Implementazione di Linguaggi 2
PARTE 3

Massimo Ancona
DISI Università di Genova

Testo: A.V. Aho, R. Sethi, J.D.Ullman Compilers Principles,Techniques and Tools, Addison Wesley

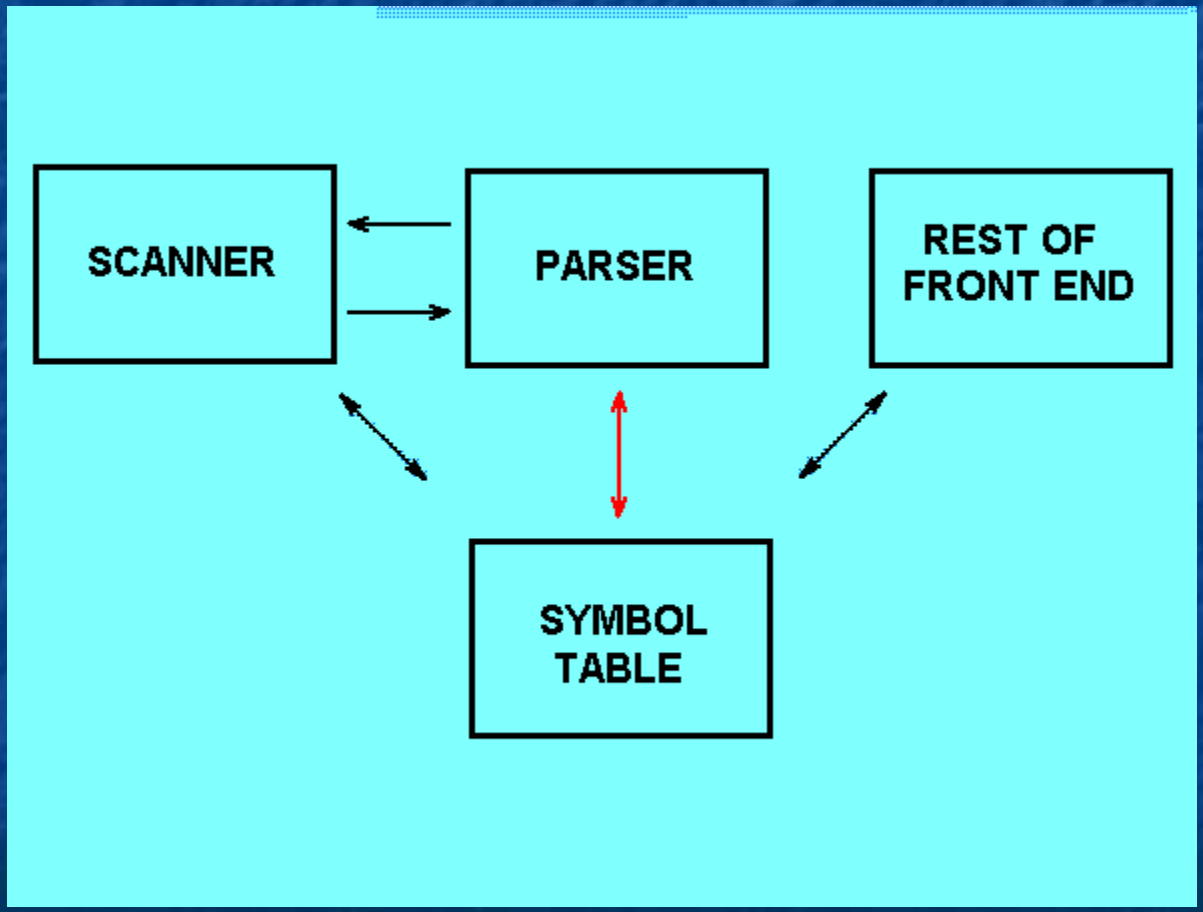
Parsing

Il processo che stabilisce se una stringa di token apprartiene al linguaggio generato da una grammatica G.

Per ogni CFG esiste un parser (con back-tracking) di complessita' $O(n^3)$ (in teoria $O(n^{2.7})$) che analizza una stringa di n token e costruisce un parse tree di essa.

I linguaggi di programmazione hanno algoritmi di complessita' $O(n)$ che leggono l'input una sola volta da sinistra a destra leggendo "avanti" un solo token per volta, detto look-ahead token

Parsing



Parsing: trattamento degli errori

Un parser deve fornire in modo chiaro un segnale di errore per ogni errore sintattico nel programma.

Gli errori in un programma sono di quattro tipi:

- lessicali (es. errori su identificatori o keyword)
- sintattici (es. parentesi sbilanciate)
- semantici (es. operandi incompatibili)
- logici

Parsing: trattamento degli errori

La gestione degli errori deve avere le seguenti proprieta’

- presentazione chiara e precisa
- recuperare velocemente gli errori onde individuare i successivi
- non rallentare eccessivamente il processo di traduzione

Parsing: trattamento degli errori

Le principali strategie di recupero degli errori sono:

- Panic Mode (dopo ogni errore si esegue una sincronizzazione)
- Recovery a Livello di Frase (il parser opera correzioni per continuare l'analisi). Esempio manca ";" ma trova "if": segnala l'inserimento di ";".
- Error productions: si estende la grammatica con produzioni che accettano gli errori piu' comuni.
- Global corrections (trovare il matching piu' prossimo)

Parsing

I parser si dividono in due categorie

TOP-DOWN BOTTOM-UP

A seconda di come costruiscono il parse tree.

I primi costruiscono una derivazione canonica sinistra, i secondi una destra.

I primi sono molto popolari perche' intuitivi e usabili in compiler costruiti a mano. I secondi si applicano a classi di linguaggi piu' generali.

Parser top-down

Metodo (applicato alla G seguente):

1. type→simple|
2. ^id|
3. array[simple] of type
4. simple → integer | char | num dotdot num

A partire dalla radice etichettata dal simbolo iniziale (type) si applicano i seguenti passi:

1. Nel nodo n etichettato A selezionare una produzione A→α e costruire da α i figli di n
2. Determinare il nodo successivo cui associare un sotto albero

Parser top-down

Input: ‘ array[num dotodot num] of integer ‘

type
array [simple] of type

array[num dotodot num] of integer

type
array [simple] of type
num dotodot num

array[num dotodot num] of integer

type
array [simple] of type
num dotodot num simple

array[num dotodot num] of integer

type
array [simple] of type
num dotodot num simple
integer

Parser a Discesa Ricorsiva

Metodo top-down formato da un insieme di procedure ricorsive. Si associa una procedura ad ogni non terminale della grammatica.

Per la G seguente:

1. type→simple|
2. ^id|
3. array[simple] of type
4. simple → integer | char | num dotdot num

Si scrive una proc. per type ed una per simple

Parser a Discesa Ricorsiva

```
proc match(t: token); /* scanner*/
begin
        if lookahead = t then
next_token
        else error;
end;
```

Proc type

```
proc type; /* */
begin
if lookahead in {‘integer’,’char,’num’} then simple
else if lookahead =‘^’ then
begin match(‘^’); match(id) end
else if lookahead = ‘array’ then
begin
   match(‘array’); match(‘[‘); simple; match(‘]’);
   match(‘of’); type
end
else error
end;
```

Proc simple

```
proc simple; /* */
begin
if lookahead =‘integer’ then match(‘integer’)
      else if lookahead = ‘char’ then match(’char’)
else if lookahead =‘num’ then
begin
match(‘num’);match(‘dotdot’); match(‘num’)
        end
else error
end;
```

Codice completo in Io (1)

```
MODULE PostFix(TABLES);
VAR ch: CHAR;

  PROCEDURE Find;
  BEGIN
    DO
      READ(ch)
    UNTIL (ch<>' ')AND NOT EOLN
  END; (* Find*)
```

Codice completo in Io (2)

```
PROCEDURE Expression;
   VAR Op: CHAR;
   PROCEDURE Term;
     PROCEDURE Factor;
     BEGIN
       IF ch='(' THEN  Find; Expression; (* ch=) *)
       ELSE
          WRITE(ch)
       FI; Find
     END; (* Factor *)
```

Codice completo in Io (3)

```
BEGIN (*Term*)
  Factor;
  WHILE ch='*'
  DO
    Find; Factor; WRITE('*')
  OD;
END; (*Term*)
```

Codice completo in Io (4)

```
BEGIN (* Expression*)
   Term;
   WHILE (ch='+') OR (CH='-')
   DO
     Op:=ch;Find; Term; WRITE(Op)
   OD;
 END; (* Expression*)
```

Codice completo in Io (5)

```
BEGIN (* Main*)
  Find;
  DO
    WRITE(' ');
    Expression;
    WRITELN
  UNTIL Ch='.'
END.
```

Parser Predittivo: teoria

Lo sviluppo si basa sulla conoscenza dei primi simboli generabili nella parte destra di una produzione $A \to \alpha$ insieme indicato:

$FIRST_k(\alpha)=\{w \in T^* |\ |w|<k \ \&\ \alpha \Rightarrow^* w \text{ or } |w|=k \ \&\ \exists x \in T^* \ \alpha \Rightarrow^* wx\}$

$FIRST(\alpha)=FIRST_1(\alpha)$

$FIRST_1(\alpha)=\{a \in T \cup \{\varepsilon\} | a=\varepsilon \ \&\ \alpha \Rightarrow^* \varepsilon \text{ OR } \exists x \in T^* \ \alpha \Rightarrow^* ax\}$

Se $A \to \alpha$ e $A \to \beta$ sono due produzioni di G allora esiste un parser a discesa ricorsiva senza backtraking se: $FIRST(\alpha) \cap FIRST(\beta)=\varnothing$

Le produzioni $A \to \varepsilon$ richiedono un trattamento speciale

Parser Predittivo: ricorsione sinistra

Un parser a discesa ricorsiva puo' entrare in loop quando incontra una produzione (o un set di produzioni) che originano una ricorsione sinistra. Ad esempio le produzioni:

$E \rightarrow E+T \mid E-T \mid T$

Sono entrambe ricorsive a sinistra (in E). Questo fa parte di un caso generale

$A \rightarrow A\alpha \mid \beta$

In cui α=+T (-T) e β=T. Questo problema e' eludibile sostituendo le produzioni precednti con:

$A \rightarrow \beta A'$ $A' \rightarrow \alpha A' \mid \varepsilon$

Ricorsione sinistra caso generale

Definizione. Una CFG e' ricorsiva a sinistra se esiste un non terminale A e una stringa α per cui $A \Rightarrow^{+} A\alpha$.

In generale la ricorsione diretta si elimina trasformando tutte le A-produzioni:

$A \rightarrow A\alpha_1$, $A \rightarrow A\alpha_2$,..., $A \rightarrow A\alpha_n$ $A \rightarrow \beta_1$, $A \rightarrow \beta_2$,...,$A \rightarrow \beta_m$ in

$A \rightarrow \beta_1 A'$, $A \rightarrow \beta_2 A'$,...,$A \rightarrow \beta_m A'$;

$A' \rightarrow \alpha_1 A'$ $A \rightarrow \alpha_2 A'$ $A \rightarrow \alpha_b A'$, $A \rightarrow \varepsilon$

La ricorsione indiretta (es. $S \rightarrow Aa|b; A \rightarrow Ac|Sd|\varepsilon$) e' eliminabile con il seguente algoritmo

Ricorsione sinistra indiretta

- Si ordinano i non terminali in modo arbitrario: $A_1, A_2, \ldots, A_n$.
- for i:=1 to n do
  for j:=1 to i-1 do
  replace all $A_i \rightarrow A_j\gamma$ with $A_i \rightarrow \delta_1\gamma|\delta_2\gamma|\ldots|\delta_k\gamma$
  ($A_j \rightarrow \delta_1|\delta_2|\delta_3|\ldots|\delta_k$ are the actual A_j productions)
  od
  od;
  eliminare le ricorsioni dirette dalle A_i-produzioni

Ricorsione indiretta: esempio

- G=({S,A},{a,b,c,d},{S→Aa|b,A→Ac|Sd|ε},S)
- ordinamento: A_1=S, A_2=A.

S→Aa|b sono OK, A→Ac e' OK per loop interno, A→Sd va sostituita da A→Aad|bd.

Le nuove produzioni di G sono

S→Aa|b

A→Ac|Aad|bd|ε

Eliminando la ricorsione diretta si ottiene:

S→Aa|b

A→bdA'|A' A' → cA'|adA'|ε

Fattorizzazione Sinistra

Esempio:

stmt→if expr then stmt else stmt | if expr then stmt schematizzato nel caso generale:

$A \rightarrow \alpha\beta_1 | \alpha\beta_2$ fattorizzabili in $A \rightarrow \alpha A'$ $A' \rightarrow \beta_1 | \beta_2$

Algoritmo generale

Per ogni non terminale A determinare il prefisso piu' lungo α comune a due o piu' alternative. Se $\alpha \neq \varepsilon$ sostituire tutte le produzioni

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 .... | \alpha\beta_n | \gamma$$

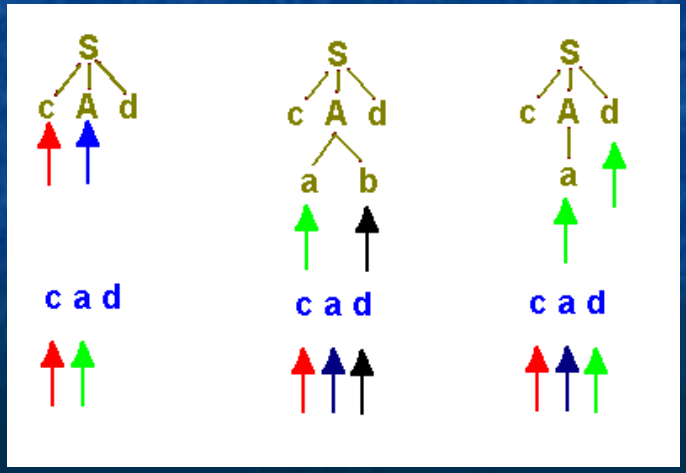
con $A \rightarrow \alpha A' | \gamma$, $A' \rightarrow \beta_1 | \beta_2 | \beta_3 .... | \beta_n$

Costruzione di Parser a discesa ricorsiva

Un parser a discesa ricorsiva puo' essere pensato come un metodo per costruire una derivazione canonica sinistra (o un parsing tree) dell'input. Per una CFG generica il metodo richiede backtracking. Ad esempio le produzioni:

$S \rightarrow cAd$ $A \rightarrow ab|a$

con l'input $w=cad$ forniscono due alternative $A \rightarrow ab$ e $A \rightarrow a$:



Diagrammi di Transizione

Si disegna un diagramma per ogni non terminale. Le transizione sono etichettate sia da terminali (token) sia da non terminali.

- Se una transizione e' etichettata da un token allora la transizione avviene se il token coincide con l'input.
- Una transizione su un non terminale A e' simile ad un'attivazione di procedura: viene attivato il diagramma corrispondente.

La costruzione dei diagrammi di traansizione di una grammatica si effettua come segue:

Diagrammi di Transizione

- Eliminare eventuali ricorsioni sinistre e fattorizzare a sinistra le alternative con prefisso comune
- Per ogni non terminale A creare:
  - uno stato iniziale ed uno finale
  - per ogni produzione $A \rightarrow X_1 X_2 \ldots X_n$ creare un cammino dallo stato inziale a quello finale con archi etichettati X_1, X_2,...,X_n.

Il metodo funziona se i diagrammi finali sono deterministici

Diagrammi di Transizione

Il parser opera come segue:

- parte dallo stato iniziale del simbolo iniziale
- nello stato s, con un arco etichettato a e incidente a t, essendo il simbolo corrente in input a, allora l'input si sposta di un token in avanti e t diviene il nuovo stato.
- Se l'arco (s,t) e' etichettato A allora il parser passa allo stato iniziale del diagramma di A senza spostare l'input.
- Raggiunto lo stato finale di A il parser passa allo stato t precedente (in effetti ha letto A dall'input durante le azioni eseguite passando da s a t dopo aver analizzato la parte destra della produzione)
- se esiste un arco etichettato ε da s a t allora il parser passa da s a t senza muovere il cursore dell'input.

Diagrammi di Transizione esempio

$E \rightarrow E+T|T$

$T \rightarrow T*F|F$

$F \rightarrow (E)|id$

$\Rightarrow$

$E \rightarrow TE'$

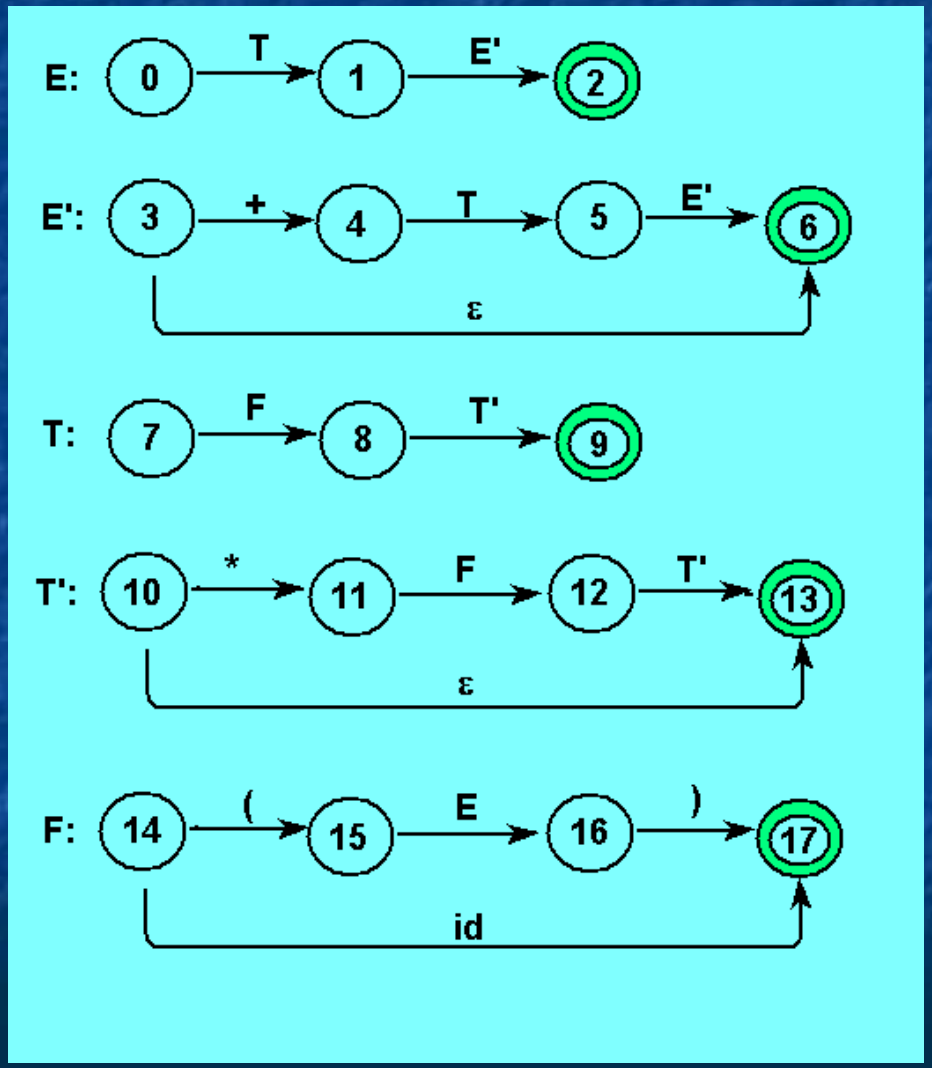
$E' \rightarrow +TE'|\varepsilon$

$T \rightarrow FT'$

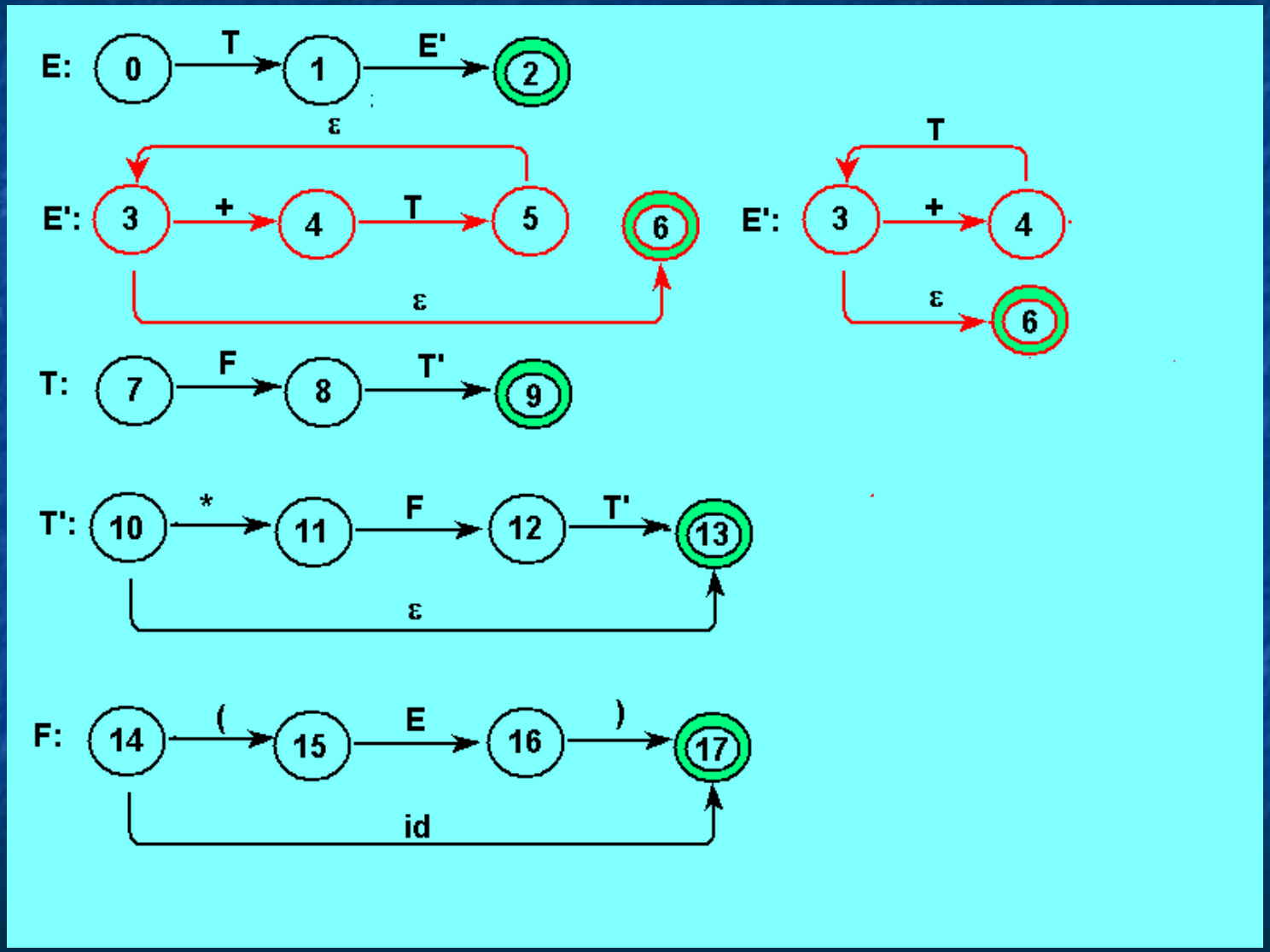
$T' \rightarrow *FT'|\varepsilon$

$F \rightarrow (E)|id$

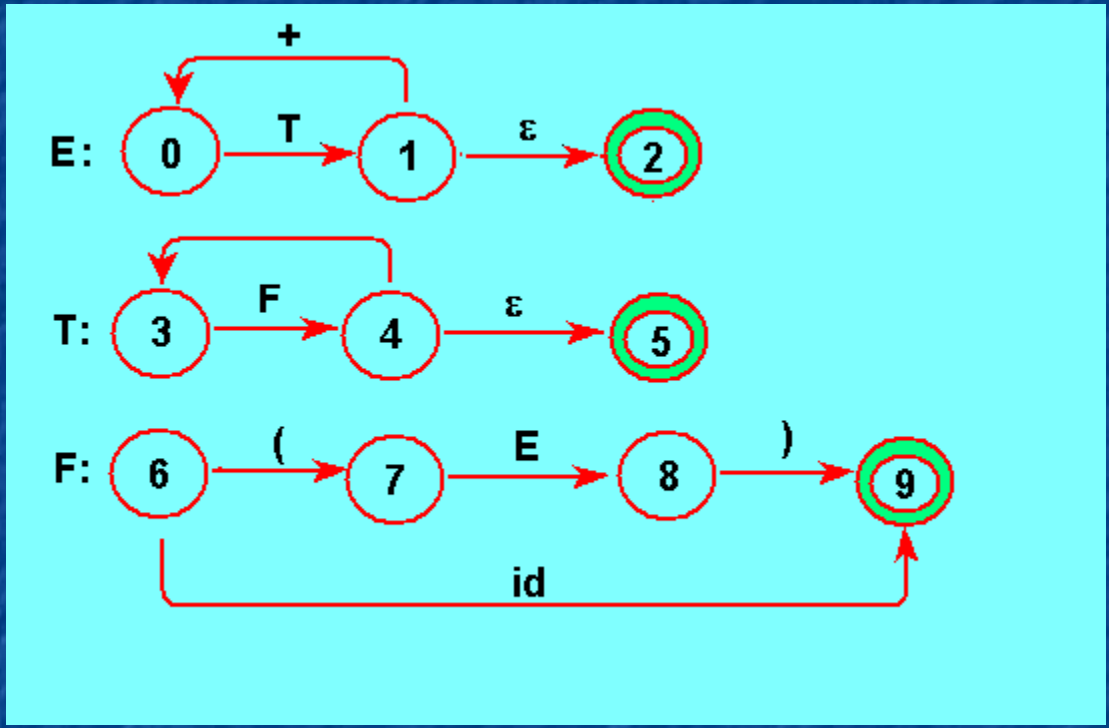
Diagrammi di Transizione esempio



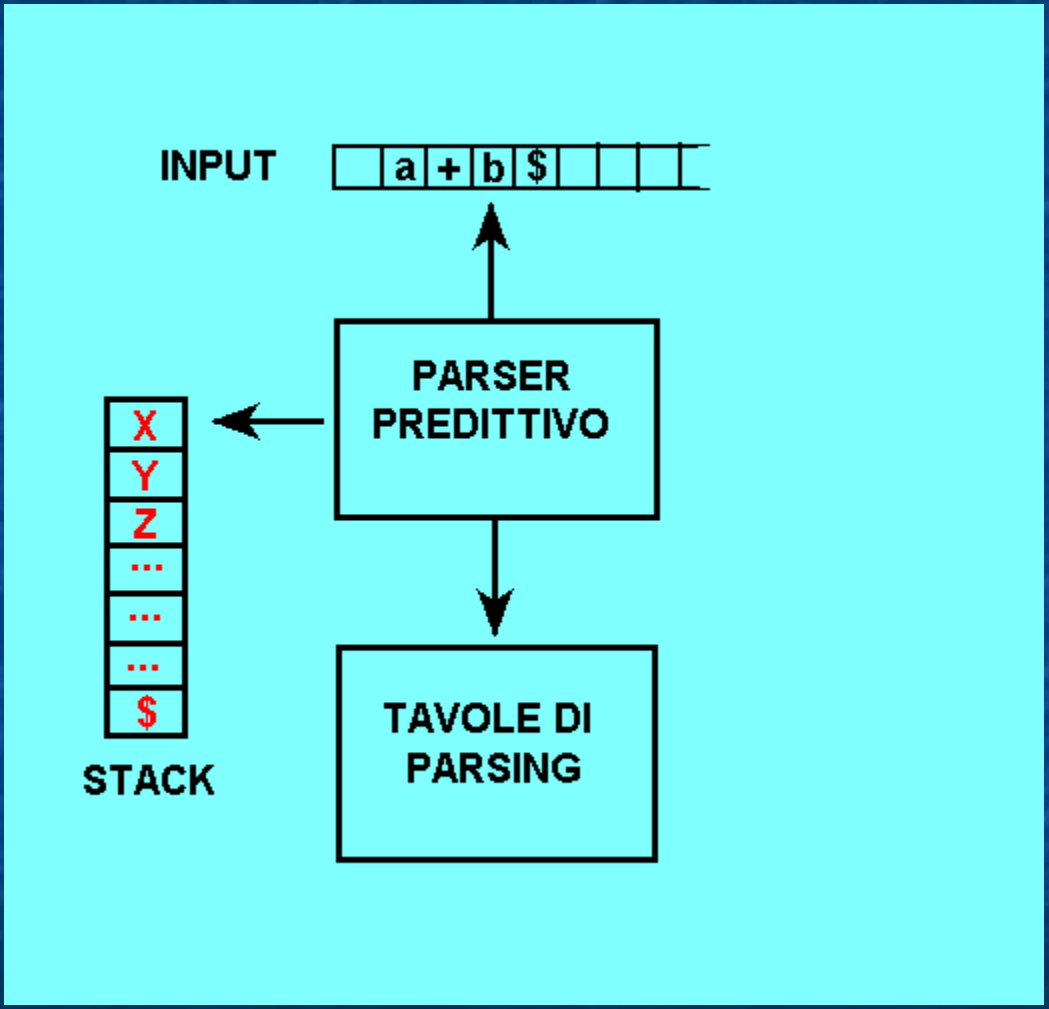
Semplificazione: call in linea



Diagrammi: da EBNF



Parser non ricorsivo



Parser predittivo non ricorsivo

- Il buffer di input contiene il programma da analizzare
- lo stack contiene simboli grammaticali con il terminatore a indicare stack empty. Inizialmente contiene il simbolo iniziale
- la tavola di parsing e' un array bidimensionale M[A,a] indicizzato da un non'terminale (primo indice) e da un terminale o dal terminatore (secondo indice).

Il parser e' controllato dal seguente algoritmo in cui X indica il top dello stack e a il token in input corrente. Esempio

$E \rightarrow E+T|T$ $\quad E \rightarrow TE'$

$T \rightarrow T*F|F \quad \Rightarrow \quad E' \rightarrow +TE'|\varepsilon$

$F \rightarrow (E)|id \quad T \rightarrow FT'$

$T' \rightarrow *FT'|\varepsilon$

$F \rightarrow (E)|id$

Tavole di Parsing

Non terminali	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Algoritmo di Parsing (Universale)

- Se $X=a=\$$ il parser termina e dichiara successo.
- Se $X=a\neq\$$ il parser spila X dallo stack e avanza il cursore al token in input successivo.
- Se $X\in N$ il parser esamina $M[X,a]$.
  - Se $M[X,a]=\{X\rightarrow UVW\}$ il parser sostituisce X con UVW (U sul top) ed emette in output "$X\rightarrow UVW$".
  - Se $M[X,a]=error$ il parser attiva una routine di recovery.

Programma di Parsing

Input: un programma w e una tavola di parsing M di una grammatica G.

Output: se $w \in L(G)$ una derivazione canonica sinistra di w error altrimenti.

Metodo: $\in N$ inizialmente il parser si trova nella configurazione iniziale con $S sullo stack S sul top e w$ in input con ip che punta al primo token di w.

Programma di Parsing cont.

DO /*let X be the TOS & ip*=a */
    IF $X \in T \cup \{\$\}$ THEN
    IF X=a THEN pop X; ip++
    ELSE Error()
    FI
    ELSIF M[X,a]=$\{X \rightarrow Y_1 Y_2 \ldots Y_k\}$ THEN
    pop X; push $Y_k \ldots Y_2 Y_1$;/*Y_1=tos*/
    out "$X \rightarrow Y_1 Y_2 \ldots Y_k$"
    ELSE Error()
    FI
UNTIL X=$

Esempio di Parsing

STACK	INPUT	OUTPUT
$E	id+id*id$	
$E'T	id+id*id$	E→TE'
$E'T'F	id+id*id$	T→FT'
$E'T'id	id+id*id$	F →id
$E'T'	+id*id$	
$E'	+id*id$	T'→ε
$E'T+	+id*id$	E'→+TE'
$E'T	id*id$	
$E'T'F	id*id$	T→FT'
$E'T'id	id*id$	F →id

Esempio di Parsing

STACK	INPUT	OUTPUT
$E'T'id	id*id$	F→id
$E'T'	*id$	
$E'T'F*	*id$	T'→*FT'
$E'T'F	id$	
$E'T'id	id$	F→id
$E'T'	$	F→ε
$E'	$	E'→ε
$	$	

Costruzione delle Tavole di Parsing

Si costruiscono con l'ausilio di due funzioni FIRST e FOLLOW. FIRST l'abbiamo gia' vista: definisce l'insieme dei primi token generati nella parte destra di una produzione $A \rightarrow \alpha$ definito da:

$FIRST_k(\alpha)=\{x \in T^* | (|x|<k \ \& \ \alpha \Rightarrow_{lm}^* x) \text{ or } (|x|=k \ \& \ \exists \beta \in V^* \ \alpha \Rightarrow_{lm}^* x\beta)\}$

$FIRST(\alpha)=FIRST_1(\alpha)$

$FIRST_1(\alpha)=\{a \in T \cup \{\varepsilon\} | a=\varepsilon \ \& \ \alpha \Rightarrow^* \varepsilon \text{ OR } \exists \beta \in V^* \ \alpha \Rightarrow_{lm}^* a\beta\}$

Definizione di FOLLOW

FOLLOW definisce l'insieme dei token che possono seguire una stringa α in una forma sentenziale:

$FOLLOW_k(\alpha)=\{w\in T^*,|w|\leq k \mid S\Rightarrow^*\theta\alpha\gamma \ \& \ w\in FIRSTk(\gamma)\}$

$FOLLOW(\alpha)=FOLLOW_1(\alpha)$

$FOLLOW_1(\alpha)=\{a\in T\cup\{\varepsilon\}| \ S\Rightarrow^*\theta\alpha a\gamma' \ \& \ a\neq\varepsilon)OR(S\Rightarrow^*\theta\alpha)\}$

Costruzione di FIRST(X)

1) Se $X \in T$ allora $FIRST(X)=\{X\}$ altrimenti $FIRST(X)=\emptyset$
2) Se $X \in N$ e $X \rightarrow \varepsilon$ allora $FIRST(X)=FIRST(X) \cup \{\varepsilon\}$
3) Se $X \in N$ e $X \rightarrow Y_1 Y_2 \ldots Y_k$ allora:
   i) $FIRST(X)=FIRST(X) \cup (FIRST(Y_1)-\{\varepsilon\})$
   ii) Se $\exists i>1 \,\&\, \exists a \in T: a \in FIRST(Y_i) \,\&\, \forall k<i \; \varepsilon \in FIRST(Y_k)$ allora $FIRST(X):=FIRST(X) \cup \{a\}$
   iii) Se $\varepsilon \in \bigcap_{1 \le i \le k} FIRST(Y_i)$ allora $FIRST(X):=FIRST(X) \cup \{\varepsilon\}$

Iterare 1) 2) e 3) finche' non e' piu' possibile aggiungere elementi.

Costruzione di FIRST($X_1 \ldots X_n$)

1) $FIRST(X_1,\ldots,X_n) := FIRST(X_1,\ldots,X_n) \cup FIRST(X_1) - \{\varepsilon\}$

2) $\varepsilon \in FIRST(X_1) \Rightarrow FIRST(X_1,\ldots,X_n) := FIRST(X_1,\ldots,X_n) \cup FIRST(X_2) - \{\varepsilon\}$

3) $\varepsilon \in FIRST(X_1) \cap FIRST(X_2) \Rightarrow FIRST(X_1,\ldots,X_n) := FIRST(X_1,\ldots,X_n) \cup FIRST(X_3) - \{\varepsilon\}$

.....

n) $\varepsilon \in \bigcap_{i<n} FIRST(X_i) \Rightarrow FIRST(X_1,\ldots,X_n) := FIRST(X_1,\ldots,X_n) \cup FIRST(X_n) - \{\varepsilon\}$

n+1) $\varepsilon \in \bigcap_{i \le n} FIRST(X_i) \Rightarrow FIRST(X_1,\ldots,X_n) := FIRST(X_1,\ldots,X_n) \cup \{\varepsilon\}$

Costruzione di FOLLOW

1) FOLLOW(S)={$}
2) $A \rightarrow \alpha B \beta \Rightarrow$ FOLLOW(B):=FOLLOW(B) $\cup$ FIRST(β)-{ε}
3) $A \rightarrow \alpha B$ OR ($A \rightarrow \alpha B \beta$ & $\varepsilon \in$ FIRST(β)) $\Rightarrow$ FOLLOW(B):=FOLLOW(B) $\cup$ FOLLOW(A)

Iterare 1) 2) e 3) finche' non e' piu' possibile aggiungere elementi. Esempio:

$E \rightarrow TE'$

$E' \rightarrow +TE' | \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

$F \rightarrow (E) | id$

Esempio di calcolo di FIRST e FOLLOW

$E \rightarrow TE'$

$E' \rightarrow +TE' | \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

$F \rightarrow (E) | id$

FIRST(E)=FIRST(T)=FIRST(F)={(,id}

FIRST(E')={+,ε} FIRST(T')={*,ε}

FOLLOW(E)=FOLLOW(E')={),$}

FOLLOW(T)=FOLLOW(T')={+,),$}

FOLLOW(F)={+,*,),$}

Costruzione delle Tavole

Il metodo si basa sulle seguenti affermazioni

- Se $A \rightarrow \alpha$ & $a \in FIRST(\alpha)$ allora il parser espande A in α se l'input e' a.
- Se $\alpha = \varepsilon$ OR $\alpha \Rightarrow^* \varepsilon$ allora si espande A in α se input appartiene a FOLLOW(A) o la lettura dell'input e' terminata (input=$) e $\$ \in FOLLOW(A)$.

Algoritmo di Costruzione delle Tavole

Input: una grammatica G

Output: La tavola M del parser

1. Per ogni $A \rightarrow \alpha$ eseguire i passi 2 e 3 seguenti:
2. Per ogni terminale $a \in FIRST(\alpha)$ aggiungre $A \rightarrow \alpha$ ad M[A,a].
3. Se $\varepsilon \in FIRST(\alpha)$ aggiungere $A \rightarrow \alpha$ a M[A,b] per ogni terminale $b \in FOLLOW(A)$. Se $\varepsilon \in FIRST(\alpha)$ e $\$ \in FOLLOW(A)$, aggiungere $A \rightarrow \alpha$ a M[a,$].
4. Porre tutte le entry non definite a Error.

Esempio di costruzione delle Tavole

$E \rightarrow TE'$

$E' \rightarrow +TE' | \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

$F \rightarrow (E) | id$

FIRST(E)=FIRST(T)=FIRST(F)={(,id}

FIRST(E')={+,ε} FIRST(T')={*,ε}

FOLLOW(E)=FOLLOW(E')={),$}

FOLLOW(T)=FOLLOW(T')={+,),$}

FOLLOW(F)={+,*,),$}

Grammatiche LL(k)

L'algoritmo di costruzione delle tavole e' applicabile ad ogni grammatica CFG. La tavola M pero' puo' avere entry multiple cioe' contenere piu' produzioni. Esempio:

$S \rightarrow iEtSS' | a;\ S \rightarrow eS | \varepsilon; E \rightarrow b$

$\Rightarrow M[S',e] = \{S' \rightarrow \varepsilon, S' \rightarrow eS\}$

Una grammatica G la cui tavola non contiene entry multiple e' detta LL(1). La prima L significa "left-to-right" e si riferisce alla scansione dell'input, la seconda significa derivazione sinistra, 1 che si usa un solo token di lookahead.

Grammatiche LL(k)

Definizione. Una CFG G=(N,T,P,S) e' LL(k), per un intero fissato k, se e solo se, per ogni coppia di derivazioni sinistre :

1. $S \Rightarrow_{lm}^* wA\alpha \Rightarrow_{lm} w\beta\alpha \Rightarrow^* wx$
2. $S \Rightarrow_{lm}^* wA\alpha \Rightarrow_{lm} w\gamma\alpha \Rightarrow^* wy$

In queste condizioni $FIRST_k(x)=FIRST_k(y) \Rightarrow \beta=\gamma$

G si dice LL se esiste k>0 tale che G sia LL(k).

Teorema. G=(N,T,P,S) e' LL(k) se e solo se:

Se $A\rightarrow\beta$ e $A\rightarrow\gamma$ sono due produzioni distinte allora per ogni $wA\alpha$ per cui $S \Rightarrow_{lm}^* wA\alpha$

si ha:

$FIRST_k(\beta\alpha) \cap FIRST_k(\gamma\alpha)=\varnothing$

Grammatiche LL(k) cont.

Teorema 1 (vale solo per k=1!).
G=(N,T,P,S) e' LL(1) se e solo se:

Se $A \rightarrow \beta$ e $A \rightarrow \gamma$ sono due produzioni distinte allora per ogni $wA\alpha$ per cui $S \Rightarrow_{lm}^* wA\alpha$

si ha:

$FIRST(\beta FOLLOW(A)) \cap FIRST(\gamma FOLLOW(A)) = \varnothing$

Error Recovery

Applicazione della panic mode error recovery: suggerimenti generali.

1. Inserire FOLLOW(A) nell'insieme di sincronizzazione di A. Se si saltano token finche' si trova un token in FOLLOW(A) e si spila A dallo stack si permette al parser di continuare l'analisi.
2. In generale i token in FOLLOW(A) non bastano (un ";" mancante puo' far saltare anche lo statement successivo [es. C++ e Oberon]). In questo caso conviene aggiungere anche i simboli che iniziano uno statement.

Error Recovery

1. Se si aggiungono i token in FIRST(A) all'insieme di sincronizzazione di A e' possibile riprendere il parsing relativo ad A quando si incontra un token in FIRST(A).
2. Se un non terminale puo' generare ε, allora la produzione usata per generare ε puo' essere usata per difetto, posticipando il rilevamento dell'errore. I questo modo si riduce il numero di non terminali da considerare drante il recupero degli errori.
3. Se un token sul top non e' presente in input allora lo si spila segnalando che e' stato inserito e si riprende l'analisi. Questo equivale a considerare ogni token appartenente all'insieme di sincronizzazione di ogni altro token.

Algoritmo di Parsing con error

Il parser opera come segue:

- Se M[A,a] e' blank allora l'input viene scartato.
- Se M[A,a] e' sync allora il top viene spilato
- Se il token sul top non corrisponde a quello in input viene spilato.

Error Recovery

Non terminali	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		Sync
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'	Sync		T→FT'	Sync	Sync
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id	Sync	Sync	F→(E)	Sync	Sync

Parsing con Errori

STACK	INPUT	OUTPUT
$E	)id*+id$	Err. Skip)
$E	id*+id$	id ∈EFIRST(E)
$E’T	id*+id$	
$E’T’F	id*+id$	
$E’T’id	*+id$	
$E’T’	*+id$	
$E’T’F*	*+id$	
$E’T’F	+id$	error M[F,+]=sync
$E’T’	+id$	F is popped
$E’	+id$	
$E’T+	+id$	

Parsing con Errori

STACK	INPUT	OUTPUT
$E'T+	+id$	
$E'T	id$	
$E'T'F	id$	
$E'T'id	id$	
$E'T'id	id$	
$E'T'	$	
$E'	$	
$	$	