Automi a pila

Dispensa del corso di Linguaggi e Traduttori

A.A. 2005-2006

Giovanni Semeraro

Una classe di macchine più potenti rispetto agli automi a stati finiti è costituita dagli ***automi a pila*** (**Push-Down Automaton, PDA**).

Un automa a pila è un automa a stati finiti più un dispositivo di memoria a pila (**stack**).

Sullo stack sono applicabili le operazioni di:
- inserzione di un dato nella pila (**push**)
- analisi del primo elemento della pila (**top**)
- rimozione del primo elemento della pila (**pop**).

Informalmente, le mosse eseguite da un PDA consistono in:
- leggere un simbolo di ingresso
- transitare in un nuovo stato, producendo una nuova configurazione della pila, in funzione del simbolo di ingresso letto e del simbolo in cima alla pila.

Possono essere deterministici o non-deterministici.

Formalmente un automa a pila non-deterministico (NPDA) è una 7-tupla:
$NPDA = (S,\Sigma,V,p,s_0,A_0,F \subset S)$

dove:

- S, insieme di stati
- Σ , alfabeto di ingresso
- V, alfabeto finito di simboli dello stack
- p (*pushdown function*) è una funzione totale da $SxVx(\Sigma \cup \{\varepsilon\})$ a insiemi finiti di SxV^*
- s_0 è lo stato iniziale ed F l'insieme degli stati finali
- A_0 è il simbolo iniziale sullo stack

La ***configurazione*** di un NPDA è un elemento di SxV^*.

Un NPDA **accetta** stringhe che portano dalla configurazione (s_0, A_0) ad un insieme di configurazioni in cui almeno una è un elemento di FxV^*.

Un NPDA accetta la classe dei linguaggi ***liberi da contesto.***

Esempio:

$E \to T \quad | \quad E+T \quad | \quad E-T$

$T+F \quad | \quad T*F \quad | \quad T/F$

$F \to a \quad | \quad b \,|\, c \quad | \quad (E)$

Inizialmente lo stack contiene lo scopo della grammatica (E), più un simbolo speciale (stack vuoto).

Ad ogni mossa dell'automa viene prelevato il simbolo al top dello stack. Se è un simbolo non terminale, viene riscritto usando una delle produzioni e la parte destra della produzione viene messa sullo stack. Se è un simbolo terminale, deve corrispondere al simbolo letto da ingresso.

Automa a pila: M= $(S, \Sigma, V, P, s_1, -|, s_3)$

dove: $S = \{s_1, s_2, s_3\}$, $V = V_N \cup V_T \cup \{-|\}$

$p(s_1, -|, \varepsilon) = \{(s_2, E, -|)\}$

$p(s_2, E, \varepsilon) = \{(s_2, T), (s_2, E+T), (s_2, E-T)\}$

$p(s_2, T, \varepsilon) = \{(s_2, T*F), (s_2, T/F)\}$

$p(s_2, F, \varepsilon) = \{(s_2, a), (s_2, b), (s_2, (E))\}$

$p(s_2, a, a) = \{(s_2, \varepsilon)\}$ $\quad$ $p(s_2, b, b) = \{(s_2, \varepsilon)\}$

$p(s_2, c, c) = \{(s_2, \varepsilon)\}$ $\quad$ $p(s_2, +, +) = \{(s_2, \varepsilon)\}$

$p(s_2, -, -) = \{(s_2, \varepsilon)\}$ $\quad$ $p(s_2, *, *) = \{(s_2, \varepsilon)\}$

$p(s_2, /, /) = \{(s_2, \varepsilon)\}$ $\quad$ $p(s_2, (, ( ) = \{(s_2, \varepsilon)\}$

$p(s_2,),)) = \{(s_2, \varepsilon)\}$

$p(s_2, -|, \varepsilon) = \{(s_3, \varepsilon)\}$

In pratica un NPDA simula la derivazione (leftmost) sul suo stack. Il simbolo che viene riscritto attraverso una produzione è quello più a sinistra (quello che si trova in cima allo stack).

Stringa: a*(b+c) :

	Sequenza di configurazioni	
Ingresso	**Stato**	**Stack**
$a*(b+c)$	s_1	$-\|$
$a*(b+c)$	s_2	$E\ -\|$
$a*(b+c)$	s_2	$T\ -\|$
$a*(b+c)$	s_2	$T*F\ -\|$
$a*(b+c)$	s_2	$F*F\ -\|$
$a*(b+c)$	s_2	$a*F\ -\|$
$*(b+c)$	s_2	$*F\ -\|$
$(b+c)$	s_2	$F\ -\|$
$(b+c)$	s_2	$(E)\ -\|$
$b+c)$	s_2	$E)\ -\|$
$b+c)$	s_2	$E+T)\ -\|$
$b+c)$	s_2	$T+T)\ -\|$
$b+c)$	s_2	$F+T)\ -\|$
$b+c)$	s_2	$b+T)\ -\|$
$+c)$	s_2	$+T)\ -\|$
$c)$	s_2	$T)\ -\|$
$c)$	s_2	$F)\ -\|$
$c)$	s_2	$c)\ -\|$
$)$	s_2	$)\ -\|$
ε	s_2	$-\|$

ε s_3 $-|$

Esistono particolari classi di linguaggi per cui il non-determinismo delle mosse dell'automa viene risolto guardando un certo numero di simboli della sequenza di ingresso (***linguaggi LL(k)***).

PDA deterministici:

Un ***automa a pila deterministico*** (DPDA) è un PDA che può fare al più una mossa in ciascuna configurazione.

La classe di linguaggi accettata dai DPDA non coincide con la classe dei linguaggi liberi da contesto (accettata da NPDA).

Un DPDA accetta ***linguaggi deterministici*** (liberi da contesto).

Esempi di grammatiche LL(1) e loro modalità di riconoscimento mediante un push_down automata:

I parsers LL(1) sono parsers ricorsivo discendenti guidati da una tavola per eseguire il parsing.

II primo L significa che la stringa è processata da sinistra a destra.
Il secondo L dice che e' costruita una derivazione left-most. (1) sta ad indicare che si deve controllare un solo simbolo successivo.

Sono necessarie alcune definizioni:

FIRST(α):
Da una stringa di terminali e non terminali (α) determina l'insieme di terminali che iniziano ogni stringa derivabile da α (inclusa la stringa vuota ε).

Esempio:
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid Id$

FIRST(E)={(,Id}
FIRST(T*Id)={(,Id}
FIRST(Id+Id)={Id}
FIRST(Id)={Id}

FOLLOW(A)

È l'insieme di terminali che possono apparire a destra di A. Se A compare nella forma: $\alpha A \beta$ dove α e β sono stringhe di terminali o non terminali allora
FOLLOW(A) = FIRST(β).

ESEMPIO:

FOLLOW(E) = {),+}
FOLLOW(T) = {+,), *}

Definizione 1:

Una grammatica LL(1) semplice (s-grammatica) è una grammatica libera da contesto (e senza produzioni che generino la stringa vuota) così che per ogni simbolo A appartenente a V_n le alternative per A cominciano con un differente simbolo terminale.

Cioè tutte le sue regole sono della forma:

$$A \to \tau_1\alpha_1 \qquad A \to \tau_2\alpha_2 \qquad \ldots \qquad A \to \tau_n\alpha_n$$

dove $\tau_i \neq \tau_j$ per $i \neq j$ e Ti e $\tau_i \in V_T$

Esempio di s-grammatica (LL(1))

0. $S' \rightarrow S\$$
1. $S \rightarrow aS$
2. $S \rightarrow bA$
3. $A \rightarrow d$
4. $A \rightarrow ccA$

Nota: è stata aumentata con il carattere \$ che non appartiene ai simboli terminali e la produzione $S' \rightarrow S\$$ per facilitarne l'analisi.

Possiamo usare un solo stack.

Traccia per la stringa aabccd$

Input non ancora usato	Stack	Output
aabccd$	S$	0
aabccd$	aS$	1
abccd$	S$	1
abccd$	aS$	11
bccd$	S$	11
bccd$	bA$	112
ccd$	A$	112
ccd$	ccA$	1124
cd$	cA$	1124
d$	A$	1124
d$	d$	11243
$	$	11243

La classe delle grammatiche trattate in questo modo si può espandere.
In particolare, si può rimuovere la condizione che il simbolo più a sinistra nella parte destra della produzione sia terminale.

Esempio di grammatica LL(1);

0. $S' \rightarrow S\$$
1. $S \rightarrow ABe$
2. $A \rightarrow dB$
3. $A \rightarrow aS$
4. $A \rightarrow c$
5. $B \rightarrow AS$
6. $B \rightarrow b$

La grammatica non è una s-grammatica (le produzioni 1 e 5 violano la condizione), ma possiamo comunque utilizzare un solo stack:

Traccia per la stringa adbbebe$

Input non ancora usato	Stack	Output
adbbebe$	S$	O
adbbebe$	ABe$	1
adbbebe$	aSBe$	13
dbbebe$	SBe$	13
dbbebe$	ABeBe$	131
dbbebe$	dBBeBe$	1312
bbebe$	BBeBe$	1312
bbebe$	bBeBe$	13126
bebe$	BeBe$	13126
bebe$	beBe$	131266
ebe$	eBe$	131266
be$	Be$	131266
be$	be$	1312666
e$	e$	1312666
$	$	1312666

Definizione 2:
Una grammatica LL(1) è una grammatica libera da contesto (e senza produzioni che generino la stringa vuota) così che per ogni simbolo A appartenente a V_n le alternative per A cominciano con un differente simbolo terminale o cominciano con simboli non terminali da cui si derivano simboli terminali disgiunti.

Cioè tutte le sue regole sono della forma:

$$A \to \alpha_1 \qquad A \to \alpha_2 \quad \dots \quad A \to \alpha_n$$

dove $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \varnothing$
per $i \neq j$ e $\alpha_i, \alpha_j \in V_T \cup V_N$

e $FIRST(\alpha) = \{w \mid \alpha \Rightarrow *w(w_a w_r - w_n) \text{ and } w \in V_T\}$

Applicandolo alla grammatica precedente:

per A:
$FIRST(dB) \cap FIRST(aS) = \{d\} \cap \{a\} = \varnothing$
$FIRST(dB) \cap FIRST(c) = \{d\} \cap \{c\} = \varnothing$
$FIRST(aS) \cap FIRST(c) = \{a\} \cap \{c\} = \varnothing$

per B:
$FIRST(AS) \cap FIRST(b) = \{a,c,d\} \cap \{b\} = \varnothing$

Quindi è LL(1).

Consideriamo adesso grammatiche che possono avere la stringa vuota nelle produzioni.

Definizione 3:
Una grammatica LL(1) è una grammatica libera da contesto (e con produzioni che possono generare la stringa vuota) così che per ogni simbolo A appartenente a V_n le alternative per A cominciano con un differente simbolo terminale o cominciano con simboli non terminali da cui si derivano simboli terminali disgiunti.

Cioè tutte le sue regole sono della forma:

$$A \to \alpha_1 \qquad A \to \alpha_2 \quad \ldots \quad A \to \alpha_n$$

dove $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \varnothing$
per $i \neq j$ e $\alpha_i, \alpha_j \in V_T \cup V_N$

ed inoltre se $\alpha_i \Rightarrow {}^*\varepsilon$

$$FIRST(\alpha_i) \cap FOLLOW(A) = \varnothing$$

Esempi di grammatiche non LL(1):

$A \rightarrow dA$
$A \rightarrow dB$
$A \rightarrow f$
$B \rightarrow g$

$FIRST(dA) \cap FIRST(dB) = \{d\}$

$S \rightarrow Xd$
$X \rightarrow C$
$X \rightarrow bA$
$C \rightarrow \varepsilon$
$B \rightarrow d$

$FIRST(Ba) \cap FOLLOW(X) = \{d\}$

L'ALGORITMO

LL(1) parsing è un parsing ricorsivo discendente guidato da un'opportuna tavola. Invece di chiamate ricorsive, si consulta una tavola per determinare la prossima azione da eseguire e si utilizza uno stack esplicito.

Immaginiamo di avere già creato la tavola e consideriamo il seguente algoritmo:

LL(1) Parsing:

Push "#" onto stack
Initialize the stack with the start symbol
REPEAT WHILE stack is not empty
    CASE top of the stack is:
    terminal: IF input symbol matches the top of the stack, THEN advance input and pop stack, ELSE error.
    non terminal: Use nonterminal and current input symbol to find correct production in the table.
Pop Stack
Push right-hand side of the production onto stack, right-most symbol first (left-most symbol on top).
END REPEAT
IF input is finished, THEN accept, ELSE error.

Esempio:

Si consideri la seguente grammatica di tipo LL(1):

1. $E \rightarrow TE'$
2. $E' \rightarrow +TE' \mid \varepsilon$
3. $T \rightarrow FT'$
4. $T' \rightarrow *FT' \mid \varepsilon$
5. $F \rightarrow Id \mid (E)$

E la seguente tavola che contiene una riga per ogni simbolo non terminale e una colonna per ogni simbolo terminale.

	Id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E′		$E' \rightarrow TE'$			$E' \rightarrow \mid \varepsilon$	$E' \rightarrow \mid \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T′		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow Id$			$F \rightarrow (E)$		

Si consideri poi la seguente stringa di input:
Id*Id + Id

Traccia nello stack:

Input non usato	Stack	Output
Id*Id+Id$	E$	1
Id*Id+Id$	TE'$	13
Id*Id+Id$	FT'E'$	135
Id*Id+Id$	Id T'E'$	135
*Id+Id$	T'E'$	1354
*Id+Id$	* F T' E'$	1354
Id+Id$	F T' E'$	13545
Id+Id$	Id T' E'$	13545
+Id$	T' E'$	135454'
+Id$	E'$	135454'2'
++Id$	T E'$	135454'2'
Id$	T E'$	135454'2'3
Id$	F T' E'$	135454'2'35
Id$	Id T' E'$	135454'2'35
$	T' E'$	135454'2'354'
$	E'$	135454'2'354'2'
$	$	

La tavola viene costruita utilizzando l'algoritmo seguente:

Per ogni produzione $A \rightarrow \alpha$ della grammatica

1. Se α può derivare una stringa che inizia con a, cioè per tutti gli a in FIRST(α):

$TABLE[A,a] = A \rightarrow \alpha$

2. Se α può derivare la stringa vuota e, allora per tutti i b che possono seguire una stringa derivata da A, cioè per tutti i b in FOLLOW(A):

$TABLE[A,b] = A \rightarrow \alpha$

Gli entry indefiniti sono interpretati come errori, mentre se un entry è definito più di una volta la grammatica non è LL(1).

Queste tavole possono essere generate automaticamente.

In pratica questi parser possono essere costruiti scrivendo esplicitamente la tabella o inglobandola direttamente nel programma con un CASE statement. La prima soluzione è più modulare e flessibile.

Oggi esistono molti generatori automatici di parsers.

La maggior parte sono comunque bottom-up.

Esistono comunque parecchi generatori di parsers bottom-up.

Il più noto è YACC nel sistema operativo UNIX.

(Yet Another Compiler).