# Building a Recursive Parser

## by the example of parsing and computing arithmetic parenthesis-free expressions

**Evgeny Pyshkin**

**Mikhail Glukhikh**

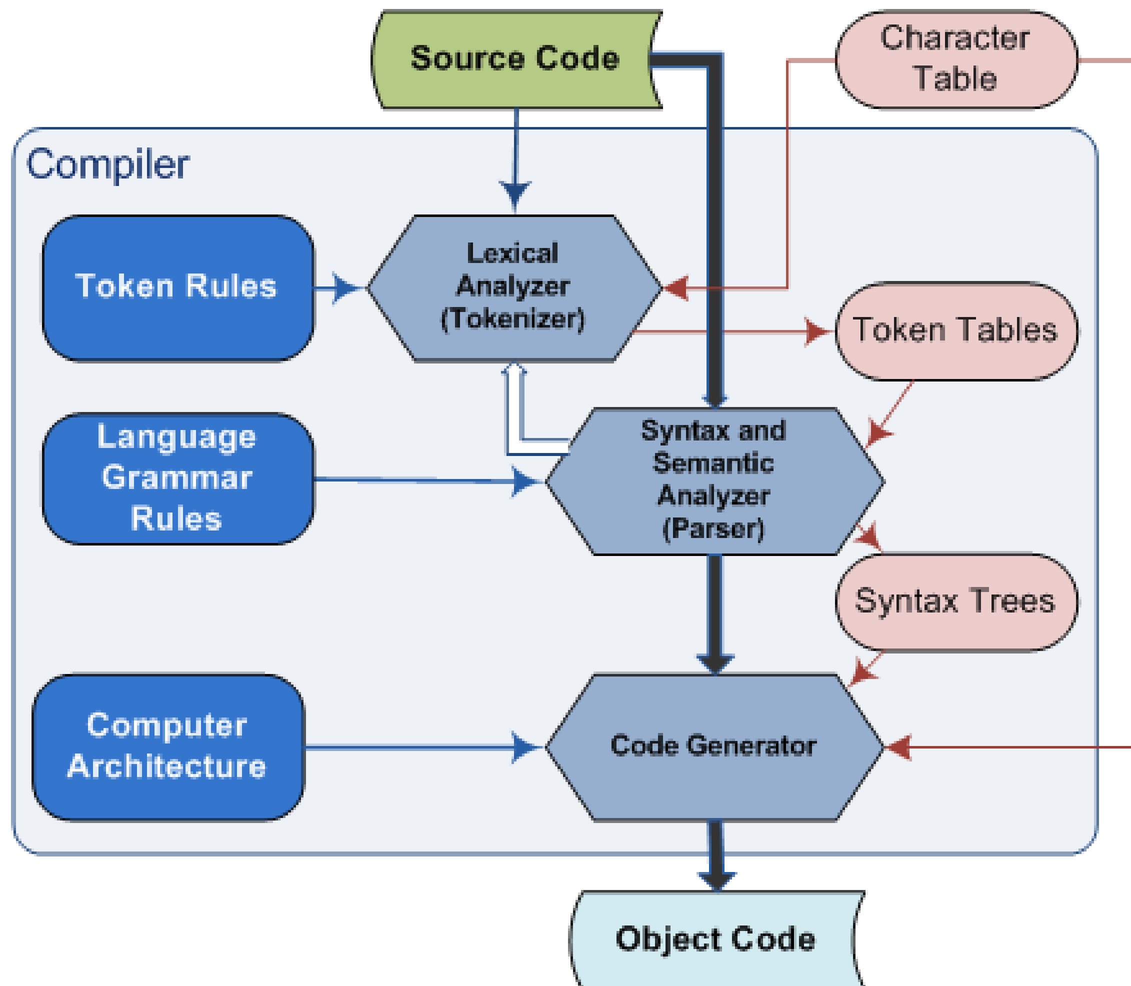
*Dept. of Computer Systems and Software Engineering*

*St. Petersburg State Polytechnic University*

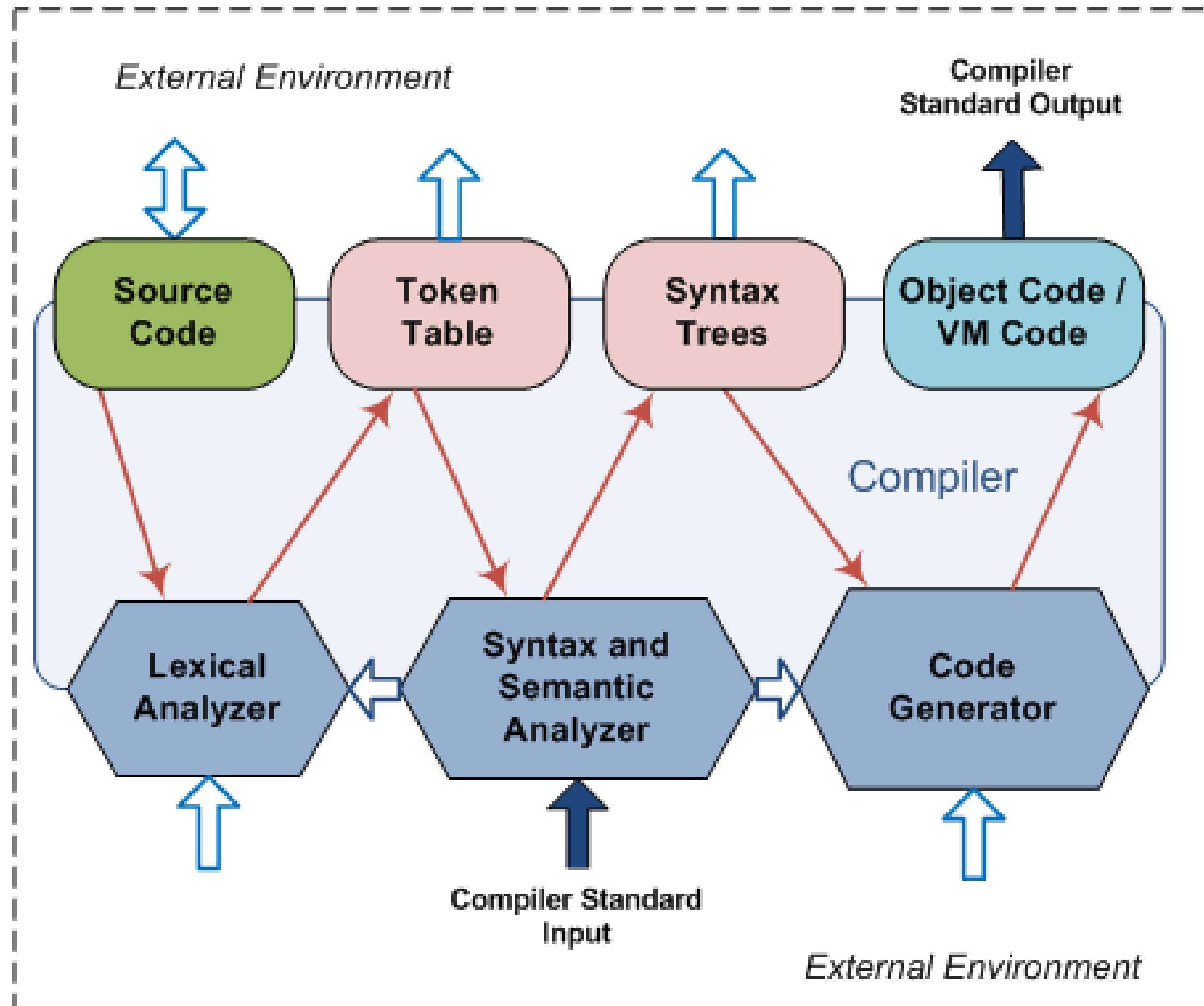
# Compiler Design

- A Compilation Problem
- "Classical" Architecture vs Component Architecture
- Compiler Components
  - Lexical Parser (Tokenizer, Lexer)
  - Syntactic and Semantic Parser (Parser)
  - Code Generation
- A Recursive Parser as a kind of Multi-Aspect Task in Software Education

# A "Classical" Architecture



# Component Architecture



# The Task

- Parsing expressions defined by a recursive context-free grammar
  - a simplified compiler-like problem
  - a kind of authentic problem illustrating basic concepts of lexical and syntactic analysis and code execution
  - the task is complex enough to examine parsing methods without risk to lose significant details but suitable to fit academic requirements

# Defining an Expression Grammar

$$E = (V_N, V_T, P, S)$$

$V_N$ ***a finite set of nonterminal symbols***

$V_T$ ***a finite set of terminal symbols***

$P$ ***a start symbol***

$S$ ***a finite set of the grammar production rules***

# Syntactic Analysis Rules

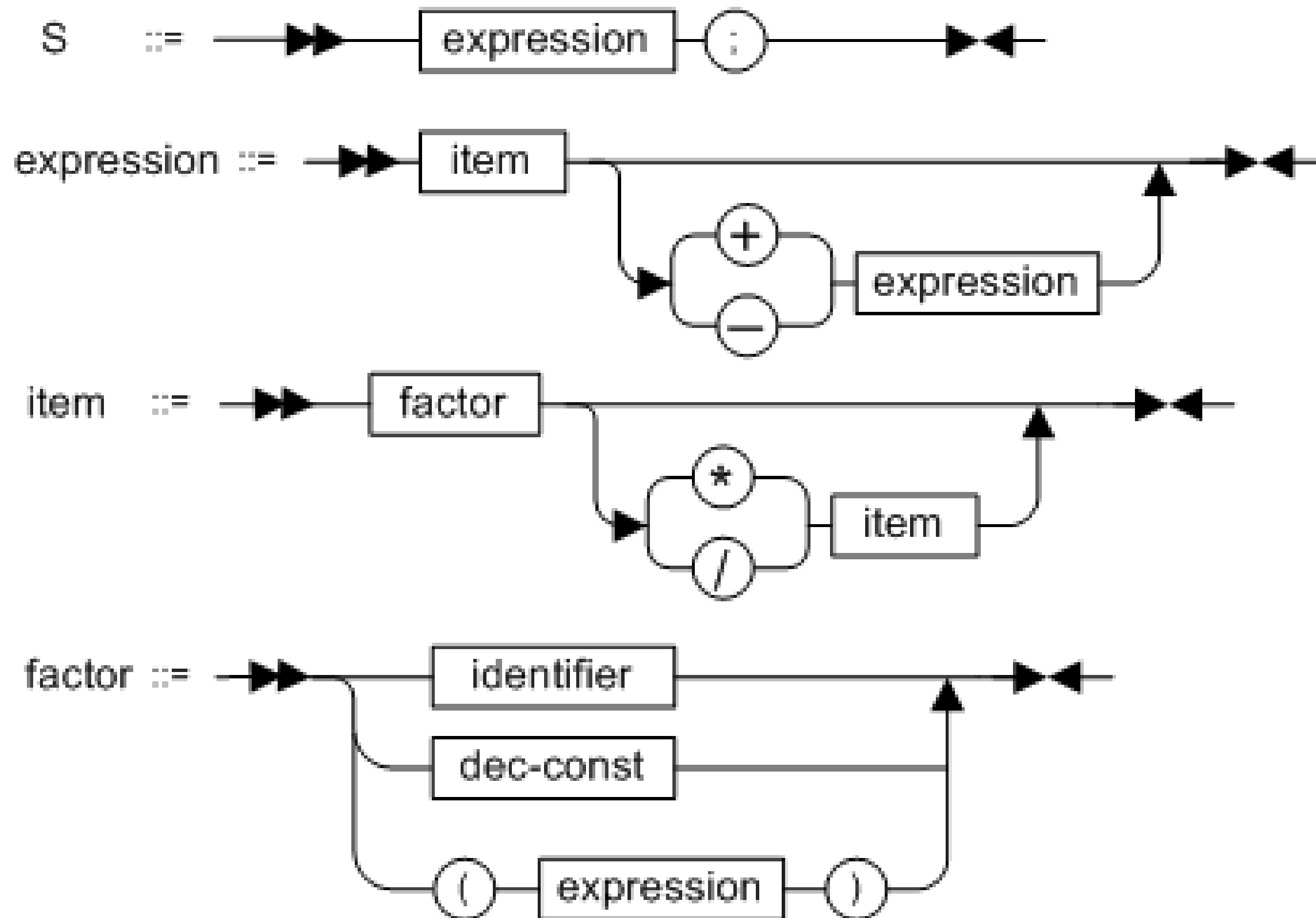
```
S ::= <expression> ";"

<expression> ::= <item>
<expression> ::= <item>{"+"|"-"}<expression>

<item> ::= <factor>
<item> ::= <factor>{"*"|"/"}<item>

<factor> ::= <identifier>
<factor> ::= <dec-const>
<factor> ::= "("<expression>")"
```

# Syntax Diagrams



# Lexical Analysis Rules

```
<identifier> ::=   <letter>[<letter>|<dec-digit>]...

<dec-const> ::= [<dec-digit>]...

<letter> ::=   {'_'|'A'|'B'|...|'Z'|'a'|'b'|...|'z'}

<dec-digit> ::=
{'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'}
```

# Lexical Parser Information Model and Bitwise Operations

- The first problem is how to recognize whether the input symbol does correspond to ***a set of symbols used in the common lexical context (synterms)***
- Possible implementation uses sets and set operations [Wirth, 1973]
  - *Easy for cases when synterms correspond to contiguous ranges (A..Z, 0..9)*
  - *Not optimal for more complex situations (performance issues)*
  - *Doesn't fit* ***intersecting synterms***

*What does it mean?*

# Basic Synterm Classes

- Single-character synterm
  - Set of synterms consisting of only one symbol
- Multi-character synterm
  - Other synterms

- Intersecting synterms
  - Synterms correspond to intersecting sets of symbols

# Synterm Recognition

Element of synterm table

/* .65 0x41 'A' */ LAT | LD10 | D10

*Combined synterm definition*

| | |
|---|---|
| 0000 0000 0000 0001 | LAT |
| 0000 0000 0000 0010 | LD10 |
| 0000 0000 0000 1000 | D10 |
| 0000 0000 0000 1011 | Synterm of character 'A' |

*Combined synterm recognition*

| | |
|---|---|
| 0000 0000 0000 1011 | Input character synterm |
| 1000 0000 0000 0001 | MASK_LAT |
| 0000 0000 0000 0001 | LAT |

Element of synterm table

/* .41 0x29 ')' */ ONLY | ')'

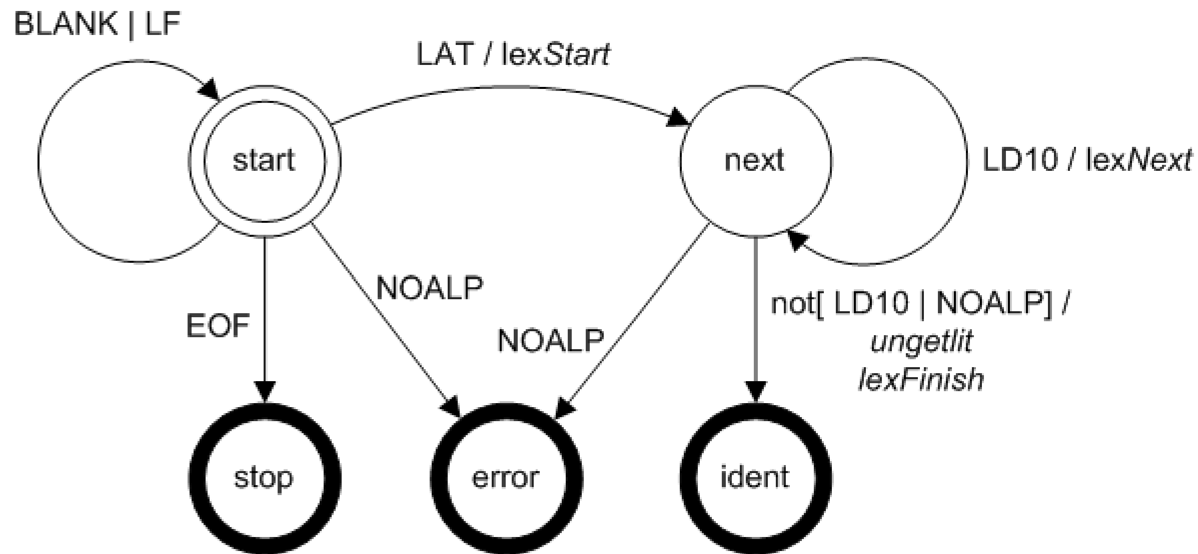
*Single-character synterm definition*

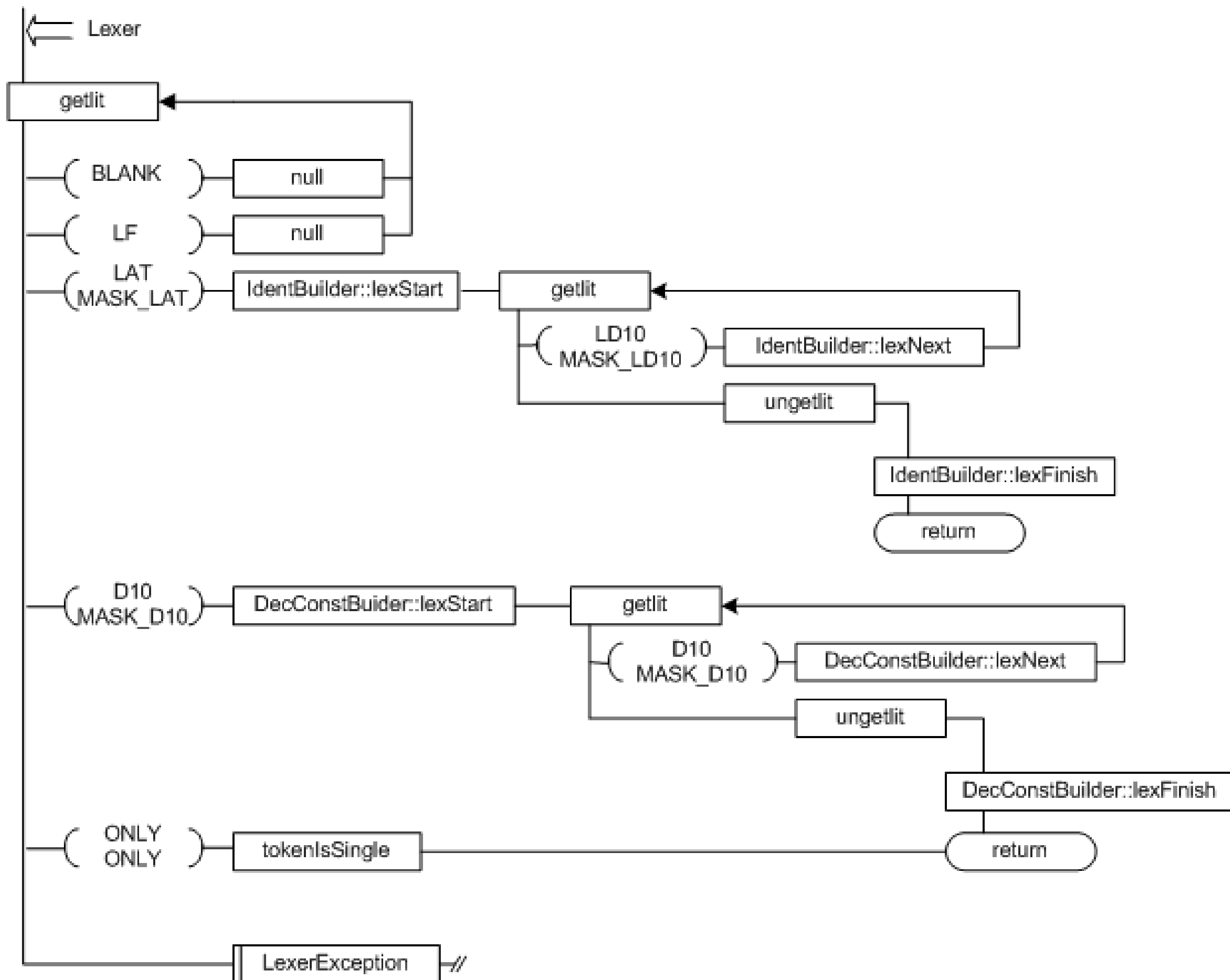
| | |
|---|---|
| 1000 0000 0010 1001 | ONLY ! ')' = RPARENT |

*Single-character synterm recognition (not mixed with LAT)*

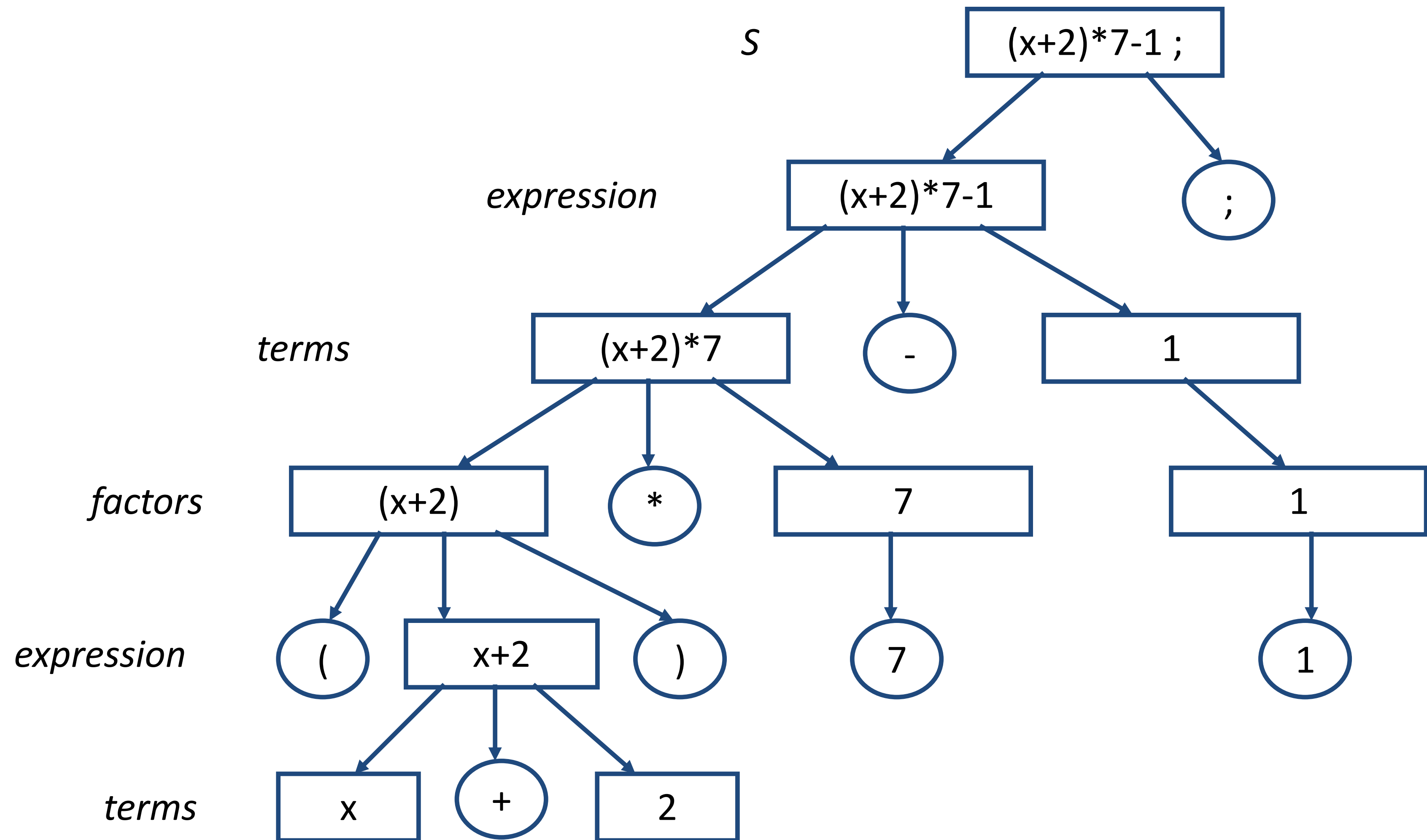
| | |
|---|---|
| 1000 0000 0010 1001 | Input character synterm |
| 1000 0000 0000 0001 | MASK_LAT |
| 1000 0000 0000 0001 | not LAT |

# Lexer Implementation




Lexer
getlit
BLANK
null
LF
null
LAT
MASK_LAT
IdentBuilder::lexStart
getlit
LD10
MASK_LD10
IdentBuilder::lexNext
ungetlit
IdentBuilder::lexFinish
return
D10
MASK_D10
DecConstBuider::lexStart
getlit
D10
MASK_D10
DecConstBuilder::lexNext
ungetlit
DecConstBuilder::lexFinish
ONLY
ONLY
tokenIsSingle
return
LexerException

# Back to the Parser: Parsing Tree



# What is a Recursive Descent Parser?

- The simples parsing algorithms
- For each non-terminal a parsing method is defined
  - The parsing method selects the grammar rule
  - Terminal symbols are being read or/and
  - the non-terminal parsing method is called
- There can be a direct or an indirect recursive method call

  ***=> That's why the parser is recursive***

# Left & Right Recursion

- Left Recursive Rules
  - Non-terminal symbol is repeated on the left

```
<non-terminal> ::=
        <non-terminal> <some>
```

- Right Recursive Rules
  - Non-terminal symbol is repeated on the right

```
<non-terminal> ::=
        <some> <non-terminal>
```

- ***Our expression grammar is right recursive***

*What does it mean for us?*

# How Parsing Methods Are Called

- Left Recursive Rules
  - Non-terminal symbol is repeated on the left

```
<non-terminal> ::=
        <non-terminal><some>
```

- *First the recursive function is called*

  *=> infinite recursion*

- Right Recursive Rules
  - Non-terminal symbol is repeated on the right

```
<non-terminal> ::=
        <some><non-terminal>
```

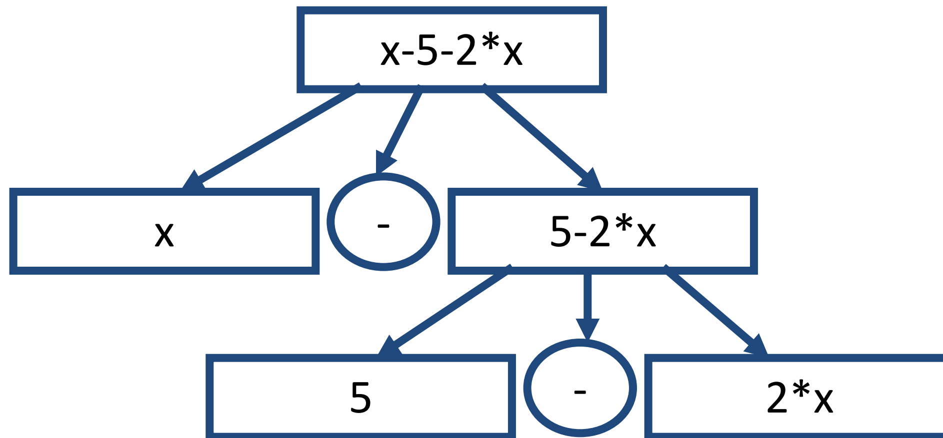
- *First we recognize the symbol (no recursion here),*

  *then the recursive function is called*

***A recursive parser works only for right recursive grammars***

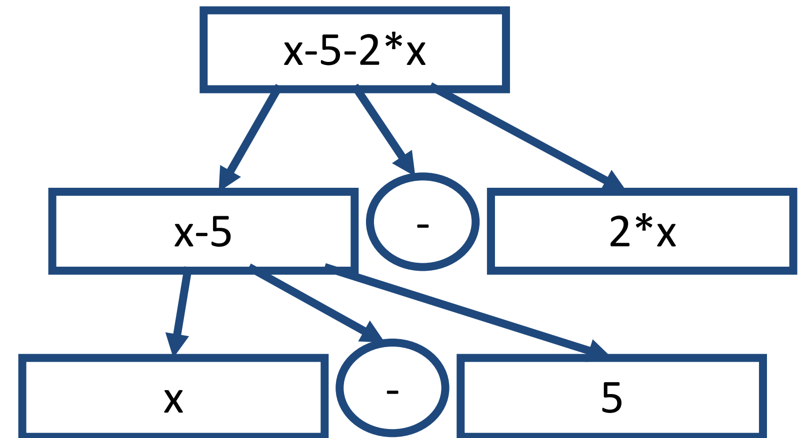
# Left & Right Recursion Trees

- Left Recursion Tree




- *Right operation order, but couldn't be parsed by a recursive parser*

- Right Recursion Tree




- *A trick:*

  ***We use right recursion grammar but an expression is read in reverse order***

# What is generated as a parsing result?

- Parsing tree
  - Represents structure – not the easiest model to use for computing parser expressions
- Polish form
  - Postfix polish form

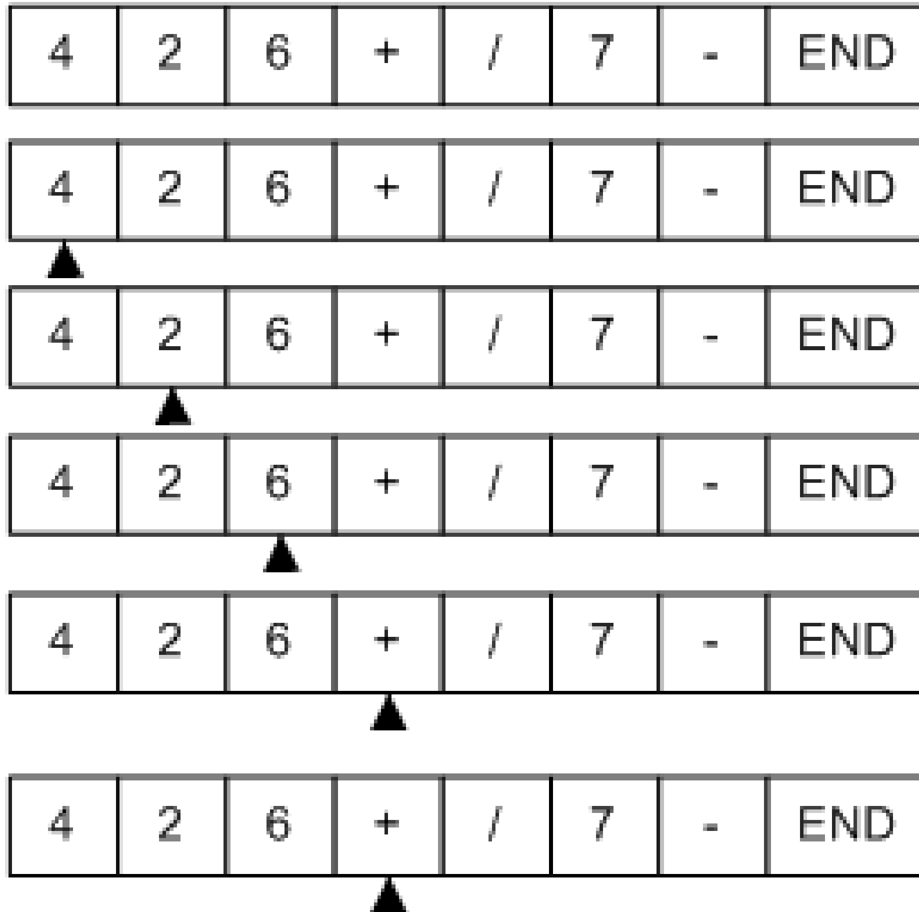
InputTokens

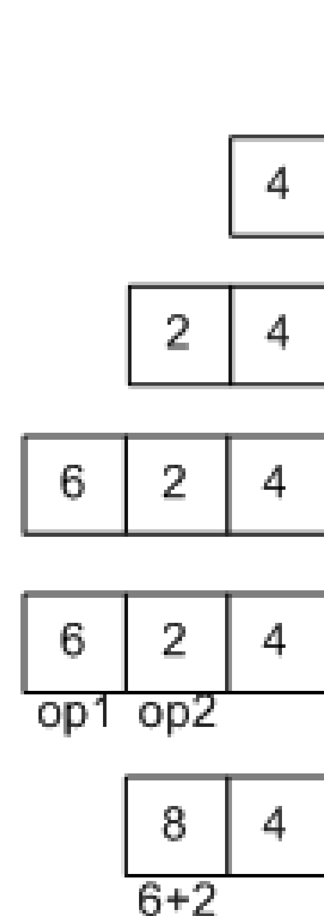
| 7 | - | ( | 6 | + | 2 | ) | / | 4 | END |
|---|---|---|---|---|---|---|---|---|---|

Polish reverse notaton

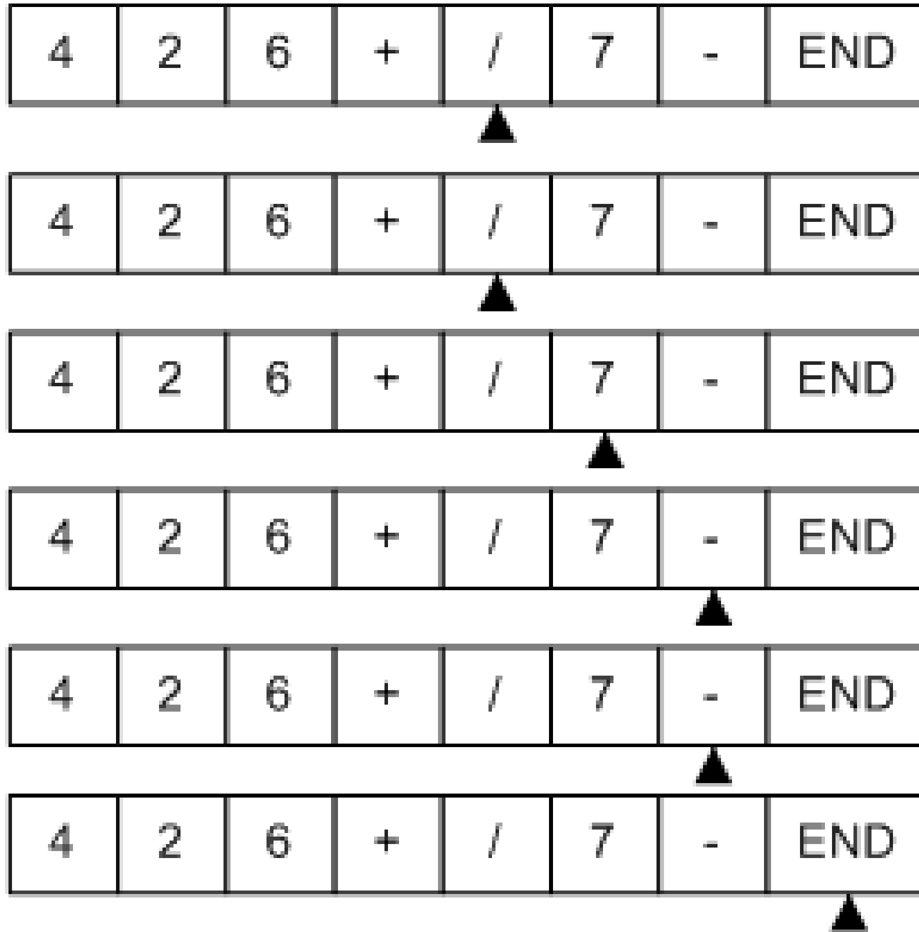
| 4 | 2 | 6 | + | / | 7 | - | END |
|---|---|---|---|---|---|---|---|

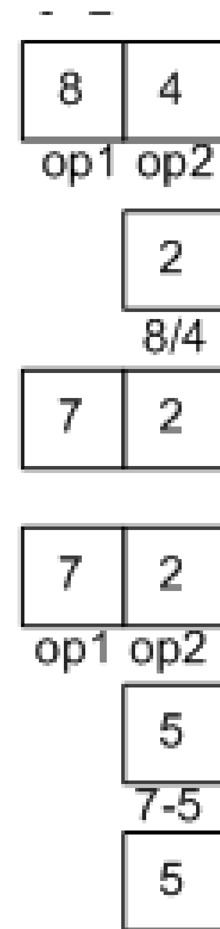
# Computing with using Polish form



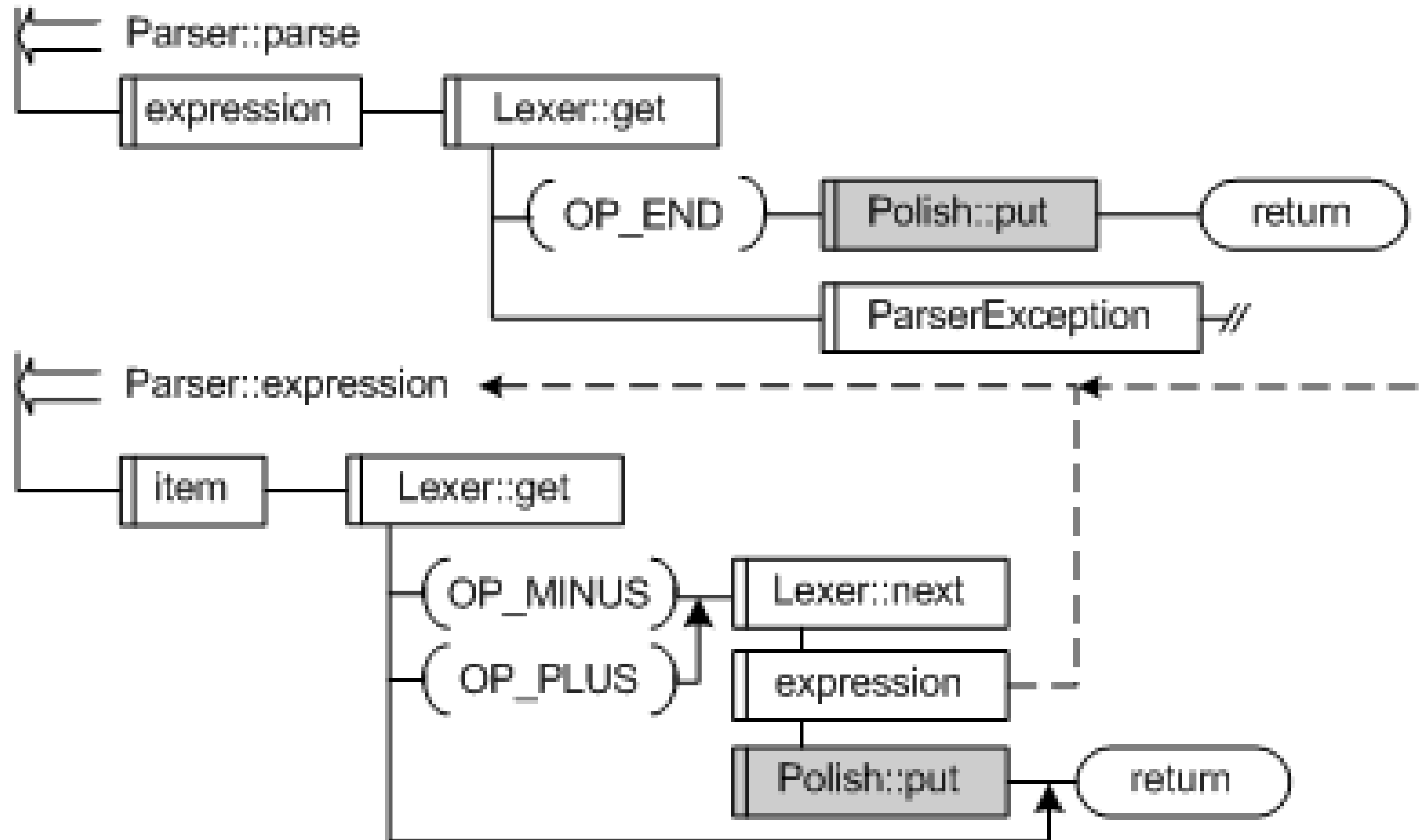



# Computing with using Polish form

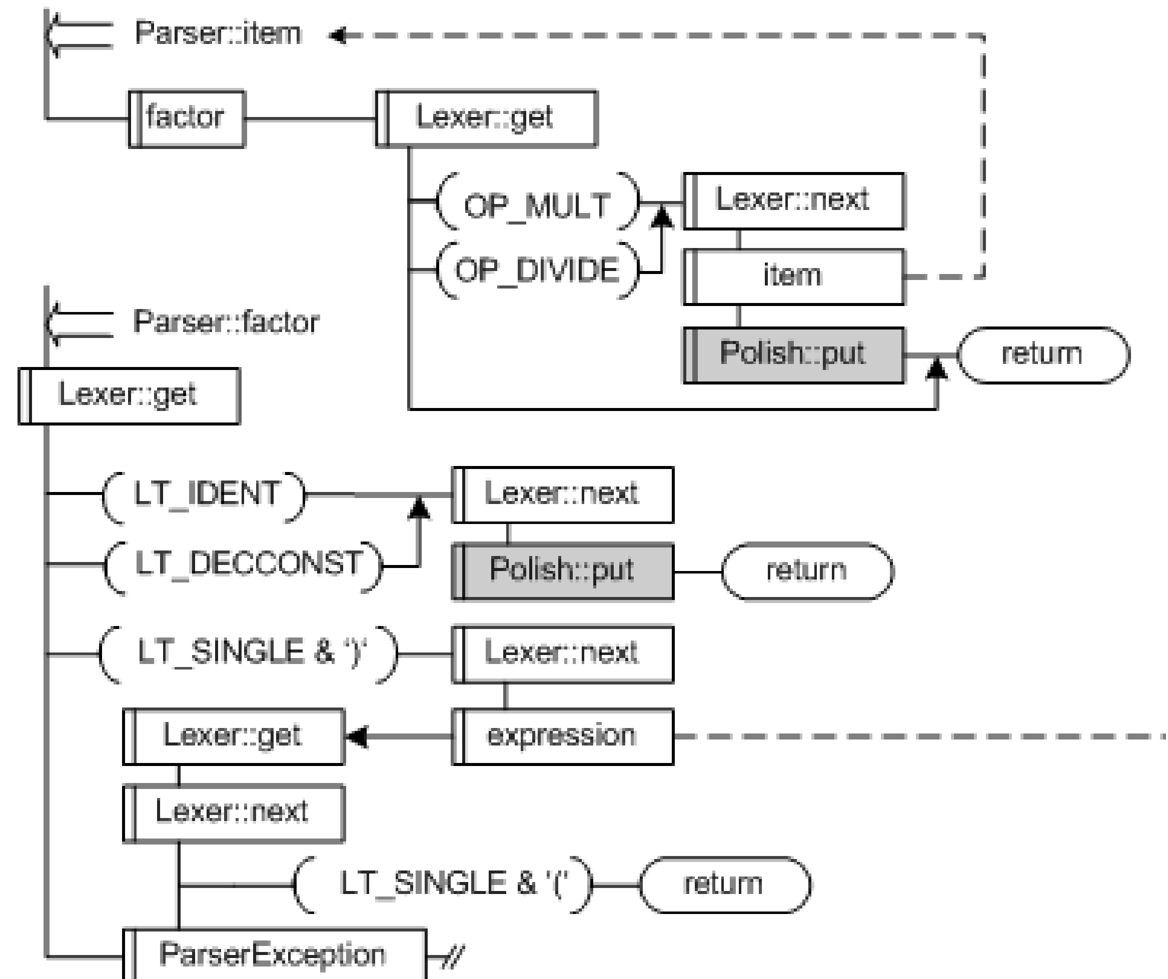


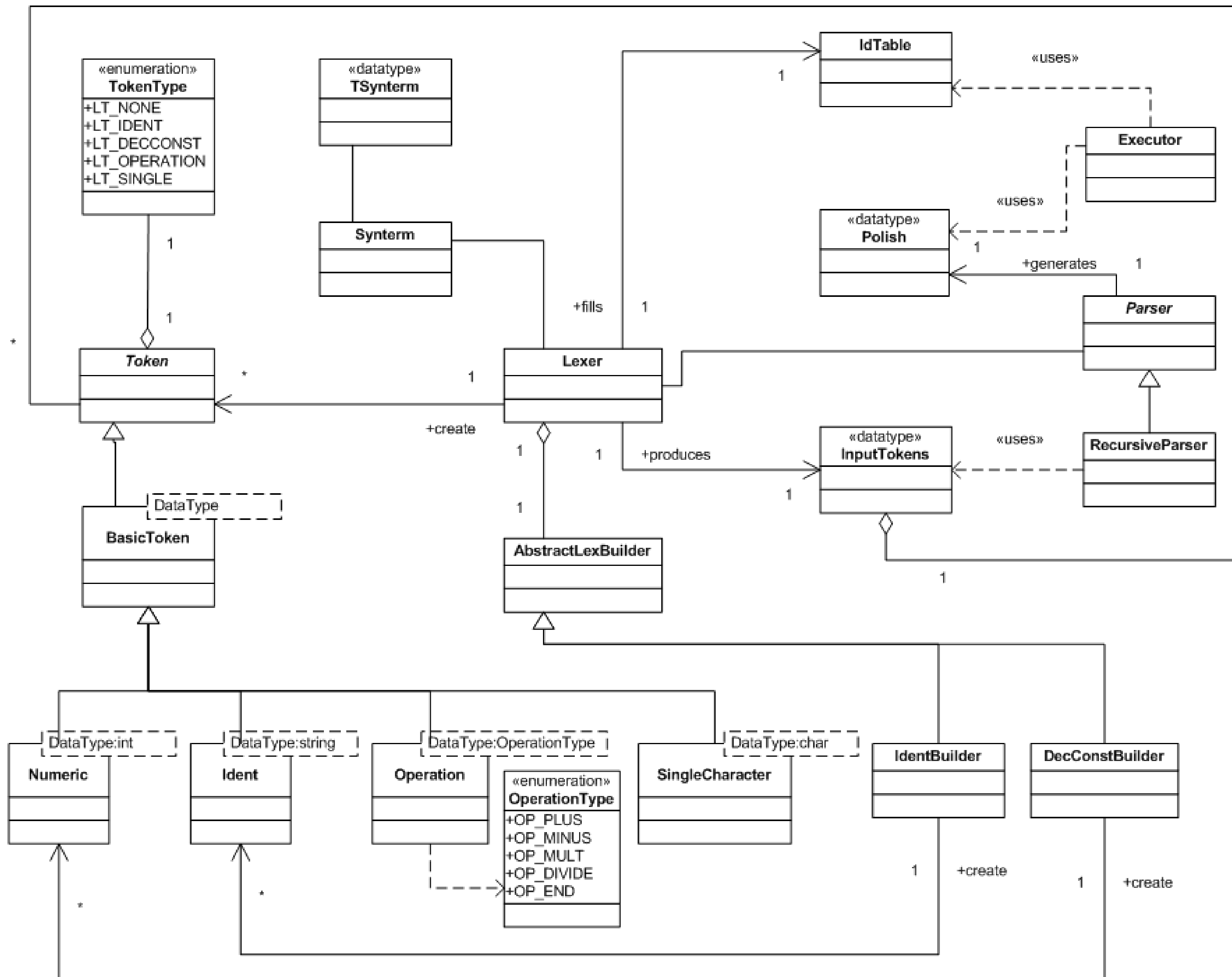
# Parser Implementation (1)



# Parser Implementation (2)



# Assembling a Jigsaw Puzzle


«enumeration»
TokenType
+LT_NONE
+LT_IDENT
+LT_DECCONST
+LT_OPERATION
+LT_SINGLE
«datatype»
TSynterm
Synterm
IdTable
«uses»
Executor
«uses»
«datatype»
Polish
+generates
Parser
+fills
Token
Lexer
+create
+produces
«datatype»
InputTokens
«uses»
RecursiveParser
DataType
BasicToken
AbstractLexBuilder
DataType:int
Numeric
DataType:string
Ident
DataType:OperationType
Operation
«enumeration»
OperationType
+OP_PLUS
+OP_MINUS
+OP_MULT
+OP_DIVIDE
+OP_END
DataType:char
SingleCharacter
IdentBuilder
DecConstBuilder
+create
+create