



3. Analisi sintattica

(Parte II: Esercizio 3.2)

Implementazione in Java di un
analizzatore sintattico a discesa ricorsiva
per espressioni di un linguaggio di
programmazione semplice

Esercizio 3.2

- Si consideri una grammatica per un semplice linguaggio di programmazione, espressa mediante un insieme di produzioni (next slide)
- Come nell'Esercizio 3.1, le **variabili** sono denotate con le **parentesi angolari** (per esempio $\langle \text{prog} \rangle, \langle \text{statlist} \rangle, \langle \text{statlistp} \rangle$, ecc.).
- I terminali della grammatica corrispondono ai token descritti in Sezione 2 (in Tabella 1).

Token	Pattern	Nome
Numeri	Costante numerica	256
Identificatore	Lettera seguita da lettere e cifre	257
Relop	Operatore relazionale ($\langle, \rangle, \langle =, \rangle, \langle ==, \rangle, \langle >, \rangle$)	258
Assegnamento	assign	259
To	to	260
Conditional	conditional	261
Option	option	262
Do	do	263
Else	else	264
While	while	265
Begin	begin	266
End	end	267
Print	print	268
Read	read	269
Disgiunzione		270
Congiunzione	&&	271
Negazione	!	33
Parentesi tonda sinistra	(40
Parentesi tonda destra)	41
Parentesi quadra sinistra	[91
Parentesi quadra destra]	93
Parentesi graffa sinistra	{	123
Parentesi graffa destra	}	125
Somma	+	43
Sottrazione	-	45
Moltiplicazione	*	42
Divisione	/	47
Punto e virgola	;	59
Virgola	,	44
EOF	Fine dell'input	-1

Tabella 1: Descrizione dei token del linguaggio

Esercizio 3.2

```

<prog> ::= <statlist> EOF

<statlist> ::= <stat> <statlist>

<statlist> ::= ; <stat> <statlist> | ε

<stat> ::= assign(<expr> to <idlist>) lista
        | print [ <exprlist> ] espressioni
        | read [ <idlist> ] lista id
        | while ( <bexpr> ) <stat>
        | conditional [ <optlist> ] end
        | conditional [ <optlist> ] else <stat> end
        | { <statlist> }

<idlist> ::= ID <idlist>

<idlist> ::= , ID <idlist> | ε

<optlist> ::= <optitem> <optlist>

<optlist> ::= <optitem> <optlist> | ε

<optitem> ::= option ( <bexpr> ) do <stat>

<bexpr> ::= RELOP <expr> <expr>

<expr> ::= + ( <exprlist> ) | - <expr> <expr>
        | * ( <exprlist> ) | / <expr> <expr>
        | NUM | ID

<exprlist> ::= <expr> <exprlist>

<exprlist> ::= , <expr> <exprlist> | ε
    
```

- Scrivere un analizzatore sintattico a discesa ricorsiva per la grammatica.
- Espressioni booleane:
 - Si noti che RELOP corrisponde a un elemento dell'insieme $\{==, <>, <=, >=, <, >\}$
- Sintassi *scheme-like*: espressioni aritmetiche, assegnamento e operatori relazionali in notazione prefissa (polacca)
- Espressioni aritmetiche:
 - notazione prefissa
 - possono anche comprendere ID
 - Operatori
 - * + : varianti n-arie: $n \geq 1$
 - - / : binarie
 - compaiono nelle espressioni booleane, nell'assegnamento, etc. -> [impatto su insieme guida](#)

Esercizio 3.2

- La grammatica è LL(1)?

```
 $\langle prog \rangle ::= \langle statlist \rangle EOF$   
 $\langle statlist \rangle ::= \langle stat \rangle \langle statlistp \rangle$   
 $\langle statlistp \rangle ::= ; \langle stat \rangle \langle statlistp \rangle \mid \varepsilon$   
 $\langle stat \rangle ::=$   
    assign  $\langle expr \rangle$  to  $\langle idlist \rangle$   
    | print [  $\langle exprlist \rangle$  ]  
    | read [  $\langle idlist \rangle$  ]  
    | while (  $\langle bexpr \rangle$  )  $\langle stat \rangle$   
    | conditional [  $\langle optlist \rangle$  ] end  
    | conditional [  $\langle optlist \rangle$  ] else  $\langle stat \rangle$  end  
    | {  $\langle statlist \rangle$  }  
 $\langle idlist \rangle ::= ID \langle idlistp \rangle$   
 $\langle idlistp \rangle ::= , ID \langle idlistp \rangle \mid \varepsilon$   
 $\langle optlist \rangle ::= \langle optitem \rangle \langle optlistp \rangle$   
 $\langle optlistp \rangle ::= \langle optitem \rangle \langle optlistp \rangle \mid \varepsilon$   
 $\langle optitem \rangle ::= option ( \langle bexpr \rangle ) do \langle stat \rangle$   
 $\langle bexpr \rangle ::= RELOP \langle expr \rangle \langle expr \rangle$   
 $\langle expr \rangle ::=$   
    + (  $\langle exprlist \rangle$  )    |    -  $\langle expr \rangle \langle expr \rangle$   
    | * (  $\langle exprlist \rangle$  )    |    /  $\langle expr \rangle \langle expr \rangle$   
    | NUM    | ID  
 $\langle exprlist \rangle ::= \langle expr \rangle \langle exprlistp \rangle$   
 $\langle exprlistp \rangle ::= , \langle expr \rangle \langle exprlistp \rangle \mid \varepsilon$ 
```



Esercizio 3.2: Tip of the day!

```

<prog> ::= <statlist> EOF

<statlist> ::= <stat> <statlist>

<statlistp> ::= ; <stat> <statlistp> | ε

<stat> ::= assign <expr> to <idlist>
        | print [ <exprlist> ]
        | read [ <idlist> ]
        | while ( <beexpr> ) <stat>
        | conditional [ <optlist> ] end
        | conditional [ <optlist> ] else <stat> end
        | { <statlist> }

<idlist> ::= ID <idlistp>

<idlistp> ::= , ID <idlistp> | ε

<optlist> ::= <optitem> <optlistp>

<optlistp> ::= <optitem> <optlistp> | ε

<optitem> ::= option ( <beexpr> ) do <stat>

<beexpr> ::= RELOP <expr> <expr>

<expr> ::= + ( <exprlist> ) | - <expr> <expr>
        | * ( <exprlist> ) | / <expr> <expr>
        | NUM | ID

<exprlist> ::= <expr> <exprlistp>

<exprlistp> ::= , <expr> <exprlistp> | ε
    
```

- La grammatica e' LL1?
- Problema delle due produzioni per conditional
- Andate per gradi:
- 1) Modificare la grammatica per ottenere una grammatica LL(1) equivalente;
- 2) Scrivere un analizzatore sintattico a discesa ricorsiva per la grammatica ottenuta.



Approccio

- Lo stesso di 3.1: da estendere opportunamente al linguaggio più ricco e con una sintassi differente per le espressioni aritmetiche
- Seguite lo stesso schema di implementazione suggerito a LFT teoria per grammatiche **LL(1)** e parsificazione deterministica look ahead
 - Partite dall'**insieme guida**
- **Nota bene:** alcuni token di Tabella 1 non vengono usati (poco male). Quali?
 - Cosa avremmo potuto farcene?

Riflessioni

- **Nota bene:** alcuni token di Tabella 1 non vengono usati (poco male). Quali?
 - Cosa avremmo potuto farcene?

Token	Pattern	Nome
Numeri	Costante numerica	256
Identificatore	Lettera seguita da lettere e cifre	257
Relop	Operatore relazionale (<,>,<=,>=,==,<,>)	258
Assegnamento	assign	259
To	to	260
Conditional	conditional	261
Option	option	262
Do	do	263
Else	else	264
While	while	265
Begin	begin	266
End	end	267
Print	print	268
Read	read	269
Disgiunzione		270
Congiunzione	&&	271
Negazione	!	33
Parentesi tonda sinistra	(40
Parentesi tonda destra)	41
Parentesi quadra sinistra	[91
Parentesi quadra destra]	93
Parentesi graffa sinistra	{	123
Parentesi graffa destra	}	125
Somma	+	43
Sottrazione	-	45
Moltiplicazione	*	42
Divisione	/	47
Punto e virgola	;	59
Virgola	,	44
EOF	Fine dell'input	-1

Tabella 1: Descrizione dei token del linguaggio

Esempi di programmi

esempio_semplice

```
1 assign 10 to a,b;
2 print[a,b];
3 read[x,y];
4 print[1,+(2,3,4)];
5 conditional [
6     option (> x y) do print[x]
7     option (== x y) do print[x,y]
8 ]
9     else print[y] end;
10 while (> x 0) {
11     assign - x 1 to x;
12     print[x]
13 }
```


Esempi di programmi

max_tre_num

```
1 read[x,y,z];
2 conditional [
3     option (> x y) do
4         conditional [ option (> x z) do print[x] ]
5         else print[z] end
6     ]
7 else
8     conditional
9     [ option (> y z) do print[y] ] else print[z] end
10 end|
```

Nota bene

- Errori a cascata

