# Modular Programming Language Design

Gil Müller, gil.mueller@nexgo.de

29.4.2014

*This paper discusses how to provide a modular programming language design. The main idea is that modules are created with respect to language features – similar to the idea of aspect-oriented programming. The paper disucsses the concept separartely for syntax and processing. Quite important for the modularization are parametric snytactic rules(or generic functions in the domain of processing), which are particularly useful to deal with dependencies. The modularization concept is demonstrated with a language which is extended several times.*

## 1. Motivation

The making of programming languages is not something for a selected few, but rather needed by many people. Domain-specific languages (DSL) are useful for many purposes. Unfortunately, with the present state of programming language design the making of such a language is no easy task in comparism to programming itself.

The reason for this it that the designer of such a language more or less starts from scratch. She or he has to reason about a full language even if for instance just a new operator shall be used. DSLs are typically a superset over a general purpose language: they rely mostly on the abilities of the host language and only extend it with new features which are needed for the domain.

We assume, that a programming language can be divided into a set of such features. A feature is given by its syntactical forms and the processing instructions for it. Features may depend on each other. E.g. a given feature may use parts of another feature. Additionally, there may be overlap: two given features may use the some form (cf. below for form overloading).

What is needed for the design of a DSL is the ability to extend a programming language syntactically and semantically without the need to redesign already existing language features. An interesting point might be here to have a sort of language toolkit which lets the user decide which language features to use. Furthermore, sometimes it is needed to implement certain "house rules" which restricts the used language features. This is particularly important for teaching a programming language. In general, language design should not be more complex than programming.

In this paper, we develop an approach for a modular programming language design (MPLD). Such a design consists of a set of features which can be added or removed as a whole – not considering dependencies. Like a Java program, which consists of a set of packages, a MPLD is divided into modules, which specifies the syntax and the processing of a language. This contrasts with the present approach where typically the language is specified as a whole divided by syntactic and semantic layers.

Such a MPLD simplifies the adding of new features since changes have less impact. Additionally, it is also possible to configure the language to the requirements of a given project. The flip-side of this approach, that it might complicate to see the language as a whole. But, this can avoided with the right tooling which create "snapshots" from a given configuration. Another issue is that the modularization of a given language requires some experience, since it determines the abilities to extend the language. But, for the initial design anyway an experienced designer is needed, since a full language has to be made. If the design is well made, later changes require in general less proficiency.

If we look to related work, there are first the means of modularization in the programming languages itself. E.g an object-oriented language like Java [11] provides modularization on class level. Additionally, there is a package concept, which particularly supports component-oriented programming [12]. Both concepts enable information hiding and abstraction in the container and deal with dependencies.

Aspect-oriented programming [7] provided insight with its focus on separation of cross-cutting concerns. Like those concerns our approach focuses on treating language features as a whole and not to deal with them in separate layers. I.e. the datatype integer as a feature contains the respective syntactic rules and the processing for its constants and operations.

Regarding work with respect to tooling for DSL, there are first compiler-compilers (e.g. [8] and [9]). Such tools typically automate the lexical and syntactic analysis.  In terms of modularity they require the use of layers.

The Xtext framework ([10]) provides with Xbase, subset of the Java language, a sort of language toolkit, which can be adapted to user's requirements. It is rather limited in scope since the language rules are not directly available for modification. But, the user is able to decide on which rules to include in the DSL.

Also programming languages like Scheme [2], Ruby [3] or Scala [4] provide support for DSLs. Typically the user can specify a macro which extends the language's syntax. This is as good as it is not necessary to change the semantics of host language or the intention is to restrict the host language.

In the following sections of the paper, we will start with examination about modularization regarding form. This is continued with modularization regarding processing. Afterwards the concept is demonstrated with a simple language. The paper closes with a discussion.

## 2. Modularization Regarding Form

A language processor like a compiler or an interpreter can be described by a number of layers. Such a layer uses some kind of source language which is processed. The processing can be just some analysis like type checking or a translation, where code in some target language is produced.

### *2.1 Syntactic Rules*

For our analysis we assume, that the source language can be specifed by some grammar – even if the syntax is abstract. For the specification of such grammar a typically the Backus-Naur-Form (BNF) is used (or some extension).

Syntactic rules, which are used here, consist just of alternatives which consist of sequences of terminals and nonterminals. A rough specification is as follows (lexical issues are rather left unspecified):

```
syntacticRule = nonterminal "=" alternatives
alternatives =
               | sequence
               | alternatives "|" sequence
```

```
sequence =
         | symbol sequence
symbol = terminal
       |  nonterminal
terminal = """ characters """
nonterminal =  nonterminalCharacters
characters =
           | character characters
nonterminalCharacters =
                      | nonterminalCharacter nonterminalCharacters

where character is any (printable) character and nonterminalCharacter
is any alphanumeric character.
```

Extensions for optional and multiple elements in a sequence can be reduced to the above form.

## *2.2 Dimensions in Syntactic Rules*

We will now try to analyse which issues arise when we decide to do a MPLD. Syntactic rules have metaphorically spoken a width dimension and a height dimension. The width dimension is given by the length of the sequence in an alternative and the height dimension is given by the number of alternatives.

We can increase the height quite easily by adding a new alternative. Sometimes the order of the alternatives is distinctive (e.g. for lexical analysis). Then, we have also to decide where to add the new alternative. We will call this action alternative extension.

Removing an alternative is another option. For this, we have to able to designate the alternative to be removed. This can be accomplished by using an index for alternatives. This is the alternative reduction action.

Regarding the width it is possible to change the sequence (adding, removing, or changing symbols wherever in the sequence). This is the sequence change action.

## *2.3 Parameterized Rules*

The non-terminals which are used in a sequence are global references. The scope is always the whole grammar. For an MPLD such references shall be avoided, since it is complicated to change the hierarchy. Additionally, rules are always bound to a particular context.

To enable rules with such local references, we introduce parameterized rules. E.g. the rule:

```
add = operand
    | operand "+" add
```

becomes

```
add(operand) = operand
             | operand "+" add(operand)
```

Now, the non-terminal operand is just local in the rule. Parameterized rules are instantiated with sequences of terminals or nonterminals (the sequence may even be empty):

```
start = add(constant)
constant = number
```

The instantiation is like the reference of a non-terminal except that also parameters are passed.

Additionally, we use the special non-terminal self to refer to the current syntactic rule:
```
      add(operand) = operand
                  |  operand "+" self
```

The grammar for the syntactic rules is now:
```
      syntacticRule = nonterminal parameters "=" alternatives
      parameters =
                 | "(" parameterList ")"
      parameterList =  nonterminal
                    |  parameterList "," nonterminal
      alternatives =
                   | sequence
                   | alternatives "|" sequence
      sequence =
               | symbol sequence
      symbol = terminal
             |  nonterminal arguments
      terminal = """ characters """
      nonterminal =  nonterminalCharacters
      arguments =
                | "(" argumentList ")"
      argumentList = sequence
                   | argumentList "," sequence
      characters =
                 | character characters
      nonterminalCharacters =
                            | nonterminalCharacter  nonterminalCharacters

      where character is any (printable) character and nonterminalCharacter is
any alphanumeric character.
```

With this new mechanism our grammar can be expressed more tightly:
```
      syntacticRule = nonterminal parameters "=" alternatives
      parameters = parList(nonterminal)
      alternatives =
                   | sequence
                   | alternatives "|" sequence
      sequence = series(symbol)
      symbol = terminal
             |  nonterminal arguments
      arguments = parList(sequence)
      terminal = """ series(character) """
      nonterminal =  series(nonterminalCharacter)
      parList(listElement) =
                           | "(" list(listElement) ")"
      list(element) =  element
                    |  self "," element
      series(element) = element
                      | self element

      where character is any (printable) character and nonterminalCharacter is
any alphanumeric character.
```

As this example demonstrates, parameterized rules can also used to deal with repeating patterns in the grammar. The use of a parameterized rule is rule abstraction. The instantiation is called rule instantiation.

Attributed grammars [5], seem similar. There, attributes are attached to non-terminals in the form of identifiers – like our parameters. In contrast to our approach the rules itself are not touched, but are rather enriched with data that is used for the processing of the parsed result. In our approach, parameters are used to provide abstractions over rules. Parameters replace part of the concrete rule.

Van Wijngaarden grammars [6] are two level grammars, where the upper grammar specifies the values for special parameters in the lower grammar. Like in our approach all non-terminals and terminals can be specified in the upper grammar and become values for the parameters. Our approach is simpler, since instantiation is restricted to single (free) arguments, while in a Van Wijngaarden grammar such arguments can assume an infinite number of values.

## *2.4 Partial Grammars*

It is quite common for some language feature to have a group of syntactic rules, which are not necessarily dependent on each other. In our example the following rules integrate the parameterized rule in the language:

```
      symbol = terminal
             |  nonterminal arguments
      syntacticRule = nonterminal parameters "=" alternatives
      parameters = parList(nonterminal)
      arguments = parList(sequence)
      parList(listElement) =
                           | "(" list(listElement) ")"
      list(element) =  element
                    |  self "," element
```

In order to have a clear separation, we transform the first two rules and the respective dependent rule:

```
      symbol = terminal
             | nonterminal
             | nonterminalWithArguments
      nonterminalWithArguments =  nonterminal parList(sequence)
      syntacticRule = standardRule
                    | parameterizedRule
      standardRule = nonterminal "=" alternatives

      parameterizedRule = nonterminal parList(sequence) "=" alternatives
      parList(listElement) = "(" list(listElement) ")"
```

We can now divide the grammar into three partial grammars:

```
      grammar Basics:
            alternatives =
                         | sequence
                         | alternatives "|" sequence
            terminal = """ series(character) """
            nonterminal =  series(nonterminalCharacter)
            sequence = series(symbol)
            series(element) = element
                            | self element

      grammar ParameterizedRule:
            parameterizedRule =  nonterminal parList(nonterminal) "="
alternatives
            nonterminalWithArguments =  nonterminal parList(sequence)
            parList(listElement) = "(" list(listElement) ")"
            list(element) =  element
                          |  self "," element
```

```
grammar Main:
      syntacticRule = standardRule
                    | parameterizedRule
      standardRule = nonterminal "=" alternatives
```

We have now dependencies from Main to Basics and ParameterizedRule and from ParameterizedRule to Basics. Later, we will show how to mark those dependencies.

## 2.5 Form Overloading

Up to now, language features were mainly regarded with respect to their syntax. This is not a problem, if the processing we mentioned above can be based on the respectic form without considering the context. But sometimes, we have to make a distinction. Consider the following example. We have a rule for addition:

```
add = operand
    | add "+" operand
```

This form can be used for the addition of integer. But, it might also be used for string concatenation. In the processing, we have to consider the type of the operand to distinguish between the different meanings. This is operator overloading. To generalize this phenomenon that a form may have multiple meaning, we call it form overloading.

For the modularization, that can be a problem. If use we partial grammars for the integers and strings. Into which grammar we put the add rule? The simple solution is put a rule in both grammars. Regarding the processing, we explain later how to handle it.

We introduce two more rules forms:

+= adds the alternatives to given rule; if an alternative is already present in the rule, it is not added.

-= removes the alternative from the given rule.

To include those forms the following changes apply:

```
grammar Basics:
      alternatives =
                   | sequence
                   | alternatives "|" sequence
      terminal = """ series(character) """
      nonterminal =  series(nonterminalCharacter)
      sequence = series(symbol)
      series(element) = element
                      | self element
      ruleAssignment = "="
                     | "+="
                     | "-="

grammar ParameterizedRule:
      parameterizedRule = nonterminal parList(nonterminal)
                          ruleAssignment alternatives
      nonterminalWithArguments =  nonterminal parList(sequence)
      parList(listElement) = "(" list(listElement) ")"
      list(element) =  element
                    |  self "," element
```

```
grammar Main:
      syntacticRule = standardRule
                    | parameterizedRule
      standardRule = nonterminal ruleAssignment alternatives
```

## 2.6 Using Grammars

To explain the issues for integrating features, we formalize the use of partial grammar. A partial grammar has the following form:

```
grammar <name> {
      use-forms
    rules
}
```

A use form declares which other grammar is used. It has the following structure:

```
use <name>
```

Our grammar is then as follows:

```
grammar Basics {
      alternatives =
                   |  sequence
                   | alternatives "|" sequence
      terminal = """ series(character) """
      nonterminal =  series(nonterminalCharacter)
      sequence = series(symbol)
      series(element) = element
                      | self element
      ruleAssignment = "="
                     | "+="
                     | "-="
}

grammar ParameterizedRule {
      use Basics

      parameterizedRule = nonterminal parList(nonterminal)
                          ruleAssignment alternatives
      nonterminalWithArguments =  nonterminal parList(sequence)
      parList(listElement) = "(" list(listElement) ")"
      list(element) =  element
                    |  self "," element
}

grammar Main {
      use Basics
      use ParameterizedRule

      grammar = "grammar" "{" uses syntacticRules "}"
      uses = series(use)
      syntacticRules = series(syntacticRule)
      use = "use" nonterminal
      syntacticRule = standardRule
                    | parameterizedRule
      standardRule = nonterminal ruleAssignment alternatives
}
```

Use forms merge the rules from the referenced grammar with the present grammar. If a rule from the referenced grammar is already contained in the present grammar and is of the form nonterminal = alternatives,  it will not be included. The other forms are merged as it was specified above.

### *2.7 Integration of Features*

Regarding the integration of features, we have the choice, to arrange the integration in the grammar, where the feature is specified or to do it in the grammar where the feature is used.  In our example the rule parameterizedRule is integrated in the Main grammar (as another alternative for syntacticRule). But, it also possible to do the integration in  ParameterizedRule by adding the rule:

```
            syntacticRule += parameterizedRule
```

and removing the respective alternative from syntacticRule in Main.

It is up to the language designer to decide on the choice. The first choice provides a straight-forward integration. But, it might become difficult, if the integration depends on other features. E.g. if an operator needs to be integrated, it needs to be put in some place in a hierarchy of operators. The order may change if another operator is inserted. In this case it seems simpler to define the operator hierarchy in the grammar which uses the operators.

# 3. Modularization Regarding Processing

In the previous section we discussed how modularization can be achived in terms of the syntax. In this section we complete the approach by considering also the processing of the parsed data. The basis is the concept that was defined in [1]. This concept will be extended to better support modularization.

### *3.1 Mapping Syntactic Rules to Functions*

We assume that the compiler or interpreter is given by a stack of layers. Which layers and how many are used is the choice of the language designer.

A layer is given by a set of functions which give syntactic expressions some meaning. This can be a mapping between two languages  (e.g. the lexical analysis) , some validity checking (like type checking), or the interpretation of the expression.

A syntactive rule is associated with a function which bears the name of the rule. The contents of each alternative of the rule are passed to the function. The syntactic expression will be the parameters of the function. E.g the rule:

```
add = self "+" addoperand

           | addoperand
```

is transformed to

```
add(self, "+", addoperand) = <some mapping code>

add(addoperand) = <some mapping code>
```

Note, that it is possible to use an identifier for terminals to name it:

```
add(self, literal="+", addoperand) = <some mapping code>
```

The parameters are (implicitly) bound to the result of the mapping (assuming that the parsing was well). E.g if we have the following layer:

```
start(one) = one
```

```
one("1") = 1
```

Then we would get 1 as the result for processing the input: "1".

This means that the evaluation is inside-out. If we have a function f which has some some non-terminals nt1, ..., ntN as parameters. Then first the non-terminals get evaluated (parsed and mapped). If the associated syntactic rule with f matches the parsing input, then f is called with the parameters ntI (I = 1, ..., N) bound to the result of evaluating non-terminal ntI (the terminals are bound to the respective parsing results).

In some cases we refer to classes of terminals. The parameter for such a terminal starts with a capital letter. Cf. [1] for more details. E.g.

```
add(add, "+", Integer)
```

Like syntactic rules, it is possible to extend or reduce functions with the assignment += or -= respectively. If we extend a function with some new alternative and this alternative is already present, then the respective function bodies are merged. E.g.

```
add(self, "+", operand) = <function body 1>
add(self, "+", operand) += <function body 2>
```

is like

```
add(self, "+", operand) = <function body 1 and function body 2>
```

If several non-terminals with the same name are parameters, an index is used to distinguish between them:

```
statement("if", expression, statement1, statement2) = ...
```

## 3.2 Generic Functions

Instead of parametric rules generic functions are used. A generic function accepts any number of parameters:

```
genericFunction = identifier "<" genericParameters ">" "(" parameters ")"
                  functionBody
genericParameters = identifier
                  | self "," identifier
```

A generic parameter can be used in the standard parameters or in the function body. It is instantiated as a non-terminal applied to non-terminal or terminals. The instantiation is only needed for the parameters, which refer to the generic function (in the function body only the name of the parameter has to be specified):

```
parameters = parameter
           | self "," parameter
parameter = identifier
          | identifier "=" String
          | identifier "<" genericArguments ">"
          | String
genericArguments = parameter
                 | self "," parameter
```

E.g.:

```
add<operand>(operand) = operand
add<operand>(self, "+", operand) = self + operand
start(add<Integer>) = add
```

### 3.3 Layers

Regarding the definition of a layer, we use the form:

```
layer = "layer" identifier "{" layerBody "}"
```

layerBody may contain any number of function definitions.

### 3.4 Package

A feature is given by a number of layer fragments which are specified in a package:

```
package = "package" identifier "{" packageBody "}"
packageBody = uses layers
uses = use
     | uses use
use = "use" identifier
layers = layer
      | layers layer
```

A use declaration specified an import for the package. The definitions of the imported package become part of the new package.

## 4. An Example

We will now demonstrate, how the concepts introduced above let us define a modular language.

### 4.1 The Initial Language

We start with a language with the following grammar:

```
start = sequence
sequence = binaryOperator(";", add)
add = binaryOperator("+", basic)
basic = Integer
binaryOperator(op, operand) = operand
                            | self op operand
```

We will provide an interpreter for this language with a single layer. Two packages are used to provide the interpreter. One for dealing with integers and another one, which is the Main package:

```
package Integer {
      Layer Interpreter {
```

```
                add<operand>(operand) = operand
                add<operand>(self, "+", operand) = self + operand
                basic(Integer) = makeInteger(Integer)
        }
}
package Main {
        use Integer
        Layer Interpreter {
                start(sequence) = skip()
                sequence(add<basic>) = print(add)
                sequence(self, ";", add<basic>) = print(add)
        }
}
```

Note, some helper functions are used to provide the interpretation:

+ - addition of integer

makeInteger – takes a string and converts it to an integer

skip – no action

print – prints the given argument to the standard output

## 4.2 Including Another Operator

The next step will be to include the operator * (Multiplication). We first extend the package Integer:

```
package Integer {
        Layer Interpreter {
                add<operand>(operand) = operand
                add<operand>(self, "+", operand) = self + operand
                mult<operand>(operand) = operand
                mult<operand>(self, "*", operand) = self * operand
                basic(Integer) = makeInteger(Integer)
        }
}
```

In Main the changes are:

```
package Main {
        use Integer
        Layer Interpreter {
                start(sequence) = skip()
                sequence(add<mult<basic>>) = print(add)
                sequence(self, ";", add<mult<basic>>) = print(add)
        }
    }
```

### 4.3 Including Variables

For including variables we introduce a new package:

```
package Variables {
      Layer Interpreter {
            declaration<expr>("var", Identifier, "=", expr) =
                  bind(Identifier, expr)
            assignment<expr>(Identifier, "=", expr) =
                  assign(Identifier, expr)
            basic(Identifier) = lookup(Identifier)
      }
}
```

Again we use some helper functions: bind (binds the given identifier to the given value in the current environment), assign (assigns the already defined identifier a new new value in the current environment), lookup (retrieves the value for given identifier in the current environment).

To simplify the structure of Main, we will introduce a package for expressions:

```
package Expression {
      use Integer
      use Variables
      Layer Interpreter {
            expression(add<mult<basic>>) = skip()
      }
}
```

An additional package will hold statements:

```
package Statement {
      use Variables
      use Expression
      Layer Interpreter {
            statement(declaration<expression>) = skip()
            statement(assignment<expression>) = skip()
            statement(expression) = print(expression)
      }
}
```

Now the main package is as follows:

```
package Main {
      use Statement
```

```
		Layer Interpreter {
			start(sequence) = skip()
			sequence(statement) = skip()
			sequence(self, ";", statement) = skip()
		}
	}
```

### *4.4. Including the If Statement*

For the if statement we first add the new datatype boole, which holds truth values:

```
	package Boole {
		Layer Interpreter {
			basic("true") = true
			basic("false") = false
		}
	}
```

This is included in the package Expression:

```
	package Expression {
		use Integer
		use Boole
		use Variables
		Layer Interpreter {
			expression(add<mult<basic>>) = skip()
		}
	}
```

In the package Statement we add the if statement:

```
	package Statement {
		use Variables
		use Expression
		Layer Interpreter {
			statement(declaration) = skip()
			statement(assignment) = skip()
			statement(expression) = print(expression)
			statement("if", expression,
				statement1,
				"else", statement2) =
			    if (expression)
				statement1
			    else
				statement2
```

```
                statement("if", expression,
                         statement) =
                    if (expression)
                         statement
          }
     }
```

Unfortunately, this is flawed, because of the eager evaluation of the components. This means, if we have an if statement its then part (and else part) gets already evaluated before the branching decision.

Therefore, we have to delay the evaluation. We use an anonymous function in the form

fun(<any parameters>) = <function body>

to deal with that. We will have to rewrite our modules as follows:

```
     package Integer {
          Layer Interpreter {
               add<operand>(operand) = operand
               add<operand>(self, "+", operand) = fun() = self() + operand()
               mult<operand>(operand) = operand
               mult<operand>(self, "*", operand) = fun() = self() * operand()
               basic(Integer) = fun() = makeInteger(Integer)
          }
     }
     package Boole {
          Layer Interpreter {
               basic("true") = fun() = true
               basic("false") = fun() = false
          }
     }
     package Variables {
          Layer Interpreter {
               declaration<expr>("var", Identifier, "=", expr) =
                    fun() = bind(Identifier, expr())
               assignment<expr>(Identifier, "=", expr) =
                    fun() = assign(Identifier, expr())
               basic(Identifier) = fun() = lookup(Identifier)
          }
     }
     package Expression {
          use Integer
          use Boole
```

```
            use Variables
            Layer Interpreter {
                  expression(add<mult<basic>>) = add
            }
      }
      package Statement {
            use Variables
            use Expression
            Layer Interpreter {
                  statement(declaration<expression>) = declaration
                  statement(assignment<expression>) = assignment
                  statement(expression) = fun() = print(expression())
                  statement("if", expression,
                            statement1,
                            "else", statement2) =
                        fun() =
                        if (expression())
                              statement1()
                        else
                              statement2()
                  statement("if", expression,
                            statement) =
                        fun() =
                        if (expression())
                              statement()
            }
      }
      package Main {
            use Statement
            Layer Interpreter {
                  start(sequence) = skip()
                  sequence(statement) = statement()
                  sequence(self, ";", statement) = statement()
            }
      }
```

With this new code, evaluation is now top-down. Particularly, the evaluation of the if statement first evaluates the condition before any of the branches are evaluated.

### 4.5 Adding the Datatype String

For the datatype String, we add to a new module:

```
package String {

      Layer Interpreter {

            add<operand>(operand) += operand

            add<operand>(self, "+", operand) +=

              fun() = if (isString(self()))

                        concatenate(self(), operand())

            basic(String) = fun() => String

      }

}
```

Note, because of the form overload we use a type dispatch in add. This also has to be added to Integer:

```
package Integer {

      Layer Interpreter {

            add<operand>(operand) = operand

            add<operand>(self, "+", operand) =

                  fun() = if (isInteger(self()))

                                self() + operand()

            mult<operand>(operand) = operand

            mult<operand>(self, "*", operand) = fun() = self() * operand()

            basic(Integer) = fun() = makeInteger(Integer)

      }

}
```

Finally, we have to include the new module in the package Expression:

```
package Expression {

      use Integer

      use String

      use Boole

      use Variables

      Layer Interpreter {

            expression(add<mult<basic>>) = add

      }

}
```

## 5. Discussion

In the section modularization regarding form we looked at the dimensions of syntactic rules. That illumniated where changes are simple. Particularly alternatives are quite easy to change since side effects are minor. Sequences, the other dimension, have dependencies.

We introduced parametric rules, which let us factor out such dependencies when necessary. They are also useful to abstract from repeating patterns in the syntactical rules.

Partial grammars were introduced to give syntactic modularization a form. The idea is to put only such rules in a partial grammar which are part of feature.

The section on modularization regarding processing mapped the syntactic concepts to aspects of processing. In the center we have the idea that the processing takes places in a function which gains its parameters from the associated syntactic rule. In correspondence to parametric rules we have generic functions, which permit to inject dependencies in the processing.

Layers and packages enable us to group functions with respect to the hierarchy of processing and with respect to the language feature.

An example language was presented, which was extended in various ways. It showed that some extensions require only minor changes in the processing. But, there are also extensions which resulted in huge changes.

The paper presented a clear road for using a feature-oriented approach for MPLD. It is based on principles which are used for the modularization of programming languages. Namely, hiding or making entities local and dealing with dependencies. Unfortunately, like in programming changes or extensions might require deep modifications on the total design.

Typically, for such programming language processors are that several layers are used. E.g. a hierarchy could be as follows: scanner (lexical analysis), parser, type checking and any other checking, and code generation. In our approach the syntactic side is rather hidden. It remains to be investigated, how the approach can be extended to layers, which work rather on the abstract syntax tree.

## 6. References

[1] G. Müller: "*IAS 0.2 - Interaction Server*", http://www.gil-mueller.com/ias/ias.pdf, November 2010.

[2] M. Sperber et. al.: "*Revised$^6$ Report on the Algorithmic Language Scheme*", http://www.r6rs.org/

[3] Ruby, https://www.ruby-lang.org

[4] Scala, http://www.scala-lang.org/

[5] D.E. Knuth: "*Semantics of Context-Free Languages*", Mathematical systems theory, 1968, Volume 2, Issue 2, pp 127-145.

[6] A. van Wijngaarden et al.: "*Revised Report on the Algorithmic Language ALGOL 68", Acta Informatica.* Vol. 5, No. 1/3, 1975, ISSN 0001-5903, S. 1–236,

[7] Kiczales, G.; Lamping, J; Mehdhekar, A; Maeda, C; Lopes, C. V.; Loingtier, J; Irwin: "*Aspect-Oriented Programming*", J. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 1241. June 1997.

[8] M. E. Lesk, E. Schmidt: "*Lex — A Lexical Analyzer Generator"*, Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey 07974 (Oktober 1975).

[9] S. C. Johnson,: "*Yacc: Yet Another Compiler Compiler",* Computing Science Technical Report

No. 32, 1975, Bell Laboratories, Murray Hill, New Jersey 07974

[10] Xtext, http://www.eclipse.org/Xtext/

[11] J. Gosling et.al. : "*The Java Language Specification – Java SE 8 Edition*", http://docs.oracle.com/javase/specs/, March 2014.

[12] C.Szyperski et.al.: "*Component Software Beyond Object-Oriented Programming*", Second Edition, Addison-Wesley, London, 2002.