Precedence parsing table

xpojez00, xbudin05

November 12, 2021

1 Introduction

Operator precedence parser is used as a bottom-to-top parser in our Syntax analyzer for expressions only. We have opted for the implementation via table. Expressions are considered all possibilities, which are accepted by LL-grammar rules of $< expression >$[1]. We have introduced a new symbol (See Precedence parsing table). This symbol is used to notify the parser about function, which needs to be recursively parsed. We also need to allow this recursive call to write in the sequence of rules applied for later code generation. As mentioned above, we might use already used $< expression >$ rules to guarantee equality between the parsing of $< expression >$ between top-to-bottom and bottom-to-top parsers. As we have allowed the use of boolean variables and boolean expression our table might appear larger than expected.

2 Introduction to the problem

One problem with multiple returned values occurs. With allowance of function calls with expressions as arguments few problems emerge. These arguments are separated by commas. Small problem occurs with parsing commas, where we want them to be parsed only upon function calls. Another problem comes from the fact, that function can be called with zero arguments. To preserve determinism we can not allow use of ϵ. Problem with function having arguments as function calls makes the need for recursive parsing more apparent.

3 Our current solution

Function's arguments have to create in some way a tree[2], which will be represented with a root and as only non-terminal E[3], which is then added to the function call as a single argument. This E has to be able to consist of further arguments, where each argument is an expression as well. This is solved by allowance of rule as follows: $E \rightarrow E, E$. To preserve simplicity we have opted to create one more rule, which will create an so-called *void* E, which is an expression. This E is then by default appended as first argument of every function. As we can see the rule still allows for function calls with single or multiple arguments but also solves the problem of potential ϵ. Every new argument's tree is is then appended with the afford-mentioned rule to create a single E tree containing arguments.

Only when parsing through a function call we are allowed to consume commas, otherwise this is not allowed, to preserve determinism with top-to-bottom parser and not consume it's commas.

If there is an function call within current function call we can recursively call expression parsing upon this function call and append the whole E as a single E in our currently parsed argument.

Problem with multiple returned values is currently being discussed. Current idea of the early solution is to use the rule $E \rightarrow E, E$ to redeploy return values from the function. The first value is used always. The remaining values are used only in case, that the function is called as the last one and we use the amount of values needed to fill all arguments of the function call. Otherwise only the first value is used and the tree is cut

[1] See Syntax_analyzer.pdf

[2] We are not creating tree just a sequence of rules. Referencing to tree just for easier explanation and picturing

[3] Semantic action upon this subtree is currently being discussed. Some ideas are already thought of

4 Terminals

$T = \{"(", ")", "\#", \textbf{not}, "*", "/", "//", "+", "-", "..", ">", "<", ">=", "<=", "==", "\sim=", \textbf{and}, \textbf{or}, i, \$, ",", \textbf{void}\}$[4]

i is considered as set of terminals with equal precedence, where we will less likely make changes unlike to for example relation operators.
$i = \{string, number, integer, id, true, false, nil\}$

Furthermore we would like to point the fact that comma(",") is allowed to be parsed only under afford-mentioned conditions. **Void** is terminal which is not contained in the set of possible tokens (Literal evaluation of *id* is not considered as new token).

5 Rules

For simplicity reasons we create additional two sets.

$binary\_operatos = \{"*", "/", "//", "+", "-", "..", ">", "<", ">=", "<=", "==", "\sim=", \textbf{and}, \textbf{or}\}$

$unary\_operators = \{"\#", \textbf{not}\}$

$E \rightarrow i$
$E \rightarrow \textbf{void}$
$E \rightarrow bE : b \in unary\_operators$
$E \rightarrow EbE : b \in binary\_operators$
$E \rightarrow (E)$
$E \rightarrow i(E)$
$E \rightarrow E, E$

Consider this rule $E \rightarrow i(E)$ deterministic, as there is a new character signalizing for the use of this rule discussed later.

6 Precedence parsing table

$<$ right side precedence, until this character a sequence can be consumed
$>$ left side precedence, until $<$ character a sequence can be consumed
$=$ evaluate till left parenthesis, until ¡(sequence can be consumed
α is a special case for function call, signalizes that until first occurrence of $<$ after α can be consumed
X not a valid combination. Might be considered empty as well

[4] Later referenced without quotes

Token Top	(	)	#	**not**	*	/	//	+	**-**	..	>	<	>=	<=	==	∼=	**and**	**or**	i	$
(	<	=	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	X
)	X	>	X	X	>	>	>	>	>	>	>	>	>	>	>	>	>	>	X	>
#	<	>	X	X	>	>	>	>	>	X	>	>	>	>	>	>	>	>	<	>
not	<	>	X	X	X	X	X	X	X	X	>	>	>	>	>	>	>	>	<	>
*	<	>	<	X	>	>	>	>	>	X	>	>	>	>	>	>	>	>	<	>
/	<	>	<	X	>	>	>	>	>	X	>	>	>	>	>	>	>	>	<	>
//	<	>	<	X	>	>	>	>	>	X	>	>	>	>	>	>	>	>	<	>
+	<	>	<	X	<	<	<	>	>	X	>	>	>	>	>	>	>	>	<	>
-	<	>	<	X	<	<	<	>	>	X	>	>	>	>	>	>	>	>	<	>
..	<	>	X	X	X	X	X	X	X	>	>	>	>	>	>	>	>	>	<	>
>	<	>	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	<	>
<	<	>	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	<	>
>=	<	>	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	<	>
<=	<	>	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	<	>
==	<	>	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	<	>
∼=	<	>	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	<	>
and	<	>	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	>	<	>
or	<	>	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	<	>
i	α	>	X	X	>	>	>	>	>	>	>	>	>	>	>	>	>	>	X	>
$	<	X	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	X