A New Algorithm to Parse a Mathematical Expression and its Application to Create a Customizable Programming Language

Vassili Kaplan
Zurich, Switzerland
e-mail: vassilik@gmail.com

Abstract—**This paper presents an algorithm to parse a string containing a mathematical expression. This algorithm represents an alternative to other parsing algorithms, e.g., the Dijkstra "shunting-yard" algorithm. The algorithm presented here, has the same time complexity, but in our point of view it is easier to implement and to extend. As an example of extending this algorithm, we also present how it can be used to implement a fully customizable scripting programming language. The reference implementation language will be C++.**

Keywords: software development; software engineering; algorithm; parsing; scripting language; programming language.

I. INTRODUCTION

There are already a few parsing algorithms. E. Dijkstra invented his "shunting-yard" algorithm to parse a mathematical expression in 1961 [1]. The Look-Ahead Left-to-Right (LALR) parser was invented by F. DeRemer in 1969 [2]. It is based on the so called "bottom-up" parsing. The LALR parser can be even automatically generated by YACC [3]. There are also a few algorithms for the "top-down" parsers, e.g., a Recursive Descent Parser [4], which is based on the LL(k) grammar (**L**eft-to-right, **L**eftmost derivation) [5]. The ANTLR [6] is a widely used tool to automatically generate parsers based on the LL(*) grammar. In particular, the ANTLR can even generate the Recursive Descent Parser.

Why would we want to develop yet another parsing algorithm if there are so many around, including even the parser generating tools? Aside from its own interest, we believe that it is much easier to implement in an object-oriented language. It is also much easier to extend, i.e., to add your own functionality. As an example of extending the language, we will show how one can implement a scripting language from scratch, without using any external libraries. One can add new functions to the language on the fly. One can also add programming language keywords in any human language with just a few configuration changes, which we will also illustrate in this article.

We have presented an alternative to this algorithm in [7] [8][9]. We call this alternative algorithm the "split-and-merge" algorithm, because, analogous to other parsing algorithms, it consists of two steps. In the first step, we split the whole expression into the list of so called "cells". Each cell consists of a number and an "action" to be applied to that cell. In the second step, we merge all the cells together according to the priorities of the "actions". In this article, we extend this work and show a more generalized parsing algorithm.

There is already a reference implementation of a scripting language using the split-and-merge algorithm in C# [10]. But the language described there was not very mature. In particular, there is a section towards the end of the article that mentions some of the missing features. Here, we are going to show how to implement those features, and a few more.

For simplicity, we will continue calling the parsing algorithm as the "split-and-merge" algorithm and the scripting language created by using that algorithm as **C**ustomized **S**cripting in **C**++ using the **S**plit-and-merge algorithm ("CSCS").

The rest of this paper is organized as follows. Section II presents the split-and-merge algorithm. Section III shows how to register variables and functions with the parser. Section IV addresses writing of custom functions in CSCS. Section V describes the try, throw and catch control flow statements. Section VI shows how to provide CSCS keywords in any human language. Summary and conclusions appear in Section VII.

II. THE SPLIT-AND-MERGE ALGORITHM

Here, we are going to generalize the split-and-merge algorithm described in [7][8][9]. The algorithm can parse not only a mathematical expression but any language statement. A separation character must separate all the statements. We define this separation character as a semicolon, ";".

The algorithm consists of two steps.

In the first step, we split the given string into the list of objects, called "Variables". Each "Variable" consists of an intermediate result (a number, a string, or an array of other Variables) and an "action" that must be applied to this Variable. In in [1][2][3], we called this "Variable" a "Cell" and it could have only a numerical result.

The last element of the created list of Variables has a so called "null action", which, for convenience, we denote by the character ")". It has the lowest priority of 0.

For numbers, an action can be any of "+", "-", "*", "/", "%", "&&", "||", and some others. For strings, only "+" (concatenation) and logical operators "<", "<=", ">", ">=", "==", "!=" are defined.

Listing 1 contains a part of the Variable definition:

```
struct Variable {
  string toString() const;

  bool canMergeWith(const Variable& right);
  void merge(const Variable& right);

  void mergeNumbers(const Variable& right);
  void mergeStrings(const Variable& right);

  static int getPriority(const string& action);
  // ------------------
```

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-498-5

```
  double numValue;
  string strValue;
  vector<Variable> tuple;
  string action;
  string varname;
  Constants::Type type;
};
```

Listing 1: Variable data structure

The separation criteria for splitting the string into Variables are: an action, an expression in parentheses, or a function, previously registered with the parser. We are going to describe how to register a function with the parser in the next section. In case of an expression in parentheses or a function, we apply recursively the whole split-and-merge algorithm to that expression in parentheses or the function argument in order to get a Variable object as a result. So, at the end of the first step, we are going to have a list of Variables, each one having an action to be applied to the next Variable in the list. See the main parsing cycle of the first part of the algorithm in Listing 2.

```
do { // Main processing cycle of the first part.
  char ch = data[from++];
  string action = Constants::EMPTY;
  bool keepCollecting =  stillCollecting(data, from,
                         parsingItem, to, action);
  if (keepCollecting) {
    // The char still belongs to the previous operand
    parsingItem += ch;

    if (contains(to, data[from])) {
            continue;
    }
  }
  ParserFunction func(data, from, parsingItem,
                      ch, action);
  Variable current = func.getValue(data, from);

  char  next  =  from  <  data.size()  ?  data[from]  :
        NULL_CHAR;

  bool done = listToMerge.empty() &&
    (action == NULL_ACTION || next == END_STATEMENT);
  if (done) { // Not a math expression.
    listToMerge.push_back(current);
    return listToMerge;
  }
  current.action = action;
  listToMerge.push_back(current);
  parsingItem.clear();
} while (from < data.size() &&
        !contains(to, data[from]));
```

Listing 2: The split part of the split-and-merge algorithm

The second step consists in merging the list of Variables created in the first step, according to the priorities of their actions. The priorities of the actions are defined in Listing 3.

```
unordered_map<string, int> prio;
    prio["++"] = 10;
    prio["--"] = 10;
    prio["^"]  = 9;
    prio["%"]  = 8;
    prio["*"]  = 8;
    prio["/"]  = 8;
    prio["+"]  = 7;
    prio["-"]  = 7;
    prio["<"]  = 6;
    prio[">"]  = 6;
    prio["<="] = 6;
    prio[">="] = 6;
    prio["=="] = 5;
    prio["!="] = 5;
    prio["&&"] = 4;
    prio["||"] = 3;
    prio["+="] = 2;
    prio["-="] = 2;
    prio["*="] = 2;
    prio["/="] = 2;
    prio["%="] = 2;
    prio["="]  = 2;
```

Listing 3: Priorities of the actions

Two Variable objects can only be merged together if the priority of the action of the Variable on the left is greater or equal than the priority of the action of the Variable on the right. Otherwise, we merge the Variable on the right with the Variable on its right first, and so on, recursively. As soon as the right Variable has been merged with the Variable next to it, we return back to the original, left Variable, and try to re-merge it with the newly created right Variable. Note that eventually we will be able to merge the entire list since the last variable in this list has a null action with the priority zero.

The implementation of the second step is shown in Listing 4. The function merge() is called from outside with the mergeOneOnly parameter set to false.

```
Variable Parser::merge(Variable& current, size_t&
 index, vector<Variable>& listToMerge, bool mergeOne) {
  while (index < listToMerge.size()) {
    Variable& next = listToMerge[index++];

    while (!current.canMergeWith(next)) {
      merge(next, index, listToMerge,
            true/*mergeOne*/);
    }
    current.merge(next);
    if (mergeOne) {
      break;
    }
  }
  return current;
}
```

Listing 4: The merge part of the split-and-merge algorithm

*A. Example of parsing 1 – 2*sin(0)*

Let us see the algorithm in action, applying it to the "1 – 2*sin(0)" string. "1-" and "2*" tokens are parsed and converted into Variables directly, without any problem:

Split(1 – 2*sin(0)) →
  Variable(numValue = 1, action = "-"),
  Varaible(numValue = 2, action = "*"),
  Split-And-Merge(sin(0)).

To proceed, we need to process the "sin(0)" string first, applying the whole spilt-and-merge algorithm to it.

When the parser gets the "sin" token, it maps it to the sine function registered earlier (we will discuss registering functions in the next section). Then the parser evaluates the sin(0) and returns 0 as a result.

Therefore, the result of splitting the original string "1 – 2*sin(0)" will be a list consisting of three Variables:

1. Variable(numValue = 1, action = "-")
2. Varaible(numValue = 2, action = "*")
3. Varaible(numValue = 0, action = ")")

In the second step of the algorithm, we merge the resulting list of variables one by one.

Note that we cannot merge directly the first Variable with the second one since the priority of the action of the first Variable "-", is less than the priority of the action of the second variable, "*", according to the Listing 3. Therefore, we need to merge first Variables 2 and 3. The priority of the "*" action is greater than the priority of the null action ")" (the last Variable in the list has always a "null" action). So we can merge 2 and 0: applying the action "*", the result will be 2 * 0 = 0. The resulting variable will inherit the action from the right Variable, i.e., the "null" action ")".

Now we return back to the first Variable and merge it with the newly created Variable(numValue = 0, ")"). Applying action "-" to the both variables we get the final result: 1 – 0 = 0. Therefore, the split-and-merge("1 – 2*sin(0)") = 0.

Using the algorithm above with the recursion, it is possible to parse any compound CSCS expression. The architectural diagram of the split-and-merge algorithm and its usage to parse a string appears in Figure 1. Here is an example of the CSCS code:

```
x = sin(pi^2);
cache["if"] = -10 * x;
cache["else"] = 10 * x;
if (x < 0 && log(x + 3*10^2) < 6*exp(x) ||
             x < 1 - pi) {
  print("in if, cache=", cache["if"]);
} else {
  print("in else, cache=", cache["else"]);
}
```

The code above has a few functions (sin(), exp(), log(), print()) and a few control flow statements (if, else). How does the parser know what to do with them?

III. Registering Variables and Functions with the Parser

All the functions that can be added to the parser must derive from the ParserFunction class. Listing 5 contains an excerpt from the ParserFunction class definition.

```
class ParserFunction {
public:
  ParserFunction(const string& data, size_t& from,
       const string& item, char ch, string& action);

  Variable getValue(const string& data, size_t& from){
    return m_impl->evaluate(data, from);
  }
protected:
  virtual Variable evaluate(const string& data,
                            size_t& from) = 0;
  Variable::emptyInstance;
  static StringOrNumberFunction* s_strOrNumFunction;
  static IdentityFunction*       s_idFunction;
}
```

Listing 5: The ParserFunction class definition

The Identity is a special function, which is called when we have an argument in parentheses. It just calls the main entry method of the split-and-merge algorithm to load the whole expression in parentheses:

```
class IdentityFunction : public ParserFunction {
public:
  virtual Variable evaluate(const string& data,
                            size_t& from) {
    return Parser::loadAndCalculate(data, from);
  }
};
```

The parser will call the evaluate() method on any class deriving from the ParserFunction class as soon as the parser gets a token corresponding to the function registered with the parser. There are three basic steps to register a function with the parser:

- Define the function keyword token, i.e., the name of the function in the scripting language, CSCS, e.g.:

```
static const string SIN; // in Constants.h
const string Constants::SIN = "sin"; // in .cpp
```

- Implement the class to be mapped to the keyword from the previous step. Basically, the evaluate() method must be overridden. E.g., for the sin() function:

```
class SinFunction : public ParserFunction {
public:
  virtual Variable evaluate(const string& data,
                            size_t& from) {
     Variable arg = Parser::loadAndCalculate(
                            data, from);
     return ::sin(arg.numValue);
  }
};
```

  loadAndCalculate() is the main parser entry point, which calculates the value of the passed expression.

- Map the object of the class, implemented in the previous step, with the previously defined keyword as follows:

```
ParserFunction::addGlobalFunction(Constants::SIN,
                                  new SinFunction());
```

The addGlobalFunction() method just adds a new entry to the global dictionary used by the parser to map the keywords to functions:

```
void ParserFunction::addGlobalFunction(const string&
                     name, ParserFunction* function) {
  auto it = s_functions.find(name);
  if (it != s_functions.end()) {
    delete it->second;
  }
  s_functions[name] = function;
}
```

Similarly, we can register any function with the parser, e.g., if(), while(), try(), throw(), etc.

We can also define local or global variables in the same way. In the next section, we are going to see how to define functions in CSCS and add passed arguments as local variables to CSCS.

IV. Writing Functions in CSCS

To write a custom function in the scripting language, two functions had to be introduced in C++, FunctionCreator and CustomFunction, both deriving from the ParserFunction base class. As soon as the Parser gets a token with the "function" keyword, it will call the evaluate() method on the FunctionCreator object, see Listing 6.

```
Variable FunctionCreator::evaluate(const string& data,
                                   size_t& from) {
  string funcName = Utils::getToken(data,
```

```
                        from, TOKEN_SEPARATION);
  vector<string> args =
         Utils::getFunctionSignature(data, from);
  string body = Utils::getBody (data, from, '{', '}');
  CustomFunction* custFunc = new CustomFunction(
                          funcName, body, args);
  ParserFunction::addGlobalFunction(
                       funcName, custFunc);
  return Variable(funcName);
}
```

Listing 6: The function creator class

Basically, it just creates a new object, CustomFunction, and initializes it with the extracted function body and the list of parameters. It also registers the name of the custom function with the parser, so the parser maps that name with the new CustomFunction object, which will be called as soon as the parser encounters the function name keyword.

So all of the functions that we implement in the CSCS code correspond to different instances of the CustomFunction class. The custom function does primarily two things, see Listing 7. First, it extracts the function arguments and adds them as local variables to the Parser (they will be removed from the Parser as soon as the function execution is finished or an exception is thrown). It also checks that the number of actual parameters is equal to the number of the registered ones (this part is skipped for brevity).

```
Variable CustomFunction::evaluate(const string& data,
                                  size_t& from) {
  vector<Variable> args = Utils::getArgs(data, from,
                               START_ARG, END_ARG);
  // 1. Add passed arguments as locals to the Parser.
  StackLevel stackLevel(m_name);

  for (size_t i = 0; i < m_args.size(); i++) {
   stackLevel.variables[m_args[i]] = args[i];
  }
  ParserFunction::addLocalVariables(stackLevel);

  // 2. Execute the body of the function.
  size_t funcPtr = 0;
  Variable result;

  while (funcPtr < m_body.size() - 1) {
   result = Parser::loadAndCalculate(m_body, funcPtr);
   Utils::goToNextStatement(m_body, funcPtr);
  }
  // 3. Return the last result of the execution.
  ParserFunction::popLocalVariables();
  return result;
}
```

Listing 7: The custom function class

Secondly, the body of the function is evaluated, using the main parser entry point, the loadAndCalculate() method.

If the body contains calls to other functions, or to itself, the calls to the CustomFunction can be recursive.

Let us see this with an example of a function implemented in CSCS. It calculates the so called Catalan numbers (named after a Belgian mathematician Eugène Catalan), see Listing 8.

```
// Catalan numbers function implemented in CSCS.
// C(0) = 1, C(n+1) = Sum(C(i) * C(n - i)), i: 0-> n
// for n >= 0. Equivalent to:
// C(n) = 2 * (2*n - 1) / (n + 1) * C(n-1), n > 0
function catalan(n) {
  if (!isInteger(n)) {
   exc = "Catalan is for integers only (n="+ n +")";
   throw (exc);
  }
  if (n < 0) {
   exc = "Negative number (n="+ n +") supplied";
   throw (exc);
  }
  if (n <= 1) {
   return 1;
  }
  return 2 * (2*n - 1) / (n + 1) * catalan(n - 1);
}
```

Listing 8: Recursive calculation of Catalan numbers implemented in CSCS

The Catalan function above uses an auxiliary isInteger() function:

```
function isInteger(candidate) {
  return candidate == round(candidate);
}
```

isInteger() function calls yet another, round() function. The implementation of the round() function is already in the C++ code and is analogous to the implementation of the sine function that we saw in the previous section.

To execute the Catalan function with different arguments, we can use the following CSCS code:

```
try {
  c = catalan(n);
  print("catalan(", n, ")=", c);
} catch(exc) {
  print("Caught: " + exc);
}
```

It gets the following output for different values of n:

```
Caught: Catalan is for integers only (n=1.500000) at
  catalan()
Caught: Negative number (n=-10) supplied at
  catalan()
catalan(10)=16796
```

Since the exception happened at the global level, the exception stack printed consisted only of the catalan() function itself.

The CSCS code above contains try(), throw(), and catch() control flow statements. How are they implemented in C++?

V. THROW, TRY, AND CATCH CONTROL FLOW STATEMENTS

The throw() and try() control flow statements can be implemented as functions in the same way you saw the implementation of the sine function above. The catch() is not implemented as a separate function but is processed right after the try() block.

Both implementations derive from the ParserFunction class as well. First we show the more straightforward one, the throw() function:

```
Variable ThrowFunction::evaluate(const string& data,
                                 size_t& from) {
    // 1. Extract what to throw.
    Variable arg = Utils::getItem(data, from);

    // 2. Convert it to string.
    string result = arg.toString();

    // 3. Throw it!
    throw ParsingException(result);
}
```

The try() function requires a bit more work. Here is an excerpt:

```
Variable TryStatement::evaluate(
      const string& data, size_t& from) {
  size_t startTryCondition = from - 1;
  size_t stackLevel =
         ParserFunction::currStackLevel();
   ParsingException exception;
  Variable result;
  try {
    result = processBlock(data, from);
  }
  catch(ParsingException& exc) {
    exception = exc;
  }
```

First, we note where we started the processing (so later on we can return back to skip the whole try-catch block). Then, we simply process the try block, and if the exception is thrown, we catch it. In the parser code, we throw only exceptions of type ParsingException, which is a wrapper over the C++ std::exception.

If there is an exception, then we need to skip the whole catch block. For that, we go back to the beginning of the try block and then skip it.

```
if (!exception.msg().empty()) {
  from = startTryCondition;
  skipBlock(data, from);
}
```

After the try block, we expect a catch token and the name of the exception to be caught, regardless if the exception was thrown or not:

```
string catch = Utils::getNextToken(data, from);
if (CATCH_LIST.find(catch) == CATCH_LIST.end()) {
  throw ParsingException("Expected a catch but got ["+
                         catch +"]");
}
string excName = Utils::getNextToken(data, from);
```

The reader may have noticed that when checking if the "catch" keyword was following the try-block or not, we didn't compare the extracted token with the "catch" string, but with a CATCH_LIST. The reason is that the CATCH_LIST contains all possible translations of the "catch" keyword to any of the languages that the user may supply in the configuration file. How is a keyword translation added to the parser?

VI. PROVIDING KEYWORDS IN DIFFERENT LANGUAGES

One of the main advantages of writing a custom programming language is the possibility to have the keywords in any language (besides the "base" language, understandably chosen to be English).

Here is how we can add the custom keyword translations to the CSCS language.

First, we define them in a configuration file. Here is an incomplete example of a configuration file with Russian translations:

```
function    = функция
include     = включить
if          = если
else        = иначе
elif        = иначе_если
return      = вернуться
print       = печать
size        = размер
while       = пока
```

The same configuration file may contain an arbitrary number of languages. After reading the keyword translations, we add them to the parser one by one:

```
void addTranslation(const string& originalName,
                    const string& translation) {
  ParserFunction* originalFunction =
  ParserFunction::getFunction(originalName);
  if (originalFunction != 0) {
    ParserFunction::addGlobalFunction(
         translation, originalFunction);
  }

  tryAddToSet(originalName, translation, CATCH,
              CATCH_LIST);
  tryAddToSet(originalName, translation, ELSE,
              ELSE_LIST);
  // other sets
}
```

First, we try to add a translation to one of the registered functions (like sin(), cos(), round(), try(), throw(), etc.).
Then, we try to add them to the sets of additional keywords, that are not functions (e.g., the "catch" is processed only together with the try-block, the "else" and "else if" are processed only within the if-block, etc.

The tryAddToSet() is an auxiliary template function that adds a translation to a set in case the original keyword name belongs to that set (e.g., CATCH = "catch" belongs to the CATCH_LIST).

Here is the implementation of some CSCS code using Russian keywords. The code below just goes over the defined array of strings in the while loop and prints every other element of that list:

```
слова = {"Это", "написано", "по-русски"};
разм = размер(слова);
и = 0;
пока(и < разм) {
  если (и % 2 == 0) {
    печать(слова[и]);
  }
  и++;
}
```

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we presented an algorithm to parse a string containing an expression and then we saw how we can apply this algorithm to parse a customized scripting programming language, that we called CSCS.

The implementation of the sin() function and throw() and try() control flow statements was shown. We saw that implementing a math function and a control flow statement is basically the same: one needs to write a new class, deriving from the ParserFunction class, and override its evaluate() method. Then one needs to register that function with the parser, mapping it to a keyword. The evaluate() method will be called by the parser as soon as the parser extracts a keyword corresponding to this function. For the lack of space, we didn't show how to implement if(), while(), break, continue, and return control flow statements but they are all implemented analogously. The same applies to the

prefix and postfix ++ and -- operators, that we did not have space to show.

Using the above approach of adding a new function to the parser, anything can be added to the CSCS language as long as it is possible to implement it in C++.

Even though the time complexity of the split-and-merge algorithm is the same as of the Dijkstra's algorithm (and the split-and-merge might be a bit slower since it uses a lot of recursions), we believe that the main advantage of the split-and-merge algorithm is in that it is easier to extend and add user specific code. It can be extended in two ways: one is to add new functions or control flow statements, and the second is to add keyword translations in any human language.

For the future, we plan to focus on extending the CSCS language: adding more features and more functions. CSCS already supports if-else, try-catch, throw, while, function, return, break, continue, and include file control flow statements; arrays with an unlimited number of dimensions (implemented as vectors) and dictionaries (also with an unlimited number of dimensions, implemented as unordered maps). We plan to add a few other data structures to CSCS as well.

We also plan to add more common operating system functions, for example a task manager, listing processes with a possibility to kill a process and a process scheduler system. The challenge there is that these functions are operating system dependent and require multiple implementations for each operating system. One could also add some external libraries, which hide the implementations for different operating systems, but our goal is not to have any external libraries at all.

Another idea is to extend CSCS towards a functional programming language, something like F# - where a few very short language constructs implement quite a few language statements behind the scenes. We believe that this is easy to implement using the split-and-merge function implementation approach.

REFERENCES

[1] E. Dijkstra, "Shunting-yard algorithm", https://en.wikipedia.org/wiki/Shunting-yard_algorithm Retrieved: June 2016.

[2] F. DeRemer, T. Pennello, "Efficient Computation of LALR(1) Look-Ahead Sets". Transactions on Programming Languages and Systems (ACM) **4** (4): 615–649.

[3] Yet Another Compiler Compiler, YACC, https://en.wikipedia.org/wiki/Yacc. Retrieved: June 2016

[4] Recursive Descent Parser, https://en.wikipedia.org/wiki/Recursive_descent_parser Retrieved: June 2016.

[5] LL parser, https://en.wikipedia.org/wiki/LL_parser Retrieved: June 2016.

[6] ANTNLR, https://en.wikipedia.org/wiki/ANTLR Retrieved: June 2016.

[7] V. Kaplan, "Split and Merge Algorithm for Parsing Mathematical Expressions", ACCU CVu, 27-2, May 2015, http://accu.org/var/uploads/journals/CVu272.pdf Retrieved: June 2016.

[8] V. Kaplan, "Split and Merge Revisited", ACCU CVu, 27-3, July 2015, http://accu.org/var/uploads/journals/CVu273.pdf Retrieved: June 2016.

[9] V. Kaplan, "A Split-and-Merge Expression Parser in C#", MSDN Magazine, October 2015, https://msdn.microsoft.com/en-us/magazine/mt573716.aspx Retrieved: June 2016.

[10] V. Kaplan, "Customizable Scripting in C#", MSDN Magazine, February 2016, https://msdn.microsoft.com/en-us/magazine/mt632273.aspx Retrieved: June 2016.

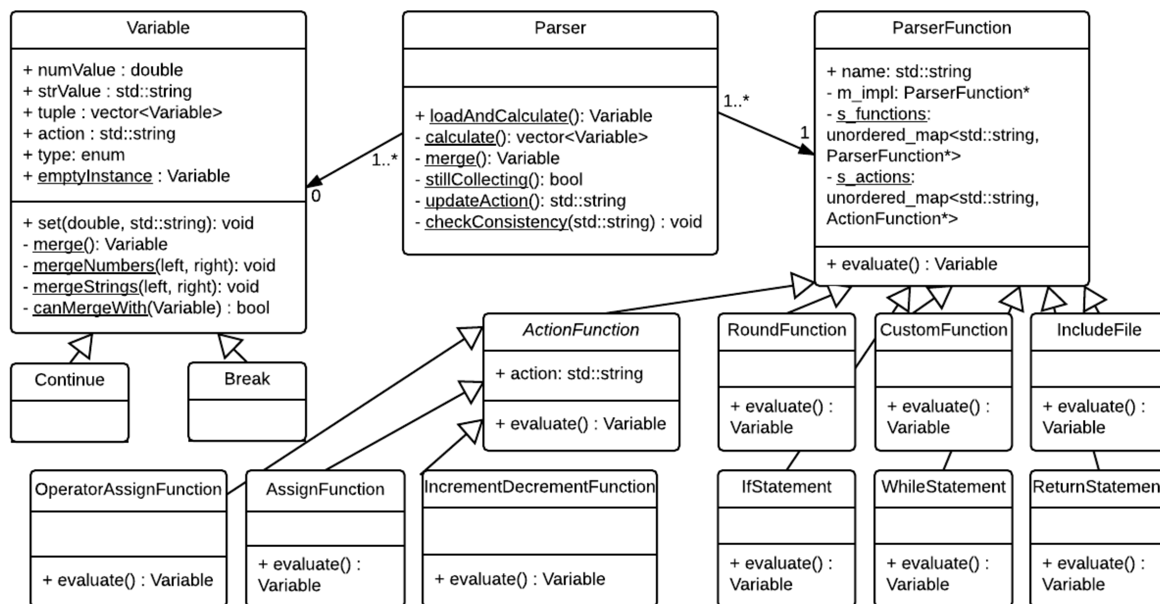

Figure 1. The Parser Framework UML Class Diagram

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-498-5