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 ⟨prog⟩,⟨statlist⟩, ⟨statlistp⟩, 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 (<,>,<=,>=,==,<>)	258
Assegnamento	`assign`	259
To	`to`	260
If	`if`	261
Else	`else`	262
While	`while`	263
Begin	`begin`	264
End	`end`	265
Print	`print`	266
Read	`read`	267
Disgiunzione	`||`	268
Congiunzione	`&&`	269
Negazione	`!`	33
Parentesi tonda sinistra	`(`	40
Parentesi tonda destra	`)`	41
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

$\langle prog \rangle$ ::= $\langle statlist \rangle$ EOF

$\langle statlist \rangle$ ::= $\langle stat \rangle$ $\langle statlistp \rangle$

$\langle statlistp \rangle$ ::= ; $\langle stat \rangle$ $\langle statlistp \rangle$ | ε

$\langle stat \rangle$ ::= assign $\langle expr \rangle$ to $\langle idlist \rangle$
| print ($\langle exprlist \rangle$)
| read ($\langle idlist \rangle$)
| while ($\langle bexpr \rangle$) $\langle stat \rangle$
| if ($\langle bexpr \rangle$) $\langle stat \rangle$ end
| if ($\langle bexpr \rangle$) $\langle stat \rangle$ else $\langle stat \rangle$ end
| { $\langle statlist \rangle$ }

lista espressioni

lista id

$\langle idlist \rangle$ ::= ID $\langle idlistp \rangle$

$\langle idlistp \rangle$ ::= , ID $\langle idlistp \rangle$ | ε

$\langle bexpr \rangle$ ::= 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

$\langle exprlist \rangle$ ::= $\langle expr \rangle$ $\langle exprlistp \rangle$

$\langle exprlistp \rangle$ ::= , $\langle expr \rangle$ $\langle exprlistp \rangle$ | 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 > = 1
    - - / : binarie
  - compaiono nelle espressioni booleane, nell'assegnamento, etc. -> impatto su insieme guida

Esercizio 3.2

$$
\begin{array}{rcl}
\langle prog \rangle & ::= & \langle statlist \rangle \ \texttt{EOF} \\
\\
\langle statlist \rangle & ::= & \langle stat \rangle \ \langle statlistp \rangle \\
\\
\langle statlistp \rangle & ::= & \texttt{;} \ \langle stat \rangle \ \langle statlistp \rangle \ \mid \ \varepsilon \\
\\
\langle stat \rangle & ::= & \texttt{assign} \ \langle expr \rangle \ \texttt{to} \ \langle idlist \rangle \\
& \mid & \texttt{print} \ \texttt{(} \ \langle exprlist \rangle \ \texttt{)} \\
& \mid & \texttt{read} \ \texttt{(} \ \langle idlist \rangle \ \texttt{)} \\
& \mid & \texttt{while} \ \texttt{(} \ \langle bexpr \rangle \ \texttt{)} \ \langle stat \rangle \\
& \mid & \texttt{if} \ \texttt{(} \ \langle bexpr \rangle \ \texttt{)} \ \langle stat \rangle \ \texttt{end} \\
& \mid & \texttt{if} \ \texttt{(} \ \langle bexpr \rangle \ \texttt{)} \ \langle stat \rangle \ \texttt{else} \ \langle stat \rangle \ \texttt{end} \\
& \mid & \{ \ \langle statlist \rangle \ \} \\
\\
\langle idlist \rangle & ::= & \texttt{ID} \ \langle idlistp \rangle \\
\\
\langle idlistp \rangle & ::= & \texttt{,} \ \texttt{ID} \ \langle idlistp \rangle \ \mid \ \varepsilon \\
\\
\langle bexpr \rangle & ::= & \texttt{RELOP} \ \langle expr \rangle \ \langle expr \rangle \\
\\
\langle expr \rangle & ::= & \texttt{+} \ \texttt{(} \ \langle exprlist \rangle \ \texttt{)} \ \mid \ \texttt{-} \ \langle expr \rangle \ \langle expr \rangle \\
& \mid & \texttt{*} \ \texttt{(} \ \langle exprlist \rangle \ \texttt{)} \ \mid \ \texttt{/} \ \langle expr \rangle \ \langle expr \rangle \\
& \mid & \texttt{NUM} \ \mid \ \texttt{ID} \\
\\
\langle exprlist \rangle & ::= & \langle expr \rangle \ \langle exprlistp \rangle \\
\\
\langle exprlistp \rangle & ::= & \texttt{,} \ \langle expr \rangle \ \langle exprlistp \rangle \ \mid \ \varepsilon
\end{array}
$$

- La grammatica è LL(1)?

Esercizio 3.2: Tip of the day!

$$
\begin{array}{rcl}
\langle prog \rangle & ::= & \langle statlist \rangle \ \texttt{EOF} \\
\langle statlist \rangle & ::= & \langle stat \rangle \ \langle statlistp \rangle \\
\langle statlistp \rangle & ::= & \texttt{;} \ \langle stat \rangle \ \langle statlistp \rangle \ \mid \ \varepsilon \\
\langle stat \rangle & ::= & \texttt{assign} \ \langle expr \rangle \ \texttt{to} \ \langle idlist \rangle \\
 & \mid & \texttt{print} \ \texttt{(} \ \langle exprlist \rangle \ \texttt{)} \\
 & \mid & \texttt{read} \ \texttt{(} \ \langle idlist \rangle \ \texttt{)} \\
 & \mid & \texttt{while} \ \texttt{(} \ \langle bexpr \rangle \ \texttt{)} \ \langle stat \rangle \\
 & \mid & \texttt{if} \ \texttt{(} \ \langle bexpr \rangle \ \texttt{)} \ \langle stat \rangle \ \texttt{end} \\
 & \mid & \texttt{if} \ \texttt{(} \ \langle bexpr \rangle \ \texttt{)} \ \langle stat \rangle \ \texttt{else} \ \langle stat \rangle \ \texttt{end} \\
 & \mid & \texttt{\{} \ \langle statlist \rangle \ \texttt{\}} \\
\langle idlist \rangle & ::= & \texttt{ID} \ \langle idlistp \rangle \\
\langle idlistp \rangle & ::= & \texttt{,} \ \texttt{ID} \ \langle idlistp \rangle \ \mid \ \varepsilon \\
\langle bexpr \rangle & ::= & \texttt{RELOP} \ \langle expr \rangle \ \langle expr \rangle \\
\langle expr \rangle & ::= & \texttt{+} \ \texttt{(} \ \langle exprlist \rangle \ \texttt{)} \ \mid \ \texttt{-} \ \langle expr \rangle \ \langle expr \rangle \\
 & \mid & \texttt{*} \ \texttt{(} \ \langle exprlist \rangle \ \texttt{)} \ \mid \ \texttt{/} \ \langle expr \rangle \ \langle expr \rangle \\
 & \mid & \texttt{NUM} \ \mid \ \texttt{ID} \\
\langle exprlist \rangle & ::= & \langle expr \rangle \ \langle exprlistp \rangle \\
\langle exprlistp \rangle & ::= & \texttt{,} \ \langle expr \rangle \ \langle exprlistp \rangle \ \mid \ \varepsilon
\end{array}
$$

- La grammatica e' LL1?
- Problema delle due produzioni per **if**
- 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
If	`if`	261
Else	`else`	262
While	`while`	263
Begin	`begin`	264
End	`end`	265
Print	`print`	266
Read	`read`	267
Disgiunzione	`||`	268
Congiunzione	`&&`	269
Negazione	`!`	33
Parentesi tonda sinistra	`(`	40
Parentesi tonda destra	`)`	41
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

```
assign 10 to a,b;
print(a,b);
read(x,y);
print(1,+(2,3,4));
if (> x y) print(x) else print(y) end;
while (> x 0) {
    assign - x 1 to x;
    print(x)
}
```

Esempi di programmi
max_tre_num

```
read(x,y,z);
if (> x y)
    if (> x z) print(x) else print(z) end
else
    if (> y z) print(y) else print(z) end
end
```

Nota bene

- Errori a cascata