JOURNAL OF SCIENCE
SAKARYA UNIVERSITY

Sakarya University Journal of Science

ISSN 1301-4048 | e-ISSN 2147-835X | Period Bimonthly | Founded: 1997 | Publisher Sakarya University | http://www.saujs.sakarya.edu.tr/

Title: Designing and Interpreting a Mathematical Programming Language

Authors: Hüseyin Pehlivan
Recieved: 2018-12-11 06:02:48

Accepted: 2019-04-16 11:50:43

Article Type: Research Article
Volume: 23
Issue: 6
Month: December
Year: 2019
Pages: 1027-1041

How to cite
Hüseyin Pehlivan; (2019), Designing and Interpreting a Mathematical Programming Language. Sakarya University Journal of Science, 23(6), 1027-1041, DOI: 10.16984/saufenbilder.494974
Access link
http://www.saujs.sakarya.edu.tr/issue/44246/494974

Sakarya University Journal of Science 23(6), 1027-1041, 2019

Designing and Interpreting a Mathematical Programming Language

Hüseyin Pehlivan [*1]

Abstract

The syntax of the programming languages has a significant impact on the definition and validation of mathematical calculations. In particular, the management of code identification and validation processes can be made easier and faster, depending on the parametric behavior of the functions. In this article, a programming language that supports the use of mathematical function structures is designed and an interpreter, which can evaluate the source code written in this language, is developed. The language syntax is represented by an LL (k) grammar defined in the BNF notation. The interpreter consists of several basic components such as parser, semantic controller and code evaluator, each of which makes a different kind of code interpretation. The LL (k) parser component used for the syntactic analysis of the language is generated via an automatic code generation tool called JavaCC. The other components work on the abstract syntactic tree that this parser generates. To illustrate the use of the language with code samples, several mathematical algorithms that include calculations on different sequences of numbers, are programmed and interpreted. The paper also performs a comparative analysis of the language with some related ones. The paper also performs a comparative analysis of the language with some related ones based on some design principles and mathematical aspects.

Keywords: programming languages, formal grammars, parsers, interpreters

1. INTRODUCTION

Programming languages are problem solving tools that play a critical role in the development of both computer systems and computer programs which are the most important component of these systems. For programs that can be used in many different service areas such as education, health and safety, it is not only important to edit their source code but also their integration with the target computer systems. A programming language can make it possible to write programs that can easily be integrated into one or several computer systems with different architectures. The efficiency of the development processes of source code depends on design principles adopted by language developers, such as readability, writability, reliability, portability, and

[*] Corresponding Author: pehlivan@ktu.edu.tr

[1] Karadeniz Technical University, Department of Computer Engineering, Trabzon, Turkey. ORCID: 0000-0002-0672-9009

extensibility [1,2]. These principles, increasing the effectiveness and common usage of programming languages, provide an easy adaptation for a wide range of programmers. As each language has its own programming practice or style, some languages are not challenging to use this practice. For example, as in C [3] and Perl [4], very complex coding styles that are subject to international competitions can be developed [5,6].

Programming languages can be divided into two groups as general and special purpose languages. General purpose languages are designed for writing computer applications that can solve different kinds of problems with high level programming structures. They can be classified based on their fundamental features as four main programming paradigms, such as procedural languages (Pascal [7], C [3]), object-oriented languages (Java [8], C# [9]), functional languages (Scheme [10], Haskell [11]) and logical languages (Prolog [12], Ciao [13]). Special purpose languages are developed for programming tasks that require higher performance during compilation, interpretation, or runtime. In these languages, a particular class of problems can be programmed in an easier way, and system security can be increased by hiding the language runtime code. Typical programming areas include text formatting and display (TeX [14], LateX [15]), database interaction (SQL [16], AQL [17]), symbolic mathematics (Matlab [18], Mathematica [19]), hardware identification (Verilog [20], VHDL [21]) and computer games (Maya [22], Unreal Engine [23]).

Functional languages have the syntax similar to the notations of writing mathematical expressions. In these languages, the behavior of mathematical functions play an important role in the definitions made for functions. For example, the scope of the variables is limited to the function bodies and the functions always produce a return value. Since the syntax of the language does not support assignment statements, the variables are single-valued. The behavior of the functions depends only on the parameters given to them. However, some non-mathematical components of the syntax can be used in formal parameter declarations and body definitions of functions.

The literature includes many mathematical programming languages that are basically developed for mathematical modeling and optimization. Typical examples of such languages are AIMMS, AMPL, GAMS, LINGO, LPL, Mosel, MPL, OMNI, OPL and PCOMP, which are all described in [24]. In the modeling languages, the real-world problems are especially represented by mathematical models constructed with a proper set of some relationships such as equalities, inequalities and logical conditions. The main focus is on the provision of programming structures required for mathematical formulations of the problems. The syntax and semantics of these structures contain the code components such as loops and selection statements, as well as strict evaluation, which are not purely mathematical. The same is true for those in special purpose languages such as Matlab and Mathematica.

The design and development processes of programming languages require the coding of a large number of components from syntax definition to code generation in a machine language. One of these components is the language parser that performs syntax analysis. In order to generate the parser code automatically, many compiler-compiler tools (also called parser generators) such as YACC [25], SableCC [26] and JavaCC [27] are developed. Each tool usually uses a different specification file in which the syntax of a particular language is defined in a similar way to the structure of formal grammars [28]. For example, the JavaCC tool that generates the source code of a parser in Java requires the use of two different definition formats adapted from the language grammar for the words and expressions contained in the language syntax.

This paper addresses the development of a mathematical programming language, shortly called MaPL. The mathematical expressions construct the main computational structure of the MaPL language represented by a formal grammar. The interpreter implemented for the language has

several components, such as lexer, parser, semantic controller and code evaluator. The JavaCC tool is used for lexical analysis, syntax analysis and generation of abstract syntax tree (shortly called as AST). The operations of semantic analysis and code evaluation are performed through the syntax tree.

In the design of the syntax of the MaPL language, basic programming structures that meet the requirements of mathematical programming are taken into consideration. Variable declaration, sequential execution, conditional selection, repeated execution and function definition are the most important programming structures. As in functional languages, variable declarations are made without specifying the data type, and repeated calculations are performed with the help of recursive definitions which may be an alternative to the looping statements. Different statements can be executed depending on certain conditions.

2. SYNTAX

Syntax is an issue related to the expression structure of a language. MaPL is designed as a programming language with syntax supporting the notation of writing mathematical expressions. The syntax of the language allows the definition of functions in two different ways, which can be given as follows:

function(p1, p2, …, pn) = body
function(p1, p2, …, pn) : {s1, s2, …, sn} = body

In this syntax, the declarations such as x or x + k (k is an integer) can be made in the formal parameter fields represented by the elements such as p1 and p2. The elements such as s1 and s2, which are specified by opening a code block, can be assignment statements or print statements that display data on the standard output. In the body of the functions, in addition to the ordinary calculation expressions, a sequence of the pairs of expressions and conditions can appear in the following form.

{ expression, condition }

The expressions in this form correspond to the definitions of segmented functions in mathematics. Here are the examples of two functions defined in the MaPL language.

f(x) = 2 * x + 1
g(x+1,y) : { z = x + y; print(z) } = x * y + z

The MaPL language does not include the looping statements provided by the imperative programming languages. In accordance with the mathematical definitions of functions, recursive functions should be used for the calculations that must be made through a looping statement. For example, for the sum of integers between 1 and n, the following function definition can be made, consisting of two equations.

fSum(1) = 1
fSum(n) = fSum (n-1) + n

It is also possible to define in a similar way to the segmented function structure in mathematics. As the function fSum is defined mathematically like

$$f(n) = \begin{cases} 1, & n = 1 \\ f(n-1) + n, & n > 1 \end{cases}$$

the corresponding definition in the MaPL language can be made as follows (instead of the keyword "otherwise", a conditional expression like “n> 1” can also be used to represent the other cases).

fSum(n) = { 1, n == 1 }
{ fSum (n-1) + n, otherwise }

Three types of data can be used in the source file; int, double and string. There is no need to explicitly declare the data types of variables. The type inference of a variable is performed by the language itself, with the type analysis of the first expression that assigns data to that variable. For example, in the following expression, the type inference for the variable x will be made as a string.

x=2 + 3.4 + "5"

Note that the addition operations in this statement associate from left to right. Thus, the value 5.4 calculated by the first operation (i.e., 2 + 3.4) will be of double type and then the data "5.45"

calculated with the next operation (i.e., 5.4 + "5") will be of string type.

The MaPL language has a flat block structure consisting of two levels: global and local. The scope of the function definitions corresponds to the global level, which is the entire source code file. The definitions of formal parameters and local variables have a scope limited only to the body of the relevant function.

As in most programming languages, the evaluation of the source code is initiated by the invocation of the function called main. A source code file can have many syntax components, such as functions, arguments, formal parameters, and segmented function equations. In the language parser, syntax classes are defined using a data structure of linked lists to represent these components. The number of syntax components that can be contained by the source code is determined dynamically by the size of the memory the interpreter uses.

3. PARSING

The parsing process requires the design of a formal grammar for the syntax of a programming language. The JavaCC tool is used to develop the parser for the MaPL language. This section shows how to use a formal grammar in the production of the parser via JavaCC.

3.1. Grammar Design

A formal grammar is specified using a mathematical notation such as BNF and EBNF. The EBNF notation is an extended version of BNF with the addition of some meta-characters (*, +, ?, |, etc.). In these notations, a grammar can consist of one or more syntactic definitions, called rules, which govern the phrase structure of a language. The recursive nature of the rules can always generate an infinite number of possible sentences out of finite number of words. A grammar rule has left and right side definitions separated by symbols such as “=”, “:=” or “->”. The left side of the rule contains a non-terminal and the right side contains a collection of some terminals (also called token) or non-terminals. In the following grammar specifications, the terminals are written in double quotes.

In the syntax analysis, first the source data is scanned from the left to the right and broken into a sequence of terminals. Then the order of terminals in the sequence is examined by means of a grammar. The token examination can be performed from left to right (LL(k) parsing) or from bottom to top (LR(k) parsing), where k is the amount of lookahead the parser needs to select a rule. Since the JavaCC tool can produce only LL(k) parsers, the formal grammar designed for the MaPL language has to consist of a set of rules satisfying the following three properties.

- All the alternatives of a rule (the rules that contain the same non-terminal on the left side) have to produce a k number of different first tokens.
- If there is a rule that may not produce any token, this rule and the rule called after it have to produce a k number of different first tokens.
- The right side of a rule must not contain a left-recursive definition.

Considering the above properties, the LL(1) grammar developed for k = 1 is shown in Table 1, using the EBNF notation. Please note that the grammar does not provide the necessary definitions for id, num, dnum and str rules. The format of the data that is generated by these rules will be defined in the token specification block of the parser.

Table 1. Formal grammar for the MaPL language

```
program -> function ( program )?
function -> header ( block )? "=" ( expr | eqlist )
header -> id "(" ( parlist )? ")"
parlist -> param ( "," parlist )?
param -> id ("+" num)? | num
eqlist -> "{" expr "," bexpr "}" ( eqlist )?
block -> ":" "{" ( stmlist )? "}"
stmlist -> stm ( "," stmlist )?
stm -> id "=" expr
stm -> "print" "(" explist ")"
explist -> expr ( "," explist )?
```

```
expr -> ("+" | "-")? term ( ("+" | "-") term )*
term -> power ( ("*" | "/" | "%") power )*
power -> elem ( "^" power )?
elem -> id ( "(" ( explist )? ")" )? | num | dnum
elem -> str | "(" expr ")" | abs "(" expr ")"
bexpr -> and ( "||" and )*
and -> not ( "&&" not )*
not -> "!" "(" bexpr ")" | belem | "otherwise"
belem -> expr boper expr
boper -> "==" | "/=" | "<" | "<=" | ">=" | ">"
```

The grammar in Table 1 also includes the descriptions of the semantic structure of the MaPL language. The usual precedence and associativity of the operators are two semantic issues, which can be represented by the grammar rules. For example, the expr rule containing the operators "+" and "-" is invoked before the rules (i.e., term and power) containing other operators. The early invocation places these two operators in the nodes close to the root of the parsing tree), giving them a lower priority level. On the other hand, since the power rule is defined in a right-recursive way, the operator "^" contained by the rule is made right-associative.

The JavaCC tool requires three types of definitions for a programming language, such as lexical structure, syntactic structure and generating expressions of abstract syntax tree. All these definitions are stored in the same JavaCC specification file.

3.2. Lexer

The lexical structure of a language is usually defined using regular expressions. In the MaPL-language word set, in addition to 25 functional components consisting of 16 operators, 3 keywords and 6 other symbols, there is an infinite number of data components represented by the rules id, num, dnum and str in Table 1. A separate token class is defined for each of the functional components, while there is only one token class per rule for the others. In the syntax analysis, it is sufficient to make the token class definition for the types since the type of the data is more important than the value.

In Table 2, token class definitions of the MaPL language are given in the TOKEN block in accordance with the JavaCC specification format. No tokens are produced for the words covered by the definitions made within the SKIP block. The token classes whose names begin with the symbol #, such as LETTER and DIGIT, are used as part of other class definitions.

Table 2. Token definitions

```
TOKEN: {
   <PLUS: "+"> | <MINUS: "-">
  | <TIMES: "*"> | <DIVIDE: "/">
  | <MOD: "%"> | <POWER: "^">
  | <AEQ: "="> | <AND: "&&"> | <OR: "||">
  | <NOT: "!"> | <EQ: "=="> | <NE: "/=">
  | <LE: "<"> | <LT: "<="> | <GT: ">=">
  | <GE: ">"> | <COMMA: ",">
  | <COLON: ":"> | <LCURLY: "{">
  | <RCURLY: "}"> | <LPAREN: "(">
  | <RPAREN: ")"> | <ABS: "abs">
  | <PRINT: "print"> | <OTHER: "otherwise">
  | <#LETTER: ["a"-"z","A"-"Z"]>
  | <#DIGIT: [0"-"9"]>
  | <ID: <LETTER>(<LETTER> | <DIGIT>)*>
  | <NUM: (<DIGIT>)+>
  | <DNUM: (<DIGIT>)+"."(<DIGIT>)+>
  | <STR: ("\"" (~["\""] | "\\" "\"")* "\"")>
}
SKIP: { " " | "\t" | "\r" | "\n" }
```

For example, according to the definitions given in Table 2, the sequence of the tokens produced for the statement print (x, y + 1) will be as follows.

PRINT LPAREN ID COMMA ID PLUS NUM RPAREN

This token sequence indicates that the specification order in Table 2 is important. The lexer evaluates all token class definitions from top to bottom, for each word it scans in the source data, and selects the first one matching the word and produces the corresponding token. For example, there are two possible token classes that match the word "print"; PRINT and ID. However, since the print is a keyword in the MaPL language, the PRINT token must be produced. Therefore, the

PRINT definition is placed in one of the lines before the ID definition, with the aim of producing the correct token.

3.3. Parser

The parsers that are used to analyze the syntactic structure of a programming language can be developed by hand or by means of an automated code generating tool like JavaCC. With these tools, the source code of a parser is generated from the method definitions based on the grammar rules of the language. The language of the generated code varies depending on the used tool. For example, the JavaCC tool generates the parser code in the Java programming language.

With the two main deterministic parsing methods called LL(k) and LR(k), the parsing steps are managed by the grammar representing the syntax of the language. An LL(k) parser that performs a leftmost derivation of the code can be configured according to the structure of the grammar rules. It is possible to develop an LL(1) parser for the formal grammar given in Table 1, where a rule can be selected only with one token (i.e., k = 1). In this way, some method definitions are added to the parser to represent each grammar rule, keeping them in the order that the rules call each other in the grammar. Table 3 shows some of typical method definitions for an LL(1) parser using the JavaCC specification format.

Table 3. Parser methods

```
void start() : { } {program() <EOF>}
void program() : { } {function() (program())?}
void function() : { }
  {header() (block())? <AEQ> (expr()|eqlist())}
void header() : { }
  {<ID> <LPAREN> ( parlist )? <RPAREN>}
void parlist() : { }
  { param() (<COMMA> parlist() )? }
void param() : { }
  { <ID> (<PLUS> <NUM> )? | <NUM> }
void eqlist() : { }
  { <LCURLY> expr() <COMMA>
          bexpr() <RCURLY> ( eqlist() )? }
void block() : { }
  { <COLON> <LCURLY>
          ( stmlist() )? <RCURLY> }
void stmlist() : { }
  { stm() ( <COMMA> stmlist() )? }
void stm() : { }
  { <ID> <AEQ> expr()
  | <PRINT> <LPAREN>
          explist() <RPAREN> }
void explist() : { }
  { expr() (<COMMA> explist() )? }
```

As shown in Table 3, start() is the starting method of the parser. A special JavaCC token <EOF> is used to mark the end of source data that can be entered from the standard input or read from a file. The parsing process continues to perform on the source data until this token is encountered.

3.4. Syntax Classes

Syntax classes serve to build a tree-based representation of source data with object-oriented programming structures. The grammar rules of the language have an important role in what syntax classes must be defined. In general, a syntax class is defined for each grammar rule that contains an operator or keyword. The definition of syntax classes representing the rules with the alternatives is made by inheriting from the same super class and thus they can serve as alternatives to each other in a similar way to the rules.

In some cases, a single class is created by combining some of the grammar rules (especially those that are complementary to a certain expression or statement, or invoked by another rule). For example, in the case that a programming language expression is defined by more than one grammar rule, it is adequate to define one syntax class to represent all of these rules. In the grammar in Table 1, the syntax classes for the <block>, <expr>, and <eqlist> rules, which are all invoked by the <function> rule, are not defined.

A syntax class can be given the name of the related grammar rule represented or another name associated with the expression generated by that

rule. The fields of the class are defined by the data types appropriate to the terminals and non-terminals contained in the relevant rule. The classes Var, Num, DNum and Str, for which a rule is not defined, are constructed for id, num, dnum and str non-terminals, respectively. Table 4 shows some of the syntax classes.

Table 4. Syntax classes

```
class Program {
  Function def; Program prog;
  public Program(Function x, Program y)
  { def = x;  prog = y; }
}
class Function {
  Header f;  Stm s;  EList eq;
  public Function(Header x, Stm y, EList z)
  { f = x;  s = y;  eq = z; }
}
class EList {
  Exp e;  BExp b;  EList eq;
  public EList(Exp x, BExp y, EList z)
  { e = x;  b = y;  eq = z; }
}
class Stm { }
class AStm extends Stm {
  String id;  Exp e;
  public AStm(String x, Exp y) {id = x;  e = y;}
}
class PStm extends Stm {
  EList eq;
  public PStm(EList x) { eq = x; }
}
class Exp { }
class Header extends Exp {
  String id; EList eq;
  public Header(String x, EList y)
  { id = x;  eq = y; }
}
```

Although, in most programming languages, the definition expression of a function has a different format or syntax from the invocation expression, both expressions can have the same syntax in the MaPL language. Therefore, the object reference of the Header type added to the Function class will be used to represent both the function definition and calling expressions.

As seen in Table 4, a constructor that is used to create an object of the related class is provided for all the syntax classes. In addition, in the phase of interpreting the source data, a method called accept() for the use of the Visitor interface, described in Section 4, must be defined in the syntax classes as follows.

```
public ..... accept(Visitor v) {
    return v.visit(this);
}
```

The type of the return value for this method should be void for the classes such as Stm, AStm and PStm, and Object for the ones such as Exp, Header and Plus.

3.5. Abstract Syntax Tree

A syntax tree is created as a representation of the source data in the form of a tree data structure by connecting objects derived from syntax classes. The functional components (operators, keywords, etc.) of the source code form the intermediate nodes of the tree, while the data components (constants, variables, etc.) do the leaves. In this way, each node of the tree can contain objects of a different syntax class, depending on the type of the related code component.

Syntax trees are used to perform operations such as type control and code interpretation that are difficult to perform over source data. As in the syntax analysis of the source data, the JavaCC tool can be used in the production of syntax trees. For this purpose, the expressions in Java that will generate the data required for the corresponding node of the syntax tree are added to the special code blocks that are opened in the body of the parser methods. The production of the tree is provided via these expressions that are performed simultaneously with the syntax analysis. In Table 5, some parser methods are shown together with the code blocks added for the generation of the syntax tree.

Table 5. Parser methods that generate abstract syntax tree

```
Program start() : { Program prog; }
  { prog=program() <EOF> { return prog; } }
Program program() :
  { Function def; Program prog = null; }
  { def=function() ( prog=program() )?
    { return new Program(def, prog); } }
Function function() :
  { Header fn; Stm s=null; EList eq; Exp e; }
  { fn=header() ( s=block() )? <AEQ>
   ( e=expr()
    {eq = new EList(e, new BNum(true), null);}
     | eq=eqlist() )
    { return new Function(fn, s, eq); } }
Header header() :
  {Token t; EList eq = null; boolean b = false;}
  { t=<ID> <LPAREN>
       ( eq=parlist() )? <RPAREN>
    { return new Header(t.image, eq); } }
EList parlist() :{ EList eq = null; Exp e; }
  { e=param() ( <COMMA> eq=parlist() )?
    { return new EList(e, null, eq); } }
Exp param() : { Token t, t2; Exp e; }
  { t=<ID> ( <PLUS> t2=<NUM>
     return new Plus(new Var(t.image),
     new Num(Integer.parseInt(t2.image))); } )?
     { return new Var(t.image); }
   | t=<NUM> { return new
            Num(Integer.parseInt(t.image)); } }
EList eqlist() :
  { Exp e; BExp b; EList eq = null; }
  { <LCURLY> e=expr() <COMMA>
    b=bexpr() <RCURLY> ( eq=eqlist() )?
    { return new EList(e, b, eq); } }
Stm block() : { Stm s=null; }
  { <COLON> <LCURLY> ( s=stmlist() )?
    <RCURLY> { return s; } }
```

Since the syntax analysis starts with a call to the start() method, the root node of the syntax tree always has an object of the Program type. Other methods called during the analysis produce different types of objects. For example, using the class definitions in Table 4 and the method definitions in Table 5, the syntax tree produced by the parser for the expression print (x, y + 1) would be as follows.

```
PStm p = new PStm(new EList(new Var("x"),
null,
                new EList(new Plus(new Var("y"),
                  new Num(1)), null, null)));
```

The EList class, defined as a linked list, is used in object construction through several parser methods. The list of function formal parameters, the argument lists of both the print function and user-defined functions, and the list of segmented functions are created as an object of the EList class. The related list element is given as the first argument to this class constructor, the other argument is either the condition expression of the segmented function list or null for the other lists.

4. EVALUATION

The two components, the semantic controller and the code evaluator, need to interpret the source code, using the syntax tree. The implementation of these components that interpret the AST data in different forms is based on the Visitor design pattern, which is one of the behavioral design patterns [29].

4.1. Visitor Interface

The Visitor interface, which provides the type information of visit() methods, corresponds to one of the two components of the Visitor design pattern. The other component is the Accept interface and involves the definition of accept() methods within all syntax classes. The visit() and accept() methods together constitute a double dispatch mechanism that is used to evaluate the objects found in the AST data. Table 6 lists some of the methods that are described in the Visitor interface.

Table 1. Visitor interface

```
interface Visitor {
  public Object visit(Exp e);
  public Object visit(Fn e);
  public Object visit(Plus e);
```

```
    public Object visit(Minus e);
    public Object visit(Times e);
    public Object visit(Divide e);
    public Object visit(Mod e);
    public Object visit(Power e);
    public Object visit(Var e);
    public Object visit(Num e);
    public Object visit(DNum e);
    public Object visit(Str e);
    public void visit(Stm s);
    public void visit(AStm s);
    public void visit(PStm s);
 …….
 }
```

For some method declarations in Table 6, the type of the return value is specified as Object. This kind of method declarations allows the use of the same Visitor interface for different evaluation requirements (type control, interpretation, etc.) of the abstract syntax tree.

4.2. Semantic Controller

For type control operations performed with the semantic controller, a symbol table is created using the definitions in the source code. All formal parameters and local variables defined in the body of the functions are added as symbols to this table. There are some important cases to take into account when an entry is added to the symbol table, which are given below.

- The formal parameters of a function must have unique names.
- The type of a formal parameter is inferred from the expression that uses the parameter.
- The names of local variables must be different from formal parameter names.
- The type of a local variable is inferred from the expression that initializes its first value.

Table 7 shows the SymTable class defined to represent the symbol table. During the analysis of a function definition, a new symbol block is created in the symbol table, and all symbols and their types in the related definition are stored in this block. The symbol block is only accessible through the analysis of the function and is released when the analysis is finished.

Table 7. SymTable class

```
class SymTable {
  int size;
  int index = -1;
  Hashtable[] table;
  public SymTable(int s) {
    size = s;
    table = new Hashtable[s];
  }
  public int beginScope() {
    ++ index;
    if (index >= table.length)
      return -1;
    table[index] = new Hashtable();
    return 0;
  }
  public void endScope() {
    -- index;
  }
  public void put(String id, Object obj) {
    if (obj == null)
      return ;
    table[index].put(id, obj);
  }
  public Object get(String id) {
    return table[index].get(id);
  }
}
```

The semantic analysis of the source code focuses on the syntax tree created by the parser. First, traversing all nodes of the tree, the symbol definitions are determined. For each symbol defined in the source code, a pair (name, type) is stored in the symbol table. For example, the pair ("x", new Num (0)) for a variable x of type int and the pair ("y", new Str ("")) for a variable y of type string are added to the symbol table. When the use of a symbol is encountered during the traversal, the symbol table is looked up and their types are obtained. The following cases are considered for the type inspection performed through the symbol table.

- The data type, order and number of formal parameters as well as the type of returned data must be the same in all equations of a function.
- The actual parameters passed to a function must match the formal parameters in type, order and number.
- The type of a formal parameter or a local variable must be compatible with the use in all expressions, or with the type of value re-assigned to it.
- All local variables must be initialized before being used.
- Each called function must have a definition.
- All operators must be applied to the correct type of data.

The TypeVisitor class, which is implemented from the Visitor interface, is defined to represent the semantic controller. Table 8 shows a part of the TypeVisitor class with the visit() methods defined on some syntax classes.

Table 8. TypeVisitor class

```
Class TypeVisitor implements Visitor {
  Program p;
  SymTable t;
  public TypeVisitor(Program a, SymTable b)
    { p = a;  t = b; }
  public void visit(Stm s) {
    s.accept(this);
  }
  public void visit(LStm s) {
    s.a.accept(this);
    s.b.accept(this);
  }
  public void visit(AStm s) {
    Object a = s.a.accept(this);
    if (a != null) {
      Object b = t.get(s.id);
      if (b == null)
        t.put(s.id, a);
      else if ((a instanceof Num &&
                !(b instanceof Num)) ||
               (a instanceof DNum &&
                !(b instanceof DNum)) ||
               (a instanceof Str &&
                !(b instanceof Str)))
        System.out.println("Consistency error: "
        + s.id + "=" + new PrintVisitor().visit(s.a));
      }
      else
        System.out.println("Inference error: " +
          s.id + "=" + new PrintVisitor().visit(s.a));
    }
    public Object visit(Var e) {
      return t.get(e.id);
    }
    public Object visit(Num e) {
      return new Num(0);
    }
  }
```

In the code fragment shown in Table 8, there is a visit() method defined to analyze a tree node of type AStm that represents an assignment statement. This method checks the compatibility of the type of the variable on the left side of the statement with the type of data calculated by the expression on the right side. If the type cannot be inferred from the expression or the type inconsistency is encountered, then the PrintVisitor class, which is also implemented from the Visitor interface, is used to indicate the relevant expression of the source code.

4.3. Evaluator

The symbol table represented by the SymTable class in the previous section is also used to evaluate the source code. The evaluation process traverses the nodes of the syntax tree and stores the pairs (name, value) for the values assigned to formal parameters or local variables in this table. For example, the pair ("x", new DNum (3.0)) for the expression x = 3.0 is added to the symbol table. If this expression is followed by another expression y = x + 1, then the value of the variable x is queried from the symbol table and a new pair ("y", new DNum (4.0)) is added to the table.

The code evaluator component is represented by the EvalVisitor class implemented from the Visitor

interface. Table 9 shows the EvalVisitor class with some visit() methods.

Table 9. EvalVisitor class

```
class EvalVisitor implements Visitor {
  Program p;
  SymTable t;
  public EvalVisitor(Program a, SymTable b)
    { p = a;  t = b; }
  public void visit(Stm s) {
    s.accept(this);
  }
  public void visit(LStm s) {
    s.a.accept(this);
    s.b.accept(this);
  }
  public void visit(AStm s) {
    t.put(s.id, s.a.accept(this));
  }
  public Object visit(Var e) {
    return t.get(e.id);
  }
  public Object visit(Num e) {
    return new Integer(e.n);
  }
}
```

In all classes implementing the Visitor interface, the data passing between the visit() methods is performed through the objects of type Object. So, in order to be able to use the value wrapped in an object in the calculations, it needs to be converted from the Object type to the primitive data type. For example, given the statement obj=new Integer(x) wrapping the integer x with the object of type Integer, the conversion from the wrapping object to the correct type is performed by the statement x=((Integer)obj).intValue().

5. INTERPRETATION

This section describes the integration of the interpreter components and some examples of programs developed in the MaPL language.

5.1. Integration of Components

The interpreter of the MaPL language is constructed by integrating the Parser, Semantic Controller and Evaluator components introduced in the previous sections. The main() method of the interpreter contained in the Interpreter class is coded as shown in Table 10.

Table 2. Interpreter class

```
public class Interpreter {
 public static void main(String[] args) {
  try {
   Program p =
     new Parser(System.in).Prog();
   SymTable t = new SymTable(10000);
   TypeVisitor type = new TypeVisitor(p, t);
   Object res = type.visit(new Fn("main", null));
   if (res != null) {
     EvalVisitor eval = new EvalVisitor(p, t);
     eval.visit(new Fn("main", null));
   }
  }
  catch(ParseException ex) {
     System.out.println(ex.getMessage());
  }
 }
}
```

There are some source files that the interpreter components depend on. The parser is composed of the JavaCC specifications and the syntax classes for which the files Parser.jj and AST.java are developed, respectively. The type controller is stored in the file TypeVisitor.java, and the code evaluator in the file EvalVisitor.java. Using these files placed in the same directory, the interpreter can be produced as follows.

```
$>  javacc Parser.jj
$>  javac *.java
```

The interpretation of a program written in the MaPL language, which is held in a source file called prog.txt, is started by the following command

```
$>  java Interpreter < prog.txt
```

5.2. Program Examples

To illustrate the syntax of the MaPL language, five examples of programs that generate some popular sequence of numbers are given below.

Program 1: Table 11 shows the source code of a program that finds all the factors of the number 100 and displays it via the show() function.

Table 11. Program that finds the factors of 100

```
main() = factor(100, 1)
factor(n, k) =
    { 0, k > n }
    { show(n, k), n % k ==0 }
    { factor(n, k+1), otherwise }
show(n, k) : { print(k + " ") } = factor(n, k+1)
```

Output 1: The output of the program is as follows:

1 2 4 5 10 20 25 50 100

Program 2: The program in Table 12 shows the prime numbers smaller than 40 in the ascending order; the mod() function checks the divisibility of the number n with the integers k less than the number $\sqrt{n}$.

Table 3. Program that finds the primes up to 40

```
main() = prime(40, 2)
prime(n, k) =
    { 0, k >= n }
    { show(n, k), mod(k, 2)>0 }
    { prime(n, k+1), otherwise }
mod(n, k) =
    { 1, k*k > n }
    { 0, n % k == 0 }
    { mod(n, k+1), otherwise }
show(n, k) : { print(k + " ") } = prime(n, k+1)
```

Output 2: The output of the program is as follows:

2 3 5 7 11 13 17 19 23 29 31 37

Program 3: The program given in Table 13 shows the factorials of positive integers not greater than 5; The fact() function is written in accordance with the mathematical definition. The print() function with no argument writes a new line to the standard output.

Table 43. Program that shows the factorials of 0 to 5

```
main() = show(5, 0)
show(n,k) : { print((k) + "!=" + fact(k)), print()
} =
    { 0, k >= n }
    { show(n, k+1), otherwise }
fact(0) = 1
fact(n) = n * fact(n-1)
```

Output 3: The output of the program is as follows:

0!=1
1!=1
2!=2
3!=6
4!=24
5!=120

Program 4: The program in Table 14 shows the first 15 elements of the fibonacci number sequence, where the fib() function is written in accordance with the mathematical definition.

Table 54. Program that shows the 15 fibonacci numbers

```
main() = show(14, 0)
show(n, k) : { print(fib(k) + " ") } =
  { 0, k >= n }
  { show(n, k+1), otherwise }
fib(0) = 0
fib(1) = 1
fib(k) = fib(k-1) + fib(k-2)
```

Output 4: The output of the program is as follows:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

Program 5: The following program in Table 15 shows the first 10 elements of the Catalan number sequence, where the cat() function is written from the mathematical definition. The current syntax of the MaPL language does not allow the use of the

summation symbol (∑), which represents the sum of number sequences.

Table 65. Program that shows the 10 Catalan numbers

```
main() = show(9,0)
show(n, k) : { print(cat(k, 0) + " ") } =
    { 0, k >= n }
    { show(n, k+1), otherwise }
cat(n+1, i) =
    { 1, n < 0 }
    { 0, i > n }
    { cat(i, 0) * cat(n-i, 0) + cat(n+1, i+1),
      otherwise }
```

Output 5: The output of the program is as follows:

1 1 2 5 14 42 132 429 1430 4862

6. COMPARATIVE ANALYSIS

There are several design principles (also known as language metrics) that are introduced for guiding the evaluation of programming languages [30]. For a comparative analysis of the MaPL language, selecting the languages C, C++, Java, Haskell and Matlab, we perform the evaluation based on the metrics of efficiency, extensibility, maintainability, portability, orthogonality, readability, reliability, uniformity and writability. The results are presented in Table 16.

Table 76. Comparison of selected languages via some metrics

Metrics	MaPL	C	C++	Java	Haskell	Matlab
Efficiency	X	X	X	X	X	X
Extensibility		X	X	X	X	X
Maintainability	X	X	X	X	X	X
Orthogonality	X				X	
Portability	X			X		X
Readability	X	X	X	X	X	X
Reliability	X	X	X	X	X	X
Uniformity	X			X	X	
Writability	X	X	X	X	X	X

Another comparison is made for some mathematical languages from the perspective of programming aspects that can support mathematical reasoning. These aspects are generally related to the syntax and semantics of a language; the ones such as referential transparency, non-strict semantics, assignments, recursion, pattern-matching are purely mathematical, as the others such as type annotations, loops, selection statements and side-effects are not. The evaluation focuses on four mathematical languages (i.e., OPL, LPL, AMPL and CMPL) as well as MaPL, considering mathematics as another language called Math. Table 17 shows the analysis results.

Table 87. The aspects of some mathematical languages

Aspect	Math	MaPL	OPL	AMPL	LPL	CMPL
assignments	X	X	X	X	X	X
loops			X	X	X	X
non-strict semantics	X	X				
pattern-matching	X	X	X			
recursion	X	X	X	X		
referential transparency	X	X				
segmented functions	X	X	X		X	
selection statements			X	X	X	X

side-effects	X	X	X	X
type annotations	X	X		X

7. CONCLUSIONS

This study describes the development and interpretation of a programming language (called MaLP) based on the syntax notation of mathematical expressions. The language development process starts with the grammar design and ends with the interpretation stage after various definitions for the syntax. With respect to the syntax, the JavaCC tool is used for the definition of the token and expression structures, and the generation of the abstract syntax tree. During the type control and interpretation stages, two Visitor interfaces that use the abstract syntax tree are implemented. The results of some evaluations of the language interpreter are shown for several examples of MaPL programs.

The syntax of the MaPL language is designed to meet the general programming needs. The language source code supports the use of the basic programming language structures, such as variables, expression sequences, selection (or conditional) expressions, loops, and functions in a mathematical way. In the definitions of mathematical functions based on variables, expressions are written in a polynomial-like form. Different execution paths can be created depending on the selection between two or more statements controlled by conditional expressions. Loop expressions can only be represented by the definitions of recursive functions. These mathematical aspects of the language are supported by the results of the comparative analysis made with some other mathematical languages.

The language syntax needs to be extended to allow the programming of many different mathematical operations during the coding process. For example, it is important that complex numbers and fractional numbers can be represented and basic operations can be performed on these numbers. In addition, there is another need for the operations carried out by the logarithmic and trigonometric functions as well as the summation and product notations used on number sequences. These kinds of operations can be supported by adding new keywords to the language syntax or by developing a language library.

REFERENCES

[1] L. K. C. Louden, “Programming Languages: Principles and Practices", *Cengage Learning*, 2011.

[2] R. Harper, "Practical Principles for Programming Languages", 2nd ed., *Cambridge University Press*, 2016.

[3] B. W. Kernighan and D. M. Ritchie, "The C Programming Language", *Prentice Hall Professional Technical Reference*, 1988.

[4] L. Wall, T. Christiansen and J. Orwant, "Programming Perl", 3rd ed., *O'Reilly Media*, 2000.

[5] M. Mateas and N. Montfort, "A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics", In *Digital Arts and Culture: Digital Experience: Design, Aesthetics, Practice (DAC 2005)*, Copenhagen, Denmark, 2005.

[6] International Obfuscated C Code Contest., https://www.ioccc.org/, 2018.

[7] J. Wakerly, "The programming language Pascal", *Microprocessors and Microsystems*, vol. 3, no. 7, pp. 321-326, 1979.

[8] K. Arnold, J. Gosling, and D. Holmes, "The Java programming language", *Addison Wesley Professional*, 2005.

[9] A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde, "C# Programming Language", *Addison-Wesley Professional*, 2010.

[10] G. Sussman and G. L. Steele, Jr., "Scheme: A interpreter for extended lambda calculus",

Higher-Order and Symbolic Computation, vol. 11, no. 4, pp. 405-439, 1998.

[11] S. P. Jones. "Haskell 98 language and libraries: the revised report", *Cambridge University Press*, 2003.

[12] L. Sterling and E.Y. Shapiro, "The Art of Prolog: Advanced Programming Techniques", 2nd ed., *The MIT Press*, 1994.

[13] M. V. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J. F. Morales and G. Puebla, "An overview of Ciao and its design philosophy", *Theory and Practice of Logic Programming - Prolog Systems*, vol.12, no. 1-2, pp. 219-252, 2012.

[14] W. T. Richter, "TEX and Scripting Languages", *Proceedings of the Practical TEX 2004 Conference*, TUGboat, vol. 25, no. 1, pp. 71-88, 2004.

[15] L. Lamport, "LateX : A document preparation system", 2nd ed., *Addison-Wesley*, 1994.

[16] F. Houlette, "SQL: A Beginner's Guide", *McGraw-Hill Education*, New York, US, 2000.

[17] IBM Security QRadar, "Ariel Query Language (AQL) Guide v7.3.1", *IBM Corp.*, 2017.

[18] D. J. Higham , N. J. Higham, The Matlab Guide, 3rd ed., SIAM, 2017.

[19] M. Trott, "The Mathematica guidebook for programming", *Springer*, 2014.

[20] D. E. Thomas, P. R. Moorby, "The Verilog Hardware Description Language", 5th ed., *Springer*, 2002.

[21] D. L. Perry, "VHDL: Programming by Example", *McGraw-Hill Education*, 2002.

[22] M. McKinley, "The Game Animator's Guide to Maya", *Sybex*, 2006.

[23] A. Tavakkoli, "Game Development and Simulation with Unreal Technology", *Routledge*, 2015.

[24] J. Kallrath, "Modeling Languages in Mathematical Optimization", *Springer*, Boston, MA, 2004

[25] J. R. Levine, J. R. Levine, T. Mason and D. Brown, "Lex & Yacc", *O'Reilly Media, Inc*., Sebastopol, CA, USA, 1992.

[26] E. M. Gagnon and L. J. Hendren, "SableCC, an object-oriented compiler framework", In *TOOLS USA 98 (Technology of Object-Oriented Languages and Systems)*, IEEE, 1998.

[27] V. Kodaganallur, "Incorporating language processing into Java applications: A JavaCC tutorial", *IEEE Software*, vol. 21, no. 4, pp. 70–77, 2004.

[28] A. J. Dos Reis, "Compiler Construction Using Java, JavaCC, and Yacc", *IEEE Computer Society, Inc.*, 2012.

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", *Addison-Wesley Reading*, MA, 1995.

[30] R. Harper, Practical Principles for Programming Languages, 2nd ed., *Cambridge University Press*, 2016.