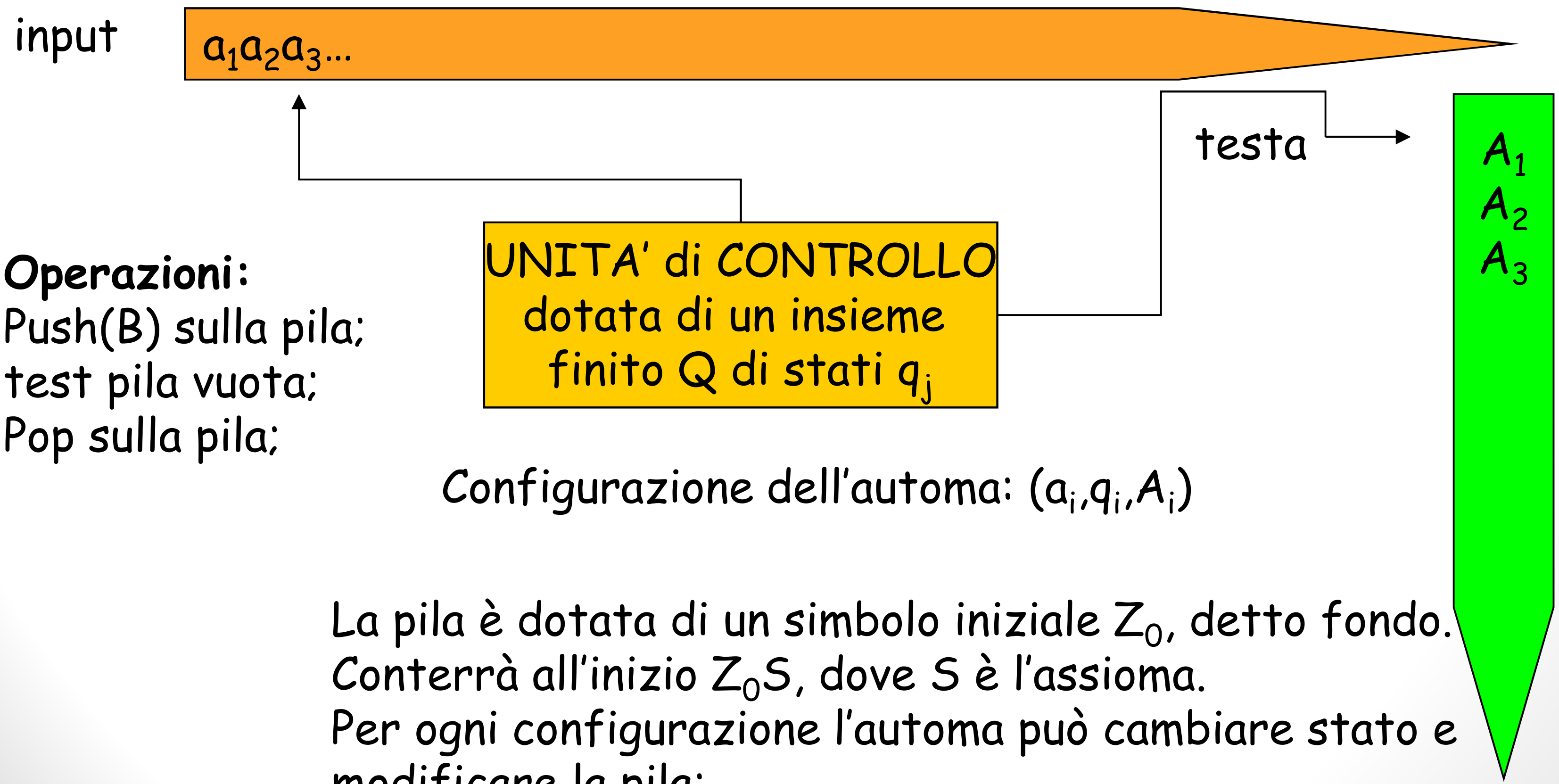
Giovedì 25 Ottobre

Grammatiche CF e automi a pila

- La classe dei linguaggi CF coincide con quella dei linguaggi accettati da automi dotati di una memoria ausiliaria a pila.
- **Un primo algoritmo di riconoscimento potrebbe basarsi sulla simulazione di un automa a pila.**
- Tuttavia il modello da considerare è non deterministico poiché la classe dei linguaggi riconosciuti da automi a pila deterministici è strettamente inclusa in quella dei linguaggi CF.
- Ad ogni passo, l'automa sceglie in modo non deterministico una delle regole applicabili in funzione dello stato, del simbolo corrente e del simbolo in cima alla pila.

Automi a pila

- Sono automi finiti dotati di una memoria ausiliaria organizzata come una pila illimitata:



Operazioni:
Push(B) sulla pila;
test pila vuota;
Pop sulla pila;

Configurazione dell'automa: (a_i, q_i, A_i)

La pila è dotata di un simbolo iniziale Z_0, detto fondo.
Conterrà all'inizio Z_0S, dove S è l'assioma.
Per ogni configurazione l'automa può cambiare stato e modificare la pila;

Dalla grammatica CF all'automa a pila non-deterministico con un solo stato

Regola	Mossa
$A \rightarrow B\alpha_1\alpha_2\ldots\alpha_n$	if testa=A then pop; (mossa spontanea) push($\alpha_n\ldots\alpha_1 B$)
$A \rightarrow b\alpha_1\alpha_2\ldots\alpha_n$	if car=b and testa=A then pop; push($\alpha_n\ldots\alpha_1$); avanza testina lettura;
$A \rightarrow \varepsilon$	if testa=A then pop; (mossa spontanea)
Per ogni carattere b	if car=b and testa=b then pop; avanza testina lettura;
---	if car=$ and testa=Z_0 then accetta; alt;

Esempio

$L=\{a^n b^m \mid n>=m>=1\}$

Regole	Mosse
1. S->aS	if car=a and testa=S then pop; push(S);avanza;
2. S->A	if testa=S then pop; push(A);
3. A->aAb	if car=a and testa=A then pop; push(bA); avanza;
4. A->ab	if car=a and testa=A then pop; push(b);avanza;
5.	if car=b and testa=b then pop; avanza;
6.	if car=$ and testa=Z_0 then accetta; alt;

Esempio: aaabb

La scelta tra 1 e 2 non è deterministica;

Automi a pila e grammatiche CF

- L’automa costruito riconosce una stringa se e solo se la grammatica la genera;
- L’automa simula le derivazioni sinistre della grammatica;
- Si dimostra che la famiglia dei linguaggi CF coincide con quella dei linguaggi riconosciuti da automi a pila
non deterministici con un solo stato.

Complessità di calcolo di un automa a pila – Limite superiore

- Se la grammatica è nella forma di Greibach (ogni regola inizia con un terminale e non contiene altri terminali) non ci sono mosse spontanee e la pila non impila mai simboli terminali.
- Data una stringa x di lunghezza n, se la derivazione da S esiste, avrà lunghezza n.
- Se K è il numero massimo di alternative per ogni non terminale A,

  $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_K$

  allora ad ogni passo si hanno al più K scelte
- Per esplorare tutte le scelte, si ha una complessità esponenziale $O(K^n)$

Si può fare molto meglio!

In un generico automa a pila sono presenti tre forme di non-determinismo

- Incertezza tra più mosse di lettura: per stato q, carattere a e simbolo A della pila, $\delta(q,a,A)$ ha più di un valore;
- Incertezza tra una mossa spontanea (senza lettura) e una mossa di lettura;
- Incertezza tra più mosse spontanee;

Se nessuna delle tre forme è presente, l'automa a pila è deterministico, e il linguaggio riconosciuto è detto context-free deterministico

Parser deterministici e non deterministici

- Un parser è l'implementazione di un automa a pila che riconosce il linguaggio generato da una certa grammatica.

- Se un parser è deterministico allora ogni frase è riconosciuta o non riconosciuta con un solo calcolo (l'automa a pila è deterministico).

Linguaggi deterministici e non deterministici

- *Un linguaggio riconosciuto da parser deterministico si dice linguaggio context-free deterministico e può essere generato da una grammatica non ambigua.*
- *La famiglia dei linguaggi context-free deterministici è strettamente contenuta in quella dei linguaggi context-free.*
- *Se un linguaggio è inerentemente ambiguo allora il parser non può mai essere deterministico.*

Per esempio: L=$\{a^i b^j c^k$ con $i=j$ oppure $j=k\}$
$L=\{a^i b^i c^* \mid i>=0\} \cup \{a^* b^i c^i \mid i>=0\}$.

- Esistono linguaggi non ambigui ma il cui parsing è effettuato da parser non deterministici (due calcoli diversi ma solo uno termina con successo):
Esempio: $L=\{a^n b^n \mid n>0\} \cup \{a^n b^{2n} \mid n>0\}$

Domanda

- E’ deterministico il linguaggio delle parole palindrome generato dalla grammatica?
  S->aSa | bSb | a | b | ε

Tipi di parser

- Metodi Universali
  - (Cocke-Younger-Kasami 1967): usa la programmazione dinamica per stabilire se una data stringa appartiene ad un dato linguaggio context-free. Richiede che la grammatica sia in forma normale di Chomsky. L'algoritmo richiede tempo $O(n^3)$.
  - (Earley 1970): in grado di trattare qualsiasi grammatica CF. L'algoritmo richiede tempo $O(n^3)$.

- Metodi Lineari in $O(n)$, su certe grammatiche riconosciute da automi a pila deterministici:
  - analisi discendente (top-down), più intuitiva, ben adatta a grammatiche semplici;
  - analisi ascendente (bottom-up), più sofisticata, più utilizzata dai generatori automatici di analizzatori sintattici, poichè necessita poche manipolazioni della grammatica.

Algoritmo di Earley

- L'idea è quella di costruire progressivamente tutte le possibili derivazioni leftmost o rightmost compatibili con la stringa in input.
- Durante il procedimento si analizza la stringa in input da sinistra verso destra scartando via via le derivazioni in cui non vi sia corrispondenza tra i simboli derivati e quelli della stringa.
- Se esistono due derivazioni leftmost o rightmost possibili l'algoritmo restituisce i due alberi di derivazione.
- La complessità di calcolo è proporzionale al cubo della lunghezza della stringa da analizzare e si riduce al quadrato se la grammatica non è ambigua e ancora di più se è deterministica.

In dettaglio…

- Data una stringa $x_1x_2\ldots x_n$, l'algoritmo la scandisce da sinistra verso destra e per ogni x_i costruisce l'insieme S[i], costituito da coppie (dot_rule, puntatore).
- Una dot_rule è una produzione di G avente sul lato destro un punto che ne marca una posizione.
- Il puntatore è un intero che indica la posizione dell'input a partire dalla quale è iniziato l'esame della produzione contenuta nella dot_rule.

In altre parole se un elemento di S[j] è del tipo:

$(A\text{->}\alpha.\beta,i)$ con $0<=i<=j$

Ciò significa che:

-si è iniziato l'esame della produzione $A\text{->}\alpha\beta$ a partire dalla posizione i+1 dell'input;

- è già stata esaminata la parte che precede il punto;
- è già stato verificato che α genera $x_{i+1}..x_j$

L'algoritmo

Per semplicità si aggiunge \$ a x e la produzione S'->S\$.

Input $x=x_1 \dots x_n$

$x_{n+1}=\$$

S[0]={(S'->.S\$,0)}

for j=0 **to** n+1 **do**

elabora ogni coppia $(A \text{->} \alpha.\beta,i)$ di S[j] applicando una delle 3 operazioni: scansione, completamento, predizione.

if S[n+1]={(S'->S\$.,0)} **then**

accetta

else rifiuta.

Scansione, completamento, predizione

Esaminiamo uno stato (A->$\alpha.\beta$, i) appartenente a S[j].

- SCANSIONE: Se β inizia con un terminale a, cioè β=aβ', allora se a=x_{j+1} , aggiungi la coppia (A->αa.β', i) a S[j+1].
- PREDIZIONE: Se β inizia con un non terminale B, cioè β=Bβ', allora, per ogni produzione B->γ, aggiungi la coppia (B->.γ, j) a S[j](si predice l'espansione di B a partire dalla posizione j)
- COMPLETAMENTO: Se β è la parola vuota allora data la coppia (A->α., i)

  per ogni coppia (C->η.Aδ, h) di S[i] si aggiunge (C->ηA.δ, h) a S[j] (la predizione A-> α si è verificata quindi si può allungare la parte riconosciuta. Si individuano allora in S[i] tutti gli stati che avevano fatto la predizione
  $\eta \Rightarrow x_{h+1} \dots x_i$. Siccome A$\Rightarrow$ $\alpha \Rightarrow x_{i+1} \dots x_j$, allora $\eta A \Rightarrow x_{h+1} \dots x_j$

Proseguendo a ritroso si costruisce la derivazione e quindi l'albero di derivazione.

Esempio

E->E+E
E->E*E
E->(E)
E->i
S'->E$

x=(i)$

S[0]=
S'->.E$,0
E->.E+E,0
E->.E*E,0
E->.(E),0
E->.i,0

S[1]=
E->(.E),0
E->.E+E,1
E->.E*E,1
E->.(E),1
E->.i,1

S[2]=
E->i.,1
E->(E.),0
E->E.+E,1
E->E.*E,1

S[3]=
E->(E).,0
S'->E.$,0
E->E.+E,0
E->E.*E,0

S[4]=
S'->E$.,0

Nell'operazione di scansione si inserisce un puntatore tra un insieme e un altro. Nell'operazione di completamento all'interno dello stesso insieme.

S'->E$->(E)$->(i)$

Esercizio: provare il parser di i+i+i

Complessità dell'algoritmo

- Ciascun insieme S[j] può avere un numero di coppie che cresce linearmente con j, quindi O(n);
- Le operazioni di scansione e predizione su ogni coppia sono indipendenti da n;
- L'operazione di completamento richiede O(j) per ogni coppia, quindi in totale $O(n^2)$
- Sommando i passi per ogni i si ha $O(n^3)$.
- In pratica l'algoritmo è più veloce: per molte grammatiche è O(n) e per ogni grammatica non ambigua è $O(n^2)$

Esercizio

- Costruire l’algoritmo di Earley per la grammatica

S->A | B

A->aAb | ab

B->aaBb| aab

e la stringa aabb

Esempio

x=aabb$

S->A
S->B
A->aAb
A->ab
B->aaBb
B->aab
S'->S$

S[0]=
S'->.S$,0
S->.A,0
S->.B,0
A->.aAb,0
A->.ab,0
B->.aaBb,0
B->.aab,0

a

S[1]=
A->a.Ab,0
A->a.b,0
B->a.aBb,0
B->a.ab,0
A->.aAb,1
A->.ab,1

a

S[2]=
B->aa.Bb,0
B->aa.b,0
A->a.Ab,1
A->a.b,1
A->.aAb,2
A->.ab,2
B->.aaBb,2
B->.aab,2

b

S[3]=
B->aab.,0
A->ab.,1
S->B.,0
A->aA.b,0
S'->S.$,0

b

S[4]=
A->aAb.,0
S->A.,0
S'->S.$,0

$

S[5]=
S'->S$.,0

S'->S$->A$->aAb$->aabb$

Nota: Se ci sono produzioni A->ε, si effettua subito il completamento (la dot_rule è A->.)

Esercizio

- Siano date la grammatica G e la stringa *aaaa*. Si esegua il riconoscimento della stringa utilizzando l'algoritmo di Earley.

- G: S -> aaS | Saaa | a | ε

Esercizi con Flex

- Scrivere un programma in Flex che riconosca e restituisca tutte le stringhe costituite da lettere minuscole in cui le vocali non compaiono né in ordine crescente né decrescente.
- Scrivere un programma in Flex che nelle parole con un numero pari di vocali sostituisca tutte le vocali con ‘a’, e trasforma in maiuscolo tutte le consonanti. Per le parole con numero dispari di vocali, ogni vocale viene trasformata nella successiva (a->e, e->i, …, u->a), le consonanti doppie vengono dimezzate (rr->r), tutte le altre vengono trasformate in maiuscolo.