# Precedence parsing table

xpojez00, xbudin05

December 3, 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 $\epsilon$. 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 call. As we can see the rule still allows for function calls with single or multiple arguments but also solves the problem of potential $\epsilon$. Every new argument's tree is then appended with the afford-mentioned rule to create a single $E$ tree containing arguments (See also Temporary forked sequence).

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. This is embedded in the Precedence parsing table

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.

[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

**Note:** 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. Please note this problem is not a problem in Precedence parsing table but rather of Semantic analyzer. Problem is mentioned here as Semantic analyzer is closely bounded with Precedence parsing table.

### 3.1 Temporary forked sequence

When a function's argument is another function and this function returns more than one number, we need to deterministically decide, whether to use only first, some or all returned values. This can be achieved with our solution of so called Temporary forked sequence. This means, that function is returning $E$, which can be then recursively parsed into all returned values as follows: $E \rightarrow E, E$. We split the initial $E$ holding the final return values into to subtrees as $E$ , $E$. We know, that the former $E$ holds the first returned value and the later $E$ contains subtree of all remaining return values. Now, when expecting new argument for the function call we need to check for remaining arguments. If any are present we abandoned this former $E$ subtree and append there the new argument. Otherwise we use as many splits of the former $E$ as the declaration of the return values allows. If there are any excessive we do not use them. In the picture below it is clear, why it is called a Temporary forked sequence. Please note, that when the function returns values into an expression only the first return value is used and others are ignored. This can be achieved by allowing Temporary forked sequence only in function calls. This solution will be implemented in Semantic analyzer rather than Syntax analyzer.

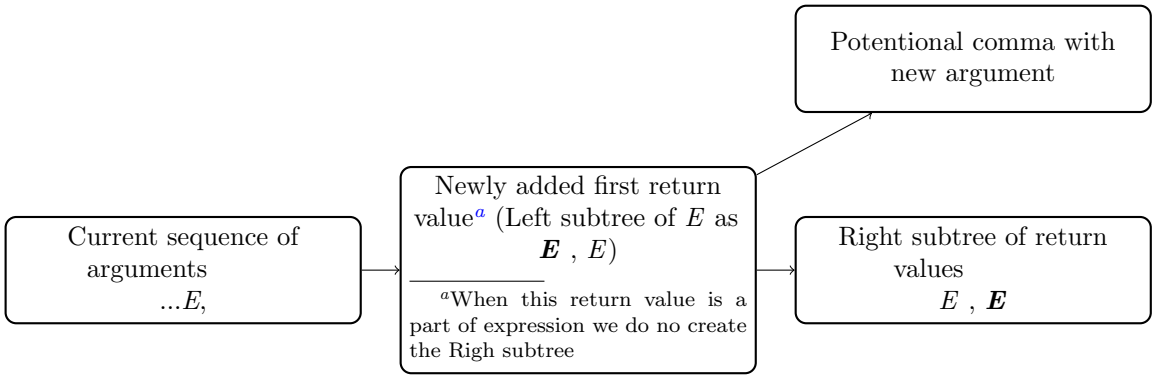


Figure 1: Function call as an argument of function call

## 4 Terminals

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

$T = \Sigma \cup \{"\lceil"\}$
Top $\in T$

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).

[4]Later referenced without quotes

# 5 Rules

For simplicity reasons we create additional two sets.

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

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

$E \to i$
$E \to bE : b \in unary\_operators$
$E \to EbE : b \in binary\_operators$
$E \to (E)$

$E \to \textbf{void}$
$E \to i[E)$
$E \to E, E$

# 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
$=$ just push Token to Top and read next Token
$\alpha$ push [ on Top, create so called *void E*, push it to the Top and read next Token
$\beta$ $<$ to Top, push , on Top, call of Precedence parsing table expecting $E$ as returned value & push this $E$ on Top
$X$ not a valid combination. Might be considered empty as well

| Token / Top | ( | ) | # | **not** | * | / | // | + | - | .. | > | < | >= | <= | == | ~= | **and** | **or** | i | , | $ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ( | < | = | < | < | < | < | < | < | < | < | < | < | < | < | < | < | < | < | < | X | X |
| ) | X | > | 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 | X | X | X | X | X | X | > | > | > | > | > | > | > | > | > | < | X | > |
| > | < | > | < | < | < | < | < | < | < | < | > | > | > | > | > | > | > | > | < | X | > |
| < | < | > | < | < | < | < | < | < | < | < | > | > | > | > | > | > | > | > | < | X | > |
| >= | < | > | < | < | < | < | < | < | < | < | > | > | > | > | > | > | > | > | < | X | > |
| <= | < | > | < | < | < | < | < | < | < | < | > | > | > | > | > | > | > | > | < | X | > |
| == | < | > | < | < | < | < | < | < | < | < | > | > | > | > | > | > | > | > | < | X | > |
| ~= | < | > | < | < | < | < | < | < | < | < | > | > | > | > | > | > | > | > | < | X | > |
| **and** | < | > | < | < | < | < | < | < | < | < | < | < | < | < | < | < | > | > | < | X | > |
| **or** | < | > | < | < | < | < | < | < | < | < | < | < | < | < | < | < | < | > | < | X | > |
| i | $\alpha$ | > | X | X | > | > | > | > | > | > | > | > | > | > | > | > | > | > | X | X | > |
| , | X | > | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | > | X |
| [ | $\beta$ | = | $\beta$ | $\beta$ | $\beta$ | $\beta$ | $\beta$ | $\beta$ | $\beta$ | $\beta$ | $\beta$ | $\beta$ | $\beta$ | $\beta$ | $\beta$ | $\beta$ | $\beta$ | $\beta$ | $\beta$ | < | X |
| $ | < | X | < | < | < | < | < | < | < | < | < | < | < | < | < | < | < | < | < | X | X |

| | |
|---|---|
| **Note:** | $\beta$ appears in positions, where it does not make sense as expression can not begin with for example an binary operator but recursive Precedence parsing should parse this as not an expression but consume it until right parenthesis or comma |

# 7 Implementation details

As Precedence parsing table should parse only an expression, it should not consume token, which does not belong to the expression. This means, we should check, if we would be able to apply valid rule in the future or not. This way we may say, that the next token either does not belong to us or in case of stack not looking like

$$\$E$$

there has been an error between the current token and previously parsed expressions. We would have found this error after we would have found invalid combination or parsed whole expression as left side of each rule is always $E$.

In $\beta$ we have to return read token and read after the recursive Precedence parsing call ends.

Later, if we get comma (,) as Token and we get character $<$ we do not push comma (,) itself on stack, as that is done in $\beta$ character. Might implement with a new $\gamma$ character.