UCLA Extension

Engineering, Information Systems, and Technical Management (EISTM)

Advanced Object-Oriented Software Engineering with C++

Example Project: Calculator

Prepared by

Nicholas Leuci

nick@noeticode.com

Project Description: Object Model for a command-line Calculator

User Story

I want to construct a software implementation of Stroustrup's command line calculator. The first requirement is to produce a UML object model using Rational Rose / Rhapsody and then generate C++ code for the command-line calculator program.

The calculator should support the following set of features:

- portable to any operating system that has a command-line interface;
- supports basic operations of addition, subtraction, multiplication and division, negation, and the use of parentheses;
- supports compound expressions;
- allows assignments of values to variables and use of variables in expressions (i.e.: basic algebraic expression evaluation).

The calculator uses the language defined by the following CFG:

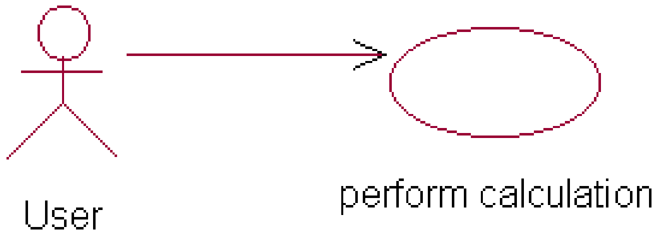
> ***program***: *END | expression_list END*
>
> ***expression_list:*** *expression PRINT | expression PRINT expression_list*
>
> ***expression:*** *term | expression + term | expression – term*
>
> ***term:*** *primary | term / primary | term * primary*
>
> ***primary:*** *NUMBER | NAME | NAME = expression | - primary | (expression)*

The user will enter expressions on the command-line, when the user hits ENTER key (denoted by PRINT symbol), the value of the expression is printed. END denotes EOF of terminal input. NUMBER is a real number and NAME starts with a letter and consists of letters and digits.

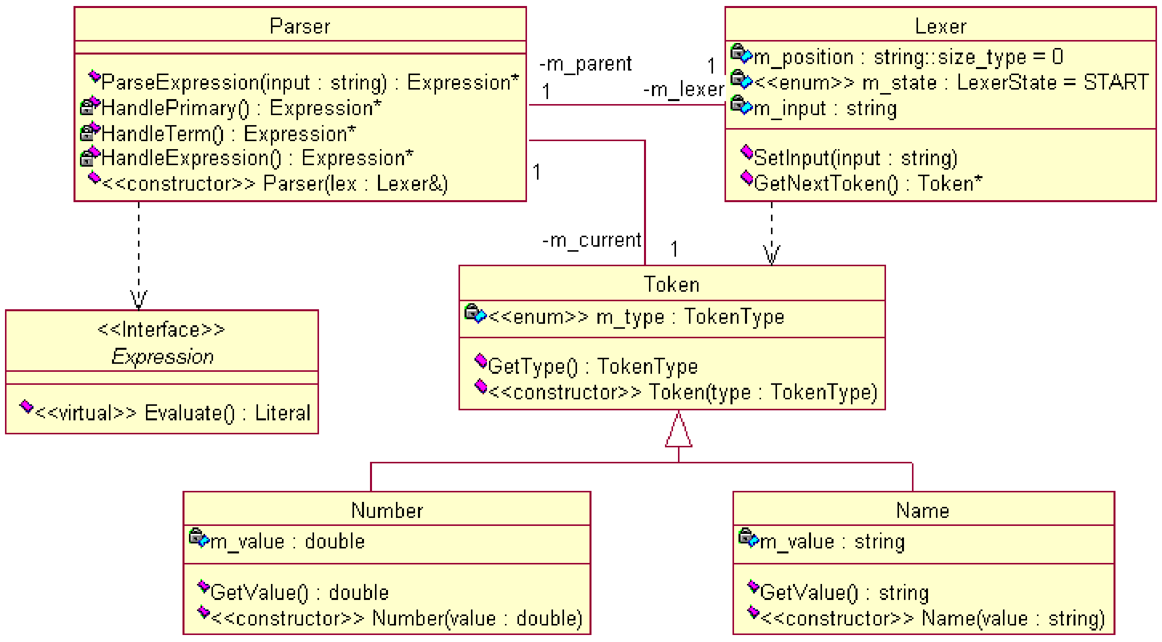
This problem idea is adapted from Bjarne Stroustrup's ***"The C++ Programming Language"*** text, from an example in which he shows a modular design and implementation of a similar command-line calculator.

I need to minimize the number of assumptions and to use a generalized approach whenever possible. Instead of simply evaluating the expression as it is being parsed, I define a hierarchy of classes supporting the **Expression** interface and then make the **Parser** build and return an expression tree (which is itself an expression). Then the driver calls **Evaluate()** on the tree and displays the result. This strategy will allow potentially extending the calculator to become a true compiler – all that is needed is to define an **Evaluate()** method for every extension class to generate code. I use a **Literal** class to represent a real number – potentially it could become a parameterized class to enable handling arbitrary data types, instead of just real numbers.

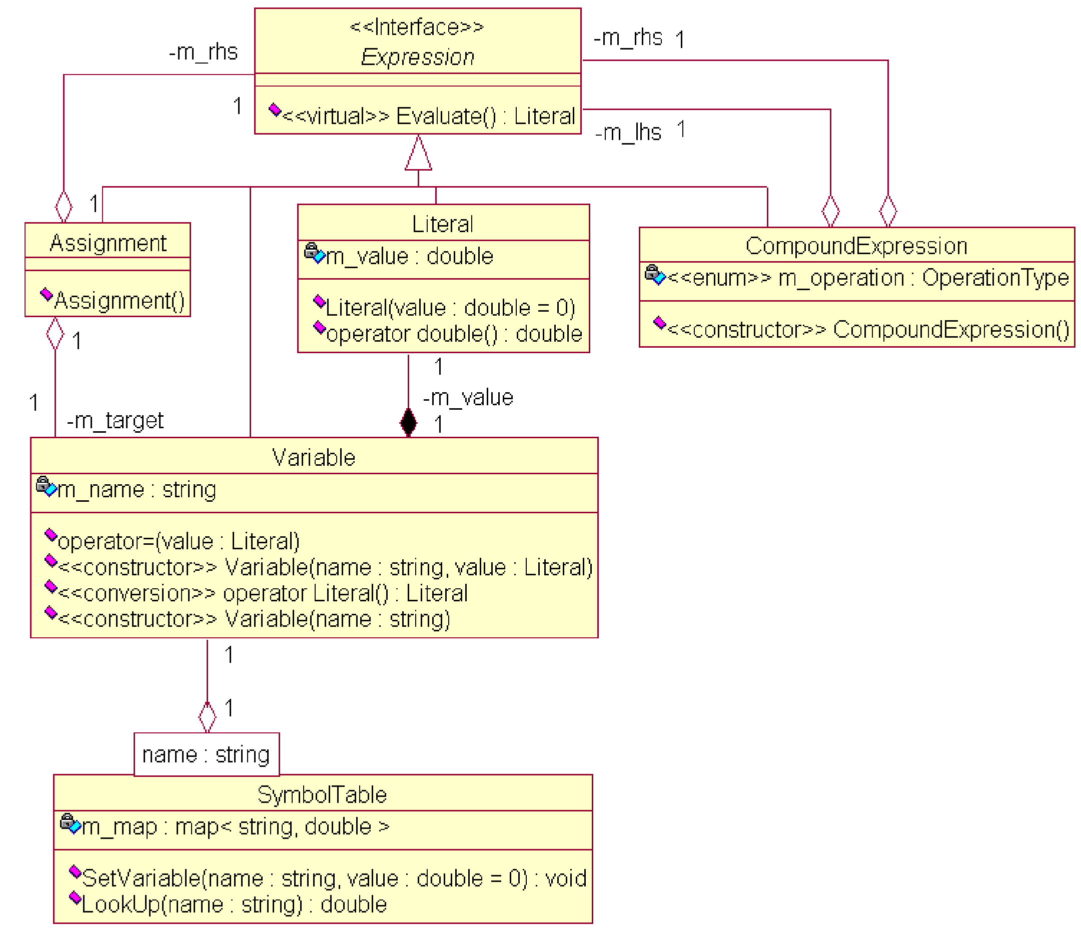
The completed project includes all object model diagrams in UML notation, code listings and sample program output. The object model diagrams are contained in a Rational Rose '.mdl' file. the C++ source files were generated by Rose, and then compiled and tested using Microsoft Visual Studio Visual C++ compiler, which generates the program executable "**calculator.exe**".



UML Calculator Use Case Diagram


Parser
ParseExpression(input : string) : Expression*
HandlePrimary() : Expression*
HandleTerm() : Expression*
HandleExpression() : Expression*
<<constructor>> Parser(lex : Lexer&)
-m_parent
1
1
-m_lexer
Lexer
m_position : string::size_type = 0
<<enum>> m_state : LexerState = START
m_input : string
SetInput(input : string)
GetNextToken() : Token*
1
-m_current
1
<<Interface>>
Expression
<<virtual>> Evaluate() : Literal
Token
<<enum>> m_type : TokenType
GetType() : TokenType
<<constructor>> Token(type : TokenType)
Number
m_value : double
GetValue() : double
<<constructor>> Number(value : double)
Name
m_value : string
GetValue() : string
<<constructor>> Name(value : string)

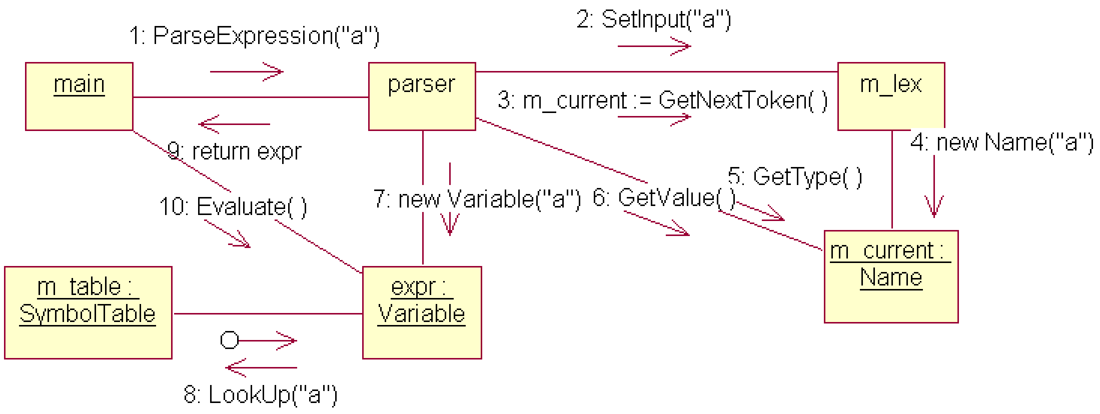
UML Calculator Class Hierarchy Class Diagram



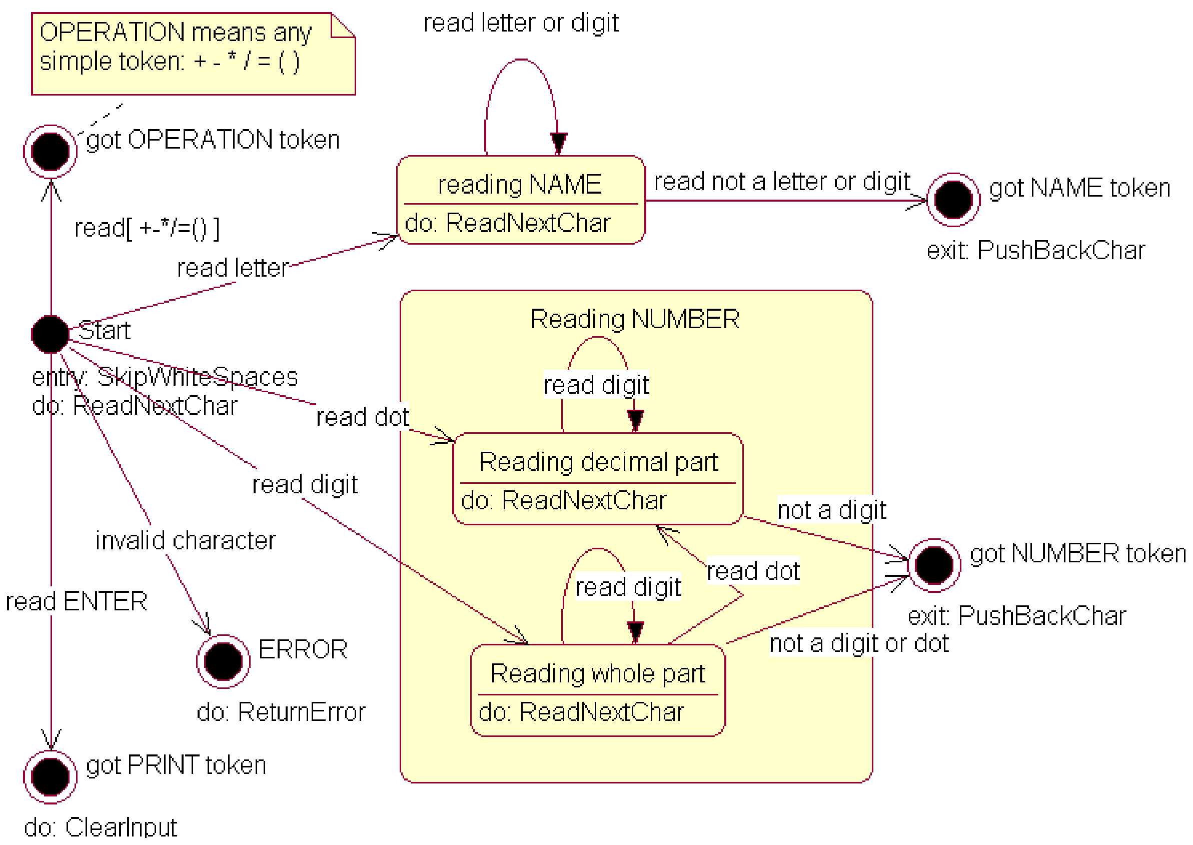
<<Interface>>
Expression
<<virtual>> Evaluate() : Literal
-m_rhs
1
-m_rhs 1
-m_lhs 1
Assignment
Assignment()
1
Literal
m_value : double
Literal(value : double = 0)
operator double() : double
CompoundExpression
<<enum>> m_operation : OperationType
<<constructor>> CompoundExpression()
1
1
-m_target
1
-m_value
1
Variable
m_name : string
operator=(value : Literal)
<<constructor>> Variable(name : string, value : Literal)
<<conversion>> operator Literal() : Literal
<<constructor>> Variable(name : string)
1
1
name : string
SymbolTable
m_map : map< string, double >
SetVariable(name : string, value : double = 0) : void
LookUp(name : string) : double

UML Expression Class Diagram

sample input: a (assume a set previously)

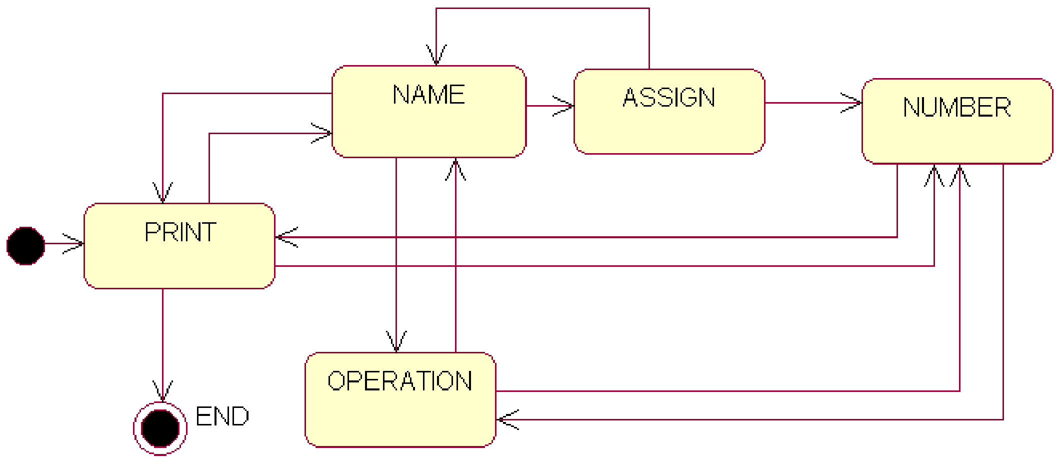


UML Group Object Colloboration (Message) Diagram



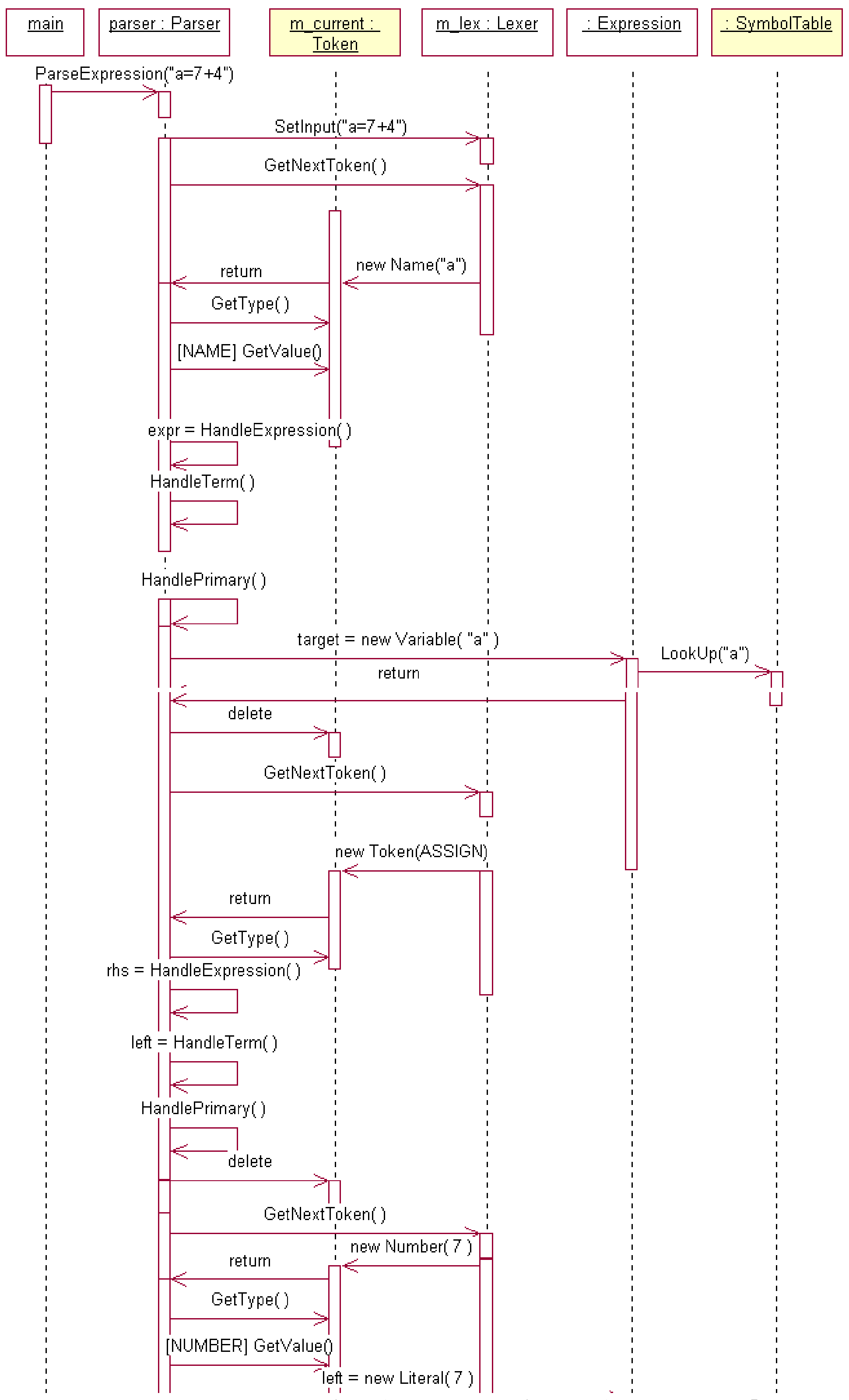
UML Lexer Class State Transition Diagram

Parser states are determined by the type of the current token
(this diagram disregards negation and parentheses)

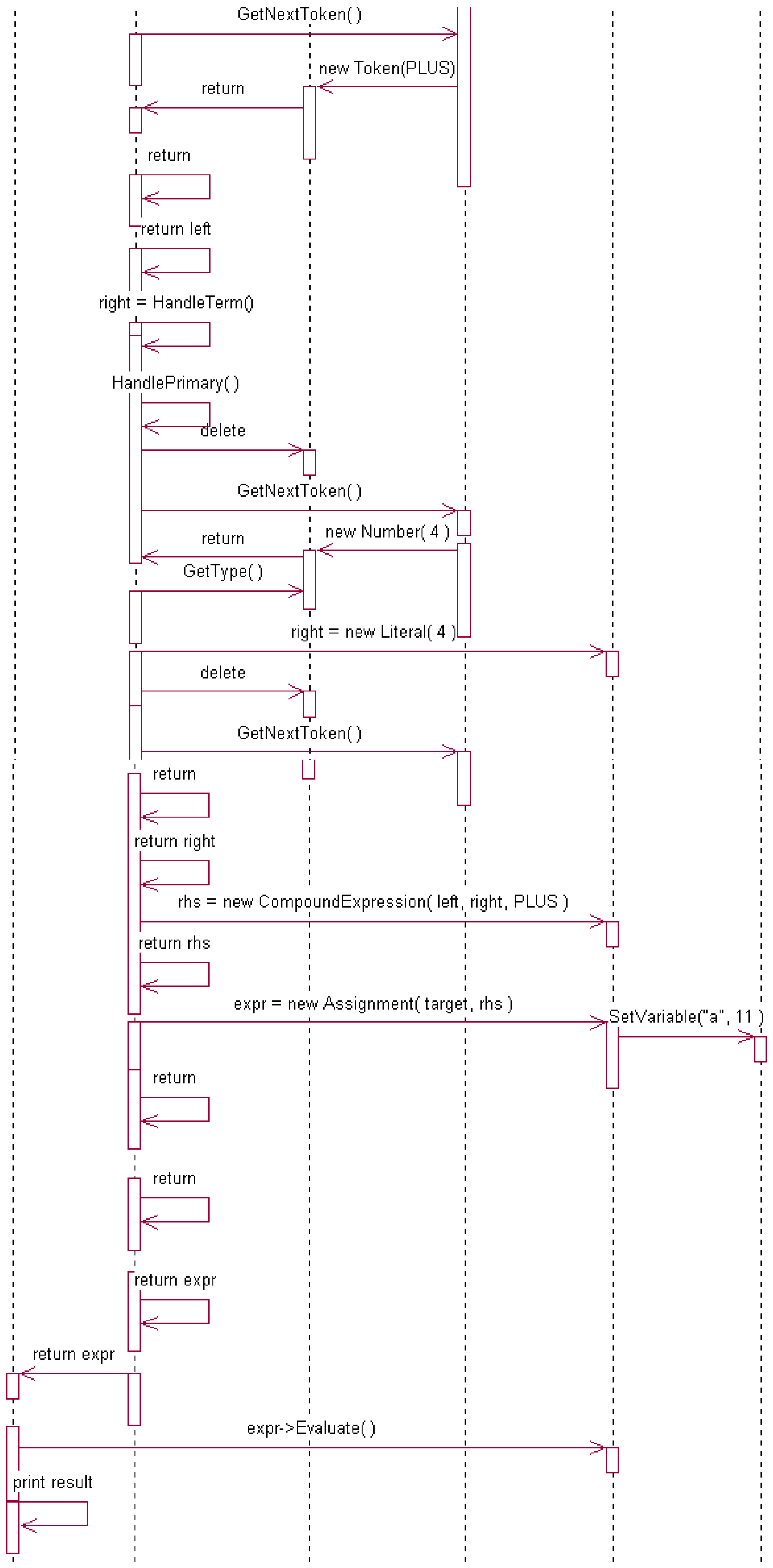


UML Parser Class State Transition Diagram

sample input: a=7+4



UML Group Interaction (Sequence Diagram), parts 1 and 2



UML Group Interaction (Sequence Diagram), parts 3 and 4