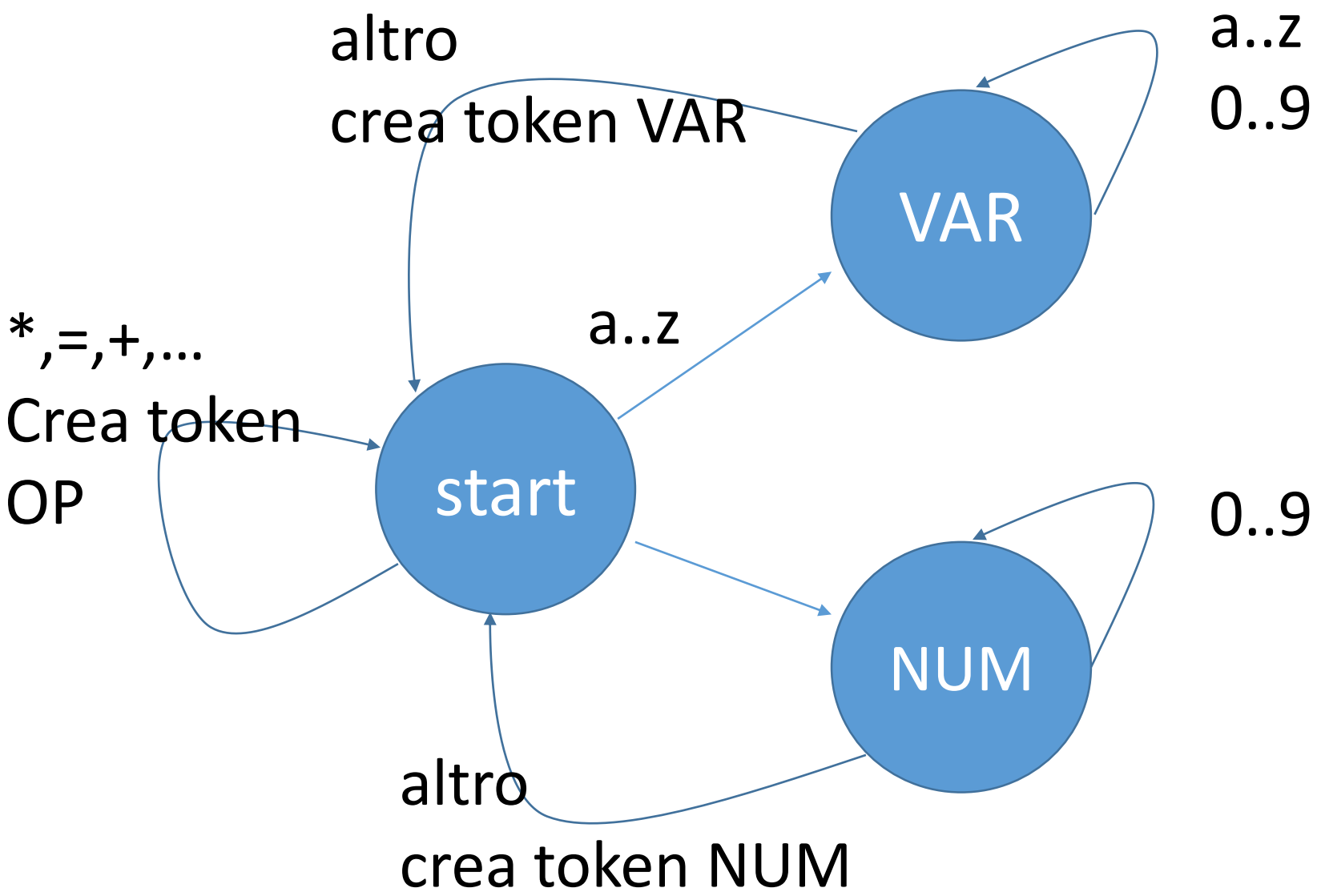
Analisi lessicale e sintattica

Nei programmi troviamo espressioni come questa:
X1=34*y2+z
Si presentano come sequenze di caratteri
‘X’, ‘1’,’=‘,’3’,’4’,’*’,….

Sarebbe meglio VAR(«X1»), OP(‘=‘), NUM(34), OP(‘*’),…

Questi oggetti si chiamano token lessicali

Questa traduzione è compito dell’analisi lessicale ed è realizzata da un’automa a stati finiti che genera output


altro
crea token VAR
a..z
0..9
VAR
*,=,+,…
Crea token
OP
a..z
start
0..9
NUM
altro
crea token NUM

Quindi alla seconda fase di analisi sintattica (parsing) arriva una sequenza di token

L'analisi sintattica si basa su una CFG che è fissata una volta per tutte al momento in cui viene definito il linguaggio

Ci sono diversi modi di farla.

Noi vediamo un caso semplice: analisi «recursive descent» che si può usare se la CFG è LL(1)
Esistono anche CFG LL(2), LL(3),…., LL(k)

Idea del parsing recursive descent

T1,T2,T3,…….

S

T1 deve indicare con quale produzione espandere S

Supponiamo sia S-> AB, allora

T2 deve indicare con quale produzione espandere A

Passeremo a B quando la procedura ha consumato i token che corrispondono all’albero generato da A

LL(2) = possiamo guardare 2 token

LL(k) ne possiamo guardare k

Questo parsing è top-down : l'albero di derivazione è «costruito» dalla radice alla frontiera

Esistono grammatiche che non sono LL(1) e nemmeno LL(k)

Ci sono metodi di parsing bottom-up, cioè dalla frontiera alla radice. Sono quelli utilizzabili per una classe più ampia di CFG.

Sarebbe comodo se ogni produzione:

A -> T B…., dove T sta per Token
Ma in generale non è così.

Però possiamo calcolarci
First(A)={ T1 | T1…Tk in L(A)}
First(A) può contenere ε

Consideriamo questa CFG per le espressioni

1.E-> E + T | T
2.T-> T*F | F
3.F-> NUM | VAR | (E)

Si inizia con First(E)=First(T)=First(F)=∅
Poi si aggiunge (, NUM e VAR

Quindi ora possiamo provare a costruire un parser recursive descent:
Con E se il primo token è NUM posso usare 1.1, ma anche 1.2 !! Non va

Non può funzionare perché la grammatica è ricorsiva a sinistra:
E ->E+T | T e quindi guardando il primo token non potrò mai sapere se si applica E+T o T
Anche E genera T+T….. E quindi non è distinguibile da T

Dobbiamo cambiare la grammatica eliminando la ricorsione a sinistra

E-> T E’
E’-> + T E’ | ε
T-> F T’
T’->*F T’ | ε
F-> NUM | VAR | (E)

1.E-> T E’

2.E’-> + T E’ | ε

3.T-> F T’

4.T’->*F T’ | ε

5.F-> NUM | VAR | (E)

First	E	E’	T	T’	F
0	0	0	0	0	0
1		+ ε		* ε	(NUM VAR
2			(NUM VAR		
3	(NUM VAR				

	(	VAR	NUM	*	+	)
E	1	1	1			
E’					2.1	
T	3	3	3			
T’				4.1		
F	5.3	5.2	5.1			

Non sappiamo come usare le ε-produzioni

Dobbiamo sapere per ogni variabile quali terminali possono venire subito dopo
Follow(A)={T | S =>* α A β e First(β) contiene T}

1.E-> T E’

2.E’-> + T E’ | ε

3.T-> F T’

4.T’->*F T’ | ε

5.F-> NUM | VAR | (E)

Follow	E	E’	T	T’	F
0	$	0	0	0	0
1	)	$	+		*
2		)	$	+	
3			)	$	+
4				)	$
5					)

1.E-> T E’	Follow(E)={$,)}
2.E’-> + T E’ | ε	Follow(E’)={$,)}
3.T-> F T’	Follow(T)={$,),+}
4.T’->*F T’ | ε	Follow(T’)={$,),+}
5.F-> NUM | VAR | (E)	Follow(F)={$,),+,*}

	(	VAR	NUM	*	+	)	$
E	1	1	1				
E’					2.1	2.2	2.2
T	3	3	3				
T’				4.1	4.2	4.2	4.2
F	5.3	5.2	5.1				

	(	VAR	NUM	*	+	)	$
E	1	1	1				

```
Token* parseE(Token* T)
{
 switch (T[0])
case ‘(‘, VAR, NUM :
        Token * k1=parseT(T+1); return parseE’(k1);

default: throw errore

}
```

	(	VAR	NUM	*	+	)	$
T’				4.1	4.2	4.2	4.2

```
Token* parseT’(Token * T)
{
  switch(T[0])
  case ‘*’ : Token* k1=parseF(T+1);
                  return parseT’(k1);
  case ‘+’, ’)’, ’$’ : return T;

}
```