FIRST PRACTICE

Recursive Descent Parser

Language Processors

Year 2022 - 2023

Laura Belizón Merchán (100452273)
Jorge Lázaro Ruiz (100452172)

Laura Belizón Merchán (100452273)
Jorge Lázaro Ruiz (100452172)

First design of the grammar

We designed the following grammar:

$$
\begin{aligned}
&S ::= \; EnC \mid Nn \\
&N ::= 0 \mid 1 \mid 2 \mid \; \ldots \\
&E ::= (OPP) \\
&P ::= E \mid N \\
&C ::= S \mid \lambda \\
&O ::= \; + \mid - \mid * \mid /
\end{aligned}
$$

Where S is the axiom, N is the nonterminal for the numbers, E is the nonterminal for expressions, P stands for "parameter", O stands for "operator" and C is the nonterminal that allows us to handle multiple expressions in one line and stands for "continue". The terminal "n" stands for the newline character, "\n".

The grammar defined in the previous section meets all LL(1) conditions. Here is the corresponding LL(1) parse table generated by JFLAP:

	(	)	+	9	n	$
C	S			S		
E	(OPP)					
N				9		
O			+			
P	E			N		
S	EnC			Nn		

Where *+* and *9* are stand-ins for any operator and number, respectively, and *n* represents *n*.

Development of a recursive descent parser

The design for this item was straightforward. Using our previously designed grammar, we managed to define the functions responsible for parsing our new nonterminals as sequences of other parsing functions.

The full code for this iteration can be viewed here.

Modifying the grammar to handle simple variables

Here is the modified version of the grammar:

$$
\begin{aligned}
&S ::= EnC \\
&E ::= (A \mid N \mid V \\
&C ::= S \mid \lambda \\
&N ::= 0 \mid 1 \mid \ldots \\
&V ::= \mathrm{A} \mid \mathrm{B} \mid \ldots \mid \mathrm{Z} \mid \mathrm{a} \mid \mathrm{b} \mid \ldots \mid \mathrm{z} \\
&O ::= \; + \mid - \mid * \mid / \\
&A ::= OEE \mid !VE
\end{aligned}
$$

Laura Belizón Merchán (100452273)
Jorge Lázaro Ruiz (100452172)

Where we added new nonterminals: V represents a variable, and A can stand for “assign” or “arithmetic”, since it can be derived into two different productions that complete the expression into either a statement assigning a value to a variable or an arithmetic operation.

Here is the grammar’s LL(1) parse table, proving that it meets the conditions for a recursive descent parser to be developed for it.

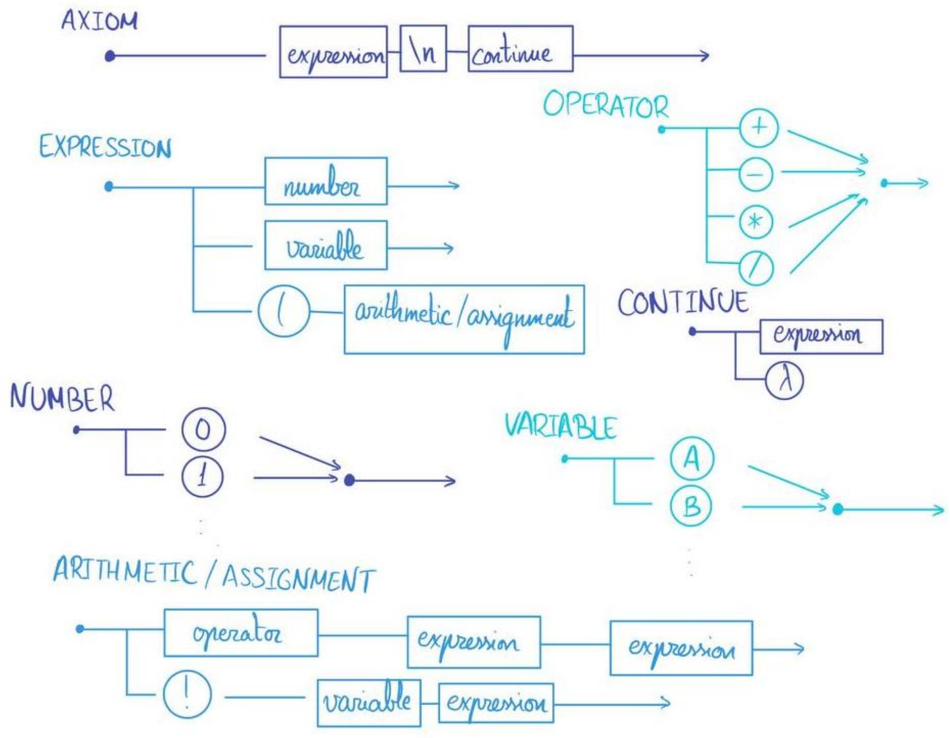
	(	)	+	9	m	n	V	$
A			OEE)		mVE)			
C	S			S			S	λ
E	(A			N			V	
N				9				
O			+					
S	EnC			EnC			EnC	
V							v	

Where *+*, *9* and *v* are stand-ins for any operator, number and letter, respectively, and *m* represents *!*.

Extended development of the parser

This time we had to add a new token for variables and a new global variable for storing the letter, just like we did with numbers. The value of each variable is stored in an array, which is accessed via the index number of a letter (its ASCII code, counting from ‘A’).

Syntax diagram


AXIOM
expression
\n
continue
OPERATOR
+
−
*
/
EXPRESSION
number
variable
(
arithmetic/assignment
CONTINUE
expression
λ
NUMBER
0
1
VARIABLE
A
B
ARITHMETIC / ASSIGNMENT
operator
expression
expression
!
variable
expression