π – a Pattern Language

Roman Knöll
Faculty of Computer Science
TUD – Technische Universität Darmstadt
knoell@st.informatik.tu-darmstadt.de

Mira Mezini
Faculty of Computer Science
TUD – Technische Universität Darmstadt
mezini@informatik.tu-darmstadt.de

Abstract

Current programming languages and techniques realize many features which allow their users to extend these languages on a semantic basis: classes, functions, interfaces, aspects and other entities can be defined. However, there is a lack of modern programming languages which are both semantically and syntactically extensible from within the language itself, i.e., with no additional tool or meta-language. In this paper we present π as an approach that aims to overcome this lack. π provides an abstraction mechanism based on parameterized symbols which is capable of semantically and syntactically unifying programming concepts like variables, control-structures, procedures and functions into one concept: the pattern. We have evaluated the abstraction potential and the syntactic extensibility of π by successfully creating patterns for the aforementioned programming concepts. π could serve as a tool for designing new experimental languages and might generally influence the view we have on current programming concepts.

Categories and Subject Descriptors D.1.2 [**Programming Techniques**]: Automatic Programming; D.2.10 [**Software Engineering**]: Design; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.2 [**Programming Languages**]: Language Classifications - *Extensible languages;* D.3.3 [**Programming Languages**]: Language Constructs and Features - *Patterns;* D.3.4 [**Programming Languages**]: Processors - *Interpreters;* F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages I.1.3 [**Symbolic and Algebraic Manipulation**]: Languages and Systems

General Terms Design, Languages, Theory

Keywords Patterns, pattern language, semiotics, extensibility, language extension, language design, domain specific languages, macros

1. Introduction

1.1. Motivation and Problem

Language Design The authors were experimenting with new programming languages in the field of naturalistic design. Hereby, the overall *process* of creating new experimental languages seemed inconvenient to us: the syntax has to be specified in a modified way fitting exactly the grammatical requirements of a particular parser; then, helper code has to be written to dissect the parse-tree. Finally, the semantics (in the form of code fragments) are added by assigning a meaning to the nodes of the parse-tree. We found that this process is very tedious and error-prone; thus, we stipulate that there should be a cleaner and easier way to create new (experimental) languages.

Macros and Notation The use of macros is popular and widespread, ranging over different languages from C to LISP. Macros give the programmers more control over the language. Yet, a lot of contemporary programming languages lack a syntactic macro facility. Current languages are "given" to us by some company, language designer or independent project group. Certainly, we would design some features different in some way or introduce new features sooner as they would maybe happen to come with the next release. A macro facility would be a way to mobilize *our* creativity for the overall advancement of a language. Therefore, there should be a clean and easy way to syntactically extend languages.

Domain Specific Languages DSLs play an increasingly important role, in research as well as in practice. This kind of languages – and especially the philosophy behind – addresses the need for well adjusted notations for specific problem domains. Furthermore, DSLs could be a great help to find a common language with both the customers and the developers to specify the requirements of software. We think that domain specific modeling needs an adequate language, i.e., there should be a clean and easy way to design new domain specific languages and notations.

Abstraction Itself Certain ideas of contemporary programming technologies have something very deep in common: they all are the result of (computer) scientists' drive for *abstraction*. Assembler programmers had to deal with ever repeating tasks in their programs. They would introduce labels so that code-fragments can be reused in an abstract way. Then, they would soon use a stack to pass "values" to the labeled code fragment. What followed is the

birth of functions and modules (of functions). Functions in turn led to generic functions and classes, classes led to generic classes and aspects, aspects lead to... ? This whole process is about abstraction. So, our question was: if all advancement in programming languages is abstraction, both semantic and syntactic, why then isn't there a language which is completely dedicated to that paradigm? Would current programming techniques appear as facets of some general abstraction mechanism behind the scene?

1.2. The Essence: Patterns

"A pattern is a plan that has some number of parts and shows you how each part turns a face to the other parts, how each joins with the other parts or stands off [...]. A pattern should give hints or clues as to when and where it is best put to use. [...] some of the parts of a pattern may be holes, or slots, in which other things may be placed at a later time." — Christopher Alexander

What do the scenarios described so far have in common? In each scenario, a programmer defines one or more code fragments, i.e. whole languages or DSLs, macros or functions, and assigns a certain syntax to them, respectively the language- or DSL-syntax, the macro-syntax or the function-syntax (the function signature). This syntax, having parameters, introduces a context that brings variability to the code fragment: the syntax is a *parameterized symbol* with an associated *meaning*, the code-fragment. The particular motivation varies from scenario to scenario. However, the essence of this is the creation of a *pattern*[1]. We denote that as follows:

```
symbol → meaning
```

1.2..1 The Parameterization of Symbols

"A good pattern will say how changes can be made in the course of time. Thus some choices of the plan are built in as part of the pattern [...]. In this way a pattern stands for a design space in which you can choose, on the fly, your own path for growth and change." — Christopher Alexander

We briefly visit semiotics as semiotics is the basis of our work. In our culture, symbols are combined to create more complex symbols with more complex and more concise meanings. For instance, the following sign has the meaning that something is prohibited. It leaves a space to symbolize this unwanted thing or action:

This symbol is a *parameterized symbol* (or short: *syntax*). Together with its meaning it establishes a *pattern*. *Slots* describe the existence of these concrete possibilities within the symbol for customizing them by other *parameter symbols* (or short: *parameters*). The slots in the following mathematical sum symbol are illustrated on the right by dashed boxes:

$$\sum_{i=1}^{10} a_i \qquad \sum_{\square}^{\square}\square$$

The slots at the following example from programming are underlined:

```
while (i <= 10) {
   sum += a[i];
}

while (expression) {
   instructions
}
```

We call the set of all symbols which could be created on the basis of a parameterized symbol by inserting other symbols as parameters into the slots the *concretization* of the parameterized symbol. The *concretization* of a pattern is then the concretization of the parameterized symbol of the pattern if it has one; synonymous notions are: the symbols *realized* by a pattern, the *applications* of a pattern or the symbols *derived* from the pattern. We say that a symbol *matches* a pattern if it is part of the concretization of the respective pattern.

Given a symbol of a pattern, we call the symbols that fill the slots of that pattern the *sub-symbols* (or primary sub-symbols) of the respective symbol, the sub-symbols of the sub-symbols accordingly *sub-sub-symbols* (or secondary sub-symbols), and so on.

The slot in the prohibition sign here is *untyped*, i.e., any symbol can be inserted here whereas the slots in the while-loop are *typed*: only expressions and instructions can be used as *valid* parameter symbols. In general, any predicate could be used to define the type of a slot – as with predicate dispatch in programming.

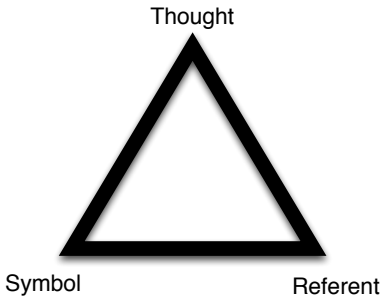
1.2..2 Programming and Semiotics

Programming is a special form of communication which aims at communicating an expected behavior from a sending system, e.g., a person, a computer or a machine, to a receiving system. The programming is *successful* when the receiving system shows the intended behavior. The language used to communicate the behavior is a *programming*

[1] Our notion of a pattern is only indirectly related to the notion of a design pattern well-known from the book "Design Patterns. Elements of Reusable Object-Oriented Software." by Erich Gamma et al.; to some extent those patterns could be modeled in a pattern language like **π**. Neither is our notion of a pattern language related to the BETA-language [24] except for the fact that BETA unifies classes and methods – like **π** does – into the concept of a "pattern".

language. This process is symbolic. The process of agreeing on the meaning of a symbol involves making the actual link between the symbol and the meaning at some time. Any parameterized entity is able to be a container for meaning.

In semiotics, the relation between a symbol and "its" meaning is represented by the semiotic triangle:[1]



In this diagram, the relation between a symbol (syntax) and its referent (meaning) is shown as an imputed relation which is created by our mind, having a thought (pattern). For instance, the sequence of Latin characters "cow" (syntax; only for the speakers of English, of course) refers to the idea of the actual entity "cow" (meaning). This is our "cow-pattern" we have in mind.

1.3. The Goal: A Pattern Language

"This leads me to claim that, from now on, a main goal in designing a language should be to plan for growth. [...] Lisp was designed by one man, a smart man, and it works in a way that I think he did not plan for. In Lisp, new words defined by the user look like primitives and, what is more, all primitives look like words defined by the user!"[2] *— Guy Steele*

Coming back from general semiotics to programming, the important questions are the following: what would the essential features of a pattern-language be? what should a pattern-language look like? We think that a pattern-language should support…

a) ***full syntactical extensibility***: the definition of arbitrary new (context-free) syntax is possible – in order to provide a mean for recording recurrent programming patterns. Speaking abstract, the ideal language realizing what we wish would have the means to associate any syntax with any given meaning – dynamically. Thus, it should have some facility of the form:

```
syntax → meaning
```

b) ***(syntactic) homogeneity***: the integration of new syntax is seamless. The same criteria has been defined for *good* macro languages; Brabrand and Schwartzbach say about the "ideal macro language" [7]: "[it] would allow all nonterminals of the host language grammar to be extended with arbitrary new productions, defining new constructs that appear to the programmer as if they were part of the original language."

c) ***full semantical grounding***: the meaning of every symbol (expression) in the language, except for a minimal predefined core language, is defined / definable by other expressions of the language in a non-circular way. This property holds for current programming languages, too; however, we want to emphasize that the core language, which all other patterns are defined on top, should be *minimal*. This property shall not be mistaken for being able to prove termination of a program.

d) ***reflection completeness***: every entity of the language – and of the interpreter, for example, instructions, functions and concepts, is referenceable from within the language.

e) ***meta-completeness***: there is a meta-level to talk about the language itself (which is provided by reflection-completeness). This meta-level language does neither syntactically nor semantically (in the way it is evaluated) differ from the rest of the language, it actually *is* (syntactically) the same language and does not open a cascade of meta-levels; for instance, LISP is meta-complete. This aspect is related to homogeneity and includes the sub-aspect *full syntactical grounding*: every aspect of the grammar describing the language is expressible in the language itself.

f) ***full semantical extensibility***: this is fulfilled by nearly all programming languages, only machine code having no mean of abstraction, e.g. functions; we add it here for the sake of completeness.

---

[1] The idea of the semiotic triangle has been developed by several philosophers at different times. Therefore, the actually used notions for three constituents of the semiotic triangle vary a lot, although the idea behind them is the same. Charles Kay Ogden and Ivor Armstrong Richards use the trinity symbol/thought/referent Charles Sanders Peirce icon/interpretant/object and Ferdinand de Saussure (Charles Sanders Peirce and Ferdinand de Saussure are two of the founders of modern semiotics) signifiant/signifié/chose. In other systems, the word "meaning" denotes what is called "thought" in the semiotic triangle as described here, since the thought is the meaning of a symbol. However, as in our notation the meaning of the symbol is explicitly stated by the meaning-part of the pattern we decided to call this part meaning instead of denotation or referent. Actually, the construction corresponding to "thought" in the semiotic triangle, is the whole pattern, or at least the part `symbol → meaning`.

[2] This citation and all others of Guy Steel are taken from his speech "Growing a Language" on the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1998.

Such a language could solve the problems we have encountered:

- The design of new languages or DSLs would be eased as these languages could be defined as libraries of an extensible language.
- A special macro facility would not be necessary because syntactic extensions would be available directly in the language.
- A pattern-language would exactly model what abstraction in programming *is*: recognizing a repeating pattern, naming it, declaring a symbol for it and making it context-dependent by parameterization.

1.4. Our Proposal in a Nutshell

A **π**-program is a sequence of instruction symbols (technically, *sentences*), each being a sequence of (Unicode) characters. The sentences are then evaluated (executed) in the respective order. There is only one language construct in **π**: the *pattern*. Patterns are, simply speaking, EBNF-expressions with an associated meaning; a pattern can be easiest understood as a function with a syntactically complex (context-free) "signature". The non-terminal symbols in the signature are then the parameters of the pattern.

A new parameterized symbol (syntax) is defined by the *pattern-declaration instruction* (the parameters are underlined here):

```
declare_pattern ≔ syntax ➞ meaning;
```

This notation reduces the pattern-definition to the absolutely necessary: syntax and meaning. In addition to that, patterns can be named and in order to provide type safety, each pattern can have an (explicit) type. This type defines all places where the occurrence of the respective symbol will be valid in the program. So the complete pattern-declaration instruction looks as follows:

```
declare_pattern name ≔ syntax ⇒ type ➞ meaning;
```

An example: the following pattern declaration defines both the syntax and the semantics of integer-potentiation symbols like, for instance, `174^3` or `2^19`. All constructs used here for defining the meaning of that pattern are predefined in **π** and can for now be interpreted in an intuitive way (see below; `%W-` suppresses the occurrence of whitespaces):

```
declare_pattern
   integer_potentiation ≔
   integer:i %W- "^" %W- integer:j
   ⇒ integer ➞
   {
      int result = i;
      for (int k = 1; k <= j-1; k++)
         result *= i;
      return result;
   };
```

After the declaration of that pattern, its applications can be immediately used at any place where an integer symbol may occur, e.g., in an instruction like `print(174^3);`.

We will see later that the pattern-declaration instruction and any other of the predefined instructions of **π**, the so called "core-pattern-set", are patterns themselves; they are, for instance, instruction patterns like the pattern-declaration instruction, the print instruction and the for-loop or data patterns like the integer pattern or the pattern pattern – yes, a pattern *is* a pattern itself, its syntax is as follows:

```
name "≔" syntax "⇒" type "➞" meaning
```

Thus, the actual pattern-declaration instruction has the following syntax, i.e., it takes a pattern as a parameter:

```
"declare_pattern" pattern ";"
```

The following little code-fragment is a complete **π**-program consisting of two pattern-declaration instructions and two other (predefined) instructions, `print` and `if`, immediately making use of the just defined symbols (the operators ">" and "? :" are predefined patterns):

```
(1) declare_pattern maximum ≔
       "max" "(" integer:a "," integer:b ")"
       ⇒ integer ➞ ( a > b ) ? a : b;

(2) print( max (13^2, 101) );

(3) declare_pattern absolute_value ≔
       "|" integer:i "|" ⇒ integer ➞
       ( i ≥ 0 ) ? i : -i;

(4) if ( max (13^2, |-171|) > 169 )
       print ("yes!");
```

After the first pattern-declaration (line 1), the interpreter would "know" the pattern "maximum". Therefore the second instruction (line 2) is correctly interpreted. After the second pattern-declaration (line 3), the interpreter knows both the user-defined patterns "maximum" and "absolute-value" and can correctly interpret the last instruction-symbol (line 4) which is making use of both of them.

1.5. Our Contribution

With this work, we contribute to the current research in two ways; the following is our hard contribution:

Development of a Powerful Abstraction Concept Based on the conviction that the essence of programming is *abstraction*, we created a programming language that is completely dedicated to that idea – **π**. **π** integrates and abstracts ideas from several different fields such as macros and DSLs into a very expressive *post-paradigmatic language*. Eventually, **π** is an approach of creating not only a semantically but also a syntactically minimal language.

Implementation of a Pattern-Language **π** is the first implementation of a pattern language. Technically, a *pattern language* is a language in which other programming languages are both syntactically as well as semantically reproducible with additional syntactic-extensibility and which fulfills all the criteria defined in section 1.3.

Our soft contribution, namely the philosophy behind π, will be discussed in chapter 7 of this work.

1.6. An Overview of this Work

In the following chapter, we describe the π-language. After that, we briefly discuss our implementation of the π-interpreter. We then evaluate the expressiveness of π by defining several language constructs existing in current programming languages in π. Afterwards, we provide an overview on related ideas. Finally, we conclude this work with a discussion of future work and a résumé.

2. The Language

"But instead of designing a thing, you need to design a way of doing." — Guy Steele

In the preceding chapter we defined the notation of a pattern as a parameterized symbol with an associated meaning and we have seen a pattern-declaration instruction to introduce new patterns to π. Now, we concretely describe how to define patterns and how to use the symbols derived from these patterns.

2.1. The Syntax-Pattern

First of all, π does not have a syntax in a traditional sense: it is not the language which has a syntax, but each single pattern has its syntax. π has predefined patterns (the *core pattern set*; short *CPS*) and user-defined-patterns; the language is the set of these patterns. However, even the syntax of predefined patterns can be dynamically changed – thus effecting all following instructions, i.e., there is no fixed syntax in π at all.

As with typical grammars, the *syntax* of a pattern defines the way the characters have to be assembled in order to represent the respective parameterized symbol of the pattern. The syntax of a pattern is defined like the right side of an EBNF production rule since the latter can be seen as a parameterized symbol with the nonterminal symbols representing the slots of the parametrized symbol. We call our form of EBNF *π-EBNF* what is something like the domain-specific syntax sub-language of π for the definition of patterns. What follows is the definition of π-EBNF written in π-EBNF itself; this is no coincidence as all "syntax" in π is just a symbol derived from the predefined syntax pattern of the core pattern set (the non-terminals, i.e., the slots, are underlined in the rules):

```
syntax = constant_syntax | slot_syntax |
         sequence_syntax | optional_syntax |
         or_syntax | zero_or_more_syntax |
         one_or_more_syntax | bracketed_syntax

constant_syntax = »"« { character } »"« [":" name]

slot_syntax     = %I ( pattern_name ) [":" name]

sequence_syntax = syntax { syntax }

optional_syntax = "[" syntax "]" [":" name]

or_syntax       = syntax { "|" syntax }

zero_or_more_syntax = "{" syntax "}" [":" name]

one_or_more_syntax  = "{" syntax "}" [":" name]

bracketed_syntax    = "(" syntax ")" [":" name]
```

We call the sequence-syntax, the or-syntax, the zero-or-more-syntax and the one-or-more-syntax *multi-slots* as these syntax-rules hold not only one slot but several slots. Nearly all syntax-patterns support an optional name so that they can be referenced when defining the meaning of a pattern. Names have to be unique only on the same level of nesting.

In π, the lexer is completely integrated into the parser as the π-lexer does nothing but classify the input characters into categories such as letters, digits, whitespaces and the like. That way, all literal symbols, e.g., the integer or floating-point-number literals, can be defined by regular π-patterns (integer pattern and floating-point-number pattern in the CPS; or short "int" and "float") and there are no name-clashes at all with "keywords" like it is the case in a lot of current programming languages. π has a notation for different whitespaces, e.g., `%S_` stands for a medium whitespace and `%W-` for no whitespace. Furthermore, π supports syntax for additional formatting as `%I` for italic font and %U for underlined text or `%SUPER` for superscript characters.

2.2. The Meaning-Pattern

We now assign a meaning to the pattern-syntax. The meaning of new patterns in π is defined by making use of symbols derived from already existing patterns, which are, after the startup of the interpreter, only the patterns predefined in the core pattern set (in this section, we will introduce several core-patterns along the way). In π, the difference between "language" and "program" vanishes: "language" is just an alternative name for the core pattern set and "program" denotes the actual symbolic input to the interpreter including the set of user-defined patterns. Let us have a look at the following two examples:

```
declare_pattern = "e" → 2.718281828459;

declare_pattern = integer:i "^2" → i * i;
```

The first of these easy pattern definitions defines the symbol "e" to be an approximation of Euler's number. The second pattern defines the meaning of a squared-integer symbol, i.e., an integer symbol followed by a constant symbol "^2" meaning the product of the integer with itself. The integer-multiplication pattern is a pattern in the CPS: its meaning is predefined as the integer-symbol which is the "conventional result" of the integer-multiplication. Let us have a look at another example:

```
pattern ≔ "count_up_to" "(" integer:i ")" →
{
   int j = 1;
   while (j <= i)
   {
      print (j);
      j++;
   }
}
```

This pattern-definition resembles very strong a conventional method declaration in C-style-languages. This is the point were semiotic theory meets with programming: on the one hand, it is true, this pattern definition *is* like a C-method definition; on the other hand, we do exactly the same as before: we are defining the meaning of a parameterized symbol by making use of already predefined symbols, respectively predefined patterns. So, the general approach is the same, it differs only in the kind of patterns we use: before, we used a constant and an operator symbol, now we use "algorithmic" symbols like the block or the while-loop symbol; both are in the CPS as well as the integer-variable-declaration pattern, the smaller-comparison pattern, the increment pattern and the print pattern:

```
"int" name ["=" integer:initial_value]

"while" "(" expression ")" instruction

integer "<=" integer

integer "++"

"print" "(" symbol ")"
```

As we have seen, the meaning of a pattern can be defined in two ways, we call them the "functional" and the "imperative" style. In **π**, this looks as follows:

```
meaning ≔ functional_meaning | imperative_meaning

functional_meaning ≔ expression

imperative_meaning ≔ instruction
```

Both styles represent two sides of the *same* coin: what else is a "calculation"(-function) than performing actions on symbols, i.e., symbol-manipulation? What else is an instruction than a function having side-effects? **π**-symbols, as any other symbol in programming, encode behavior of a computer system. So, the relevant thing is not *how* behavior is defined but that it is defined *correctly*.

2.2..1 Parameter Binding

We now have a look at how the symbols in the slots of the patterns are made available for the meaning definition, i.e., how they can be *referenced* in the meaning definition. In imperative languages this making available is done by treating the parameters of a method like locally defined variables, being implicitly initialized on a method call. In **π**, *explicit referencing* is similar to – yet not alike – the way just described.

Look at the following pattern which aims at defining a convenient lightweight notation for printing a non-empty list of integers as `print(3,2,8,1)` (the for-loop pattern is a user-defined pattern based on the while pattern):

```
print_integers ≔
   "print" "(" integer:first
   { "," integer:i }:following ")" →
{
   print (first);

   if ( present(following) )
      for (int k=0; k<=size_of(following)-1; k++)
         print( following[k].i );
}
```

The first parameter of the print-integers pattern is referenced by "first". The name of the following zero-or-more symbol of integers, each separated by a comma, is "following". The content of the sequence is referenced using the widespread array-notation: "following[k]". The "."-notation allows to access the sub-elements of multi-slots: "following[k].i". The length of the sequence can be retrieved by the size-of pattern and the present-pattern checks the presence or absence of some optional symbol (both these patterns are in the CPS).

The parameters of less complex patterns can be referenced by their type instead of a name whenever this is possible without ambiguity; we call this *implicit referencing*:

```
"twice" "(" integer ")" → 2 * integer
```

We sometimes need to reference not only the parameters of the pattern but also the sub-symbols of these parameters. We call this kind of referencing *pass-through referencing*. The notation is analogous to the "."-notation for referencing the sub-elements of multi-slots.

Assuming that the integer pattern is defined as follows:

```
integer ≔ non_zero_digit:first_digit
   { digit }:following_digits
```

Then, a pattern to calculate the n^{th} digit of an integer can be defined as follows:

```
declare_pattern ≔ "nth_digit" "(" integer:index
                                  "," integer:i ")" →
{
   if (index = 1)
      return i.first_digit;

   return i.following_digits[index-2].digit;
}
```

2.2..2 Recursion

There are two types of recursion in **π** which correspond to the two main parts a pattern consist of: *semantic recursion* and *syntactic recursion*. The former recursion is the usage of the symbols defined by the pattern in the meaning-definition of the pattern itself, the latter is the usage of the pattern in its own pattern-syntax. The following example for syntactic recursion defines an array pattern which can have multi-dimensional sub-arrays (an array, like all data patterns does not need a meaning as there is nothing to evaluate with pure data, e.g. all literals in **π** are data symbols):

```
array ≔ "[" array | integer
   { "," array | integer }:following "]"
   ⇒ data
```

2.2..3 Reflection

It is an essential part of the philosophy of **π** is that all entities and all functionality of the language (technically, the interpreter) are fully accessible from within the language. In the introduction, we called this property *reflection completeness*. Consequently, all actions the interpreter supports and all data-types it uses are available within the language as predefined patterns, for instance, there is an evaluate pattern (which is similar to the eval-command in other languages; the parsed-symbol pattern is the parse-tree in **π**):

```
evaluate ≔ "evaluate" "(" parsed_symbol ")"
   ⇒ symbol
```

2.3. The Name and the Type

Pattern-names in **π** are predefined symbols as any other, even though they fulfill the important function of – speaking in terms of the semiotic triangle – making the link between syntax and semantic tangible. The name of a pattern must be unique.

π has a strong, dynamic, explicit type system. The type of a pattern defines both where the symbols defined by the pattern may occur in the input, i.e., the replacement scheme of the grammar and the pattern of the resulting symbol possibly calculated by the pattern. For instance, the type of the integer-sum-pattern is integer, too:

```
integer_sum ≔ integer "+" integer ⇒ integer
```

The type of a symbol is the type of the corresponding pattern. The implicit super-type of all patterns is *symbol*.

π supports "higher-order-patterns"; for example, the pattern-declaration pattern takes a pattern as a parameter. Patterns in **π** are just all symbols that can be derived from the pattern-pattern.

2.3..1 Type Safety and Static Semantic

As in **π** patterns – and with the patterns new symbols – can be declared dynamically, there cannot be any static semantic of a **π**-program. Nonetheless, **π** is a dynamically type-checked programming language since all resulting symbols are dynamically checked (parsed) for their pattern. According to the philosophy of **π**, it is open to the programmer if she/he wants to establish a static semantic or not. There are basically three approaches how one can write **π**-programs:

Scenario I — "bad π" The most "dangerous" use of pattern-declarations is to introduce new syntax and semantics based on *unpredictable* user inputs or randomness:

```
declare_pattern new_pattern ≔
   "#"+read(character_string)
   ⇒ integer → random_integer();
```

In this case, the meaning of a possibly following instruction-symbol like, for instance, `print(#abc)` is not at all predictable.

Scenario II — "better π" Even if the language provides the most freedom, a good programmer will put restrictions on herself/himself: she/he would abstain from using that kind of pattern-declarations. Yet, there are still problematic cases, namely *conditional pattern declarations*, i.e., pattern-declarations whose execution depends on dynamic properties of the program:

```
if (i > 5)
{
   pattern signum ≔ "sign" "(" integer ")"
      ⇒ integer → r > 0 ? 1 : (r < 0 ? -1 : 0);
}

print ( sign (i) );
```

In this case, the instruction-symbol `print(sign(i))` is only interpretable if the variable "i" is greater than five.

Scenario III — "good π" Thus, an even better programmer would also abstain from using conditional pattern-declarations.

She/he would declare patterns in libraries (**π**-code-files) which would be loaded at startup and would at best use local unconditional pattern declarations like, e.g., in the following example:

```
pattern divisor ≔ integer:a "|" integer:b
   ⇒ boolean → b % a = 0;

integer r = random_integer;

if ( 3|r )
{
   pattern signum ≔ "sign" "(" integer ")"
      ⇒ integer → r > 0 ? 1 : (r < 0 ? -1 : 0);

   print ( sign (r) );
}
```

In this case, the **π**-program *has* a static semantic as the set of active patterns at any point in the program is fully predictable: patterns in this scenario behave like ordinary functions with a possibly complex signature; after a "renaming" from pattern-syntax to function names and a flattening of the parameter-tree, conventional static type-checking would be applicable.

2.4. The Interpretation

Every sentence that is send to the interpreter, i.e., every symbol in form of a sequence of characters, will be interpreted according to the current state of the interpreter. The state is completely determined by the set of currently active patterns (we call it the *context* of the interpreter): a symbol is accepted if it can be be recognized as the application of one or more of the currently active patterns, all other symbols are rejected as uninterpretable (meaningless). The interpreter accepts only instruction-symbols (short: *instruc-*

tions). All other symbols are rejected as inappropriate sentences.

2.4..1 Dispatch

In general, there are two cases when an input-symbol matches more than one pattern, i.e., the parse-tree is ambiguous: in the first case, the parse trees contains only patterns which are partially homonymous, i.e., basically have the same syntax but differ in the types of the slots. In this case, the dispatcher performs the selection by considering the sequence of slots as members of a lattice and decides according to the order defined on the lattice; this is the most widespread variant of realizing symmetrical multi-dispatch.

Since this is a partial order, only in the case that one interpretation of the input is more specific in every slot, this variant is selected, else the input is rejected by stating an ambiguity. In the second case, when the parse-trees differ in other ways than just being partially homonymous patterns, the input is rejected as syntactically ambiguous which is an indication for poor pattern-design.

2.4..2 The Evaluation

The *evaluation* of a symbol in **π** means evaluating the meaning of the corresponding pattern. The meaning is looked up in the pattern-list of the interpreter. The actual purpose of a meaning is twofold: either the meaning-symbol causes *side-effects* or it does not; either it does calculate/create a *resulting symbol* (short: *result*) or it does not.

If the pattern is predefined, then its predefined meaning is *realized*, i.e., the respective side effects, for instance, printing on the console, take place. If the pattern is *user-defined*, then the interpreter will interpret the associated user-defined meaning-symbol which in turn uses other user-defined and predefined symbols.

The resulting symbol of a symbol is the result of the rewriting of the respective symbol taking place during the evaluation; for instance, the integer-sum symbols do rewrite themselves to the integer symbol that is considered to be the "sum" of these two integers. A result is then inserted into the slots of other symbols in the further process of evaluation.

The way this result is created can be different: the "functional" way would be to "directly" evaluate other symbols in order to come to a result; the "algorithmic" ("imperative") way would be to describe a process which is creating the result. As already mentioned, in **π**, there is no difference between these approaches as "imperative" patterns are evaluated in exactly the same way as "functional" patterns are, by causing side-effects and creating a resulting symbol.

Patterns that do not explicitly calculate a resulting symbol, implicitly are rewritten to the *null-symbol* (we use the Unicode no-character glyph: ␀).

π basically cannot have a strict evaluation strategy as conditional instruction patterns, e.g., the if-then-else pattern, depend on the possibility to *not* evaluate/execute parameters. Consequently, **π** supports lazy evaluation in the form of *explicit evaluation* (call-by-name) and *implicit evaluation* (call-by-need) which together are equivalent to leaving the decision on the concrete evaluation of the parameters up to the programmer instead of enforcing a standard evaluation strategy.

The predefined evaluate-reference(-pattern) (in the CPS) provides a mean to explicitly evaluate parameters:

```
evaluate ≔ "evaluate" "(" reference ")"
   ⇒ reference
```

It would be used in pattern-declarations as follows:

```
unless ≔
   "unless" "(" expression ")" instruction
   ⇒ instruction ➞
   if (!expression) evaluate(instruction);
```

The expression-parameter of the unless-pattern is not explicitly evaluated but implicitly: whenever a reference to a parameter is used without the evaluation-pattern then the evaluation will take place implicitly. In this case, **π** uses the call-by-need strategy: once evaluated the results of the parameters are buffered in order to prevent multiple, possibly time-intensive re-evaluations of the parameters. The following example uses only the implicit evaluation variant:

```
integer_potentiation ≔
   integer:i "^" integer:j
   ⇒ integer ➞
{
   int result = i;

   for (int k = 1; k <= j-1; k++)
      result *= i;

   return result;
}
```

2.5. The Core Pattern Set

"If I want to help other persons to write all sorts of programs, should I design a small programming language or a large one? [...] I should not design a small language, and I should not design a large one. I need to design a language that can grow. I need to plan ways in which it might grow – but I need, too, to leave some choices so that other persons can make those choices at a later time." — Guy Steele

We have seen already a lot of the patterns built into the core pattern set. **π** is a semantically *and* syntactically minimal language because like the λ-calculus' abstraction and application constructs, **π** would as well need only these two language constructs (predefined core patterns), only the abstraction would have to be extended by a possibility to define new parametrized symbols – i.e. the abstraction would be the pattern-declaration-pattern – and the application

would then interpret symbols in the context of these patterns.
However, as the focus of this article is to describe a pattern-language as such and not its minimality in particular, we do not argue or even prove the necessity of patterns to include in the CPS or not. We therefore did define the CPS in an informal way by choosing the syntax and the semantics that most programmers are familiar with from other programming languages (LISP- or Pascal-style syntax could be added to **π**, too). There is not the *one* core pattern set. Several sets of core patterns would be adequate for a realization of the **π**-language since patterns can be defined on top of other patterns; the only absolutely unavoidable pattern is some pattern-declaration-pattern. The CPS should be interpreted rather as a proposal and an outline, mainly used for the definition of the patterns in the evaluation of this work, than as a fix definition. We provide some examples of the categories the patterns in the CPS are grouped by.

Declaration (Meta-)Patterns Considering the remarks made so far, it is not an exaggeration to say that the whole concept of **π** is based on two patterns, the pattern-pattern and the pattern-declaration pattern – which exactly reflect the philosophy of **π**:

```
pattern ≔
   name "≔" syntax "⇒" type "→" meaning ⇒ data

pattern_declaration ≔
   "declare_pattern" pattern
   ⇒ instruction
```

Mathematical Operator Patterns The most common mathematical operators are predefined in infix notation both for integers and floats and the combinations of them:

```
integer_multiplication ≔
   integer:a ("*" | "·") integer:b ⇒ integer
```

Logical Patterns The basic logical conjunctions are predefined, as well with mathematical notation:

```
logical_and ≔
   boolean:a ("∧" | "and" | "&") boolean:b
   ⇒ logical_operator

logical_operator ⇒ boolean
```

Symbol Manipulation Patterns Patterns used for manipulating symbols, i.e. in **π**, sequences of characters, play a crucial role in **π** as symbols can be seen as the "instances" of patterns.

The whole language is based on the principle of symbol manipulation, or being more precisely, every programming language is based on that principle, **π** just doesn't hide its evidence.

For instance, character-string concatenation can be done either by the "+"-operator or by a reduced space:

```
character_string_concatenation ≔
   character_string:a (%S- | "+")
   character_string:b ⇒ character_string

character_in_string ≔
   "(" positive_integer ")" "th_character_of"
   "(" character_string ")"
   ⇒ character
```

The properties of a pattern can be changed by the predefined rewrite pattern. The following application of the rewrite pattern, for instance, changes the syntax of the print pattern; it actually performs a sort of renaming:

```
rewrite (print.syntax, "write" "(" symbol ")");
```

Control-Flow Patterns **π** provides several typical control-flow patterns, among them, for instance, the if-then-else pattern:

```
if_then_else ≔
   "if" "(" expression ")" instruction
   [ "else" instruction:else ]
   ⇒ control_flow_pattern
```

Data Patterns The typical data-patterns like integer, float or boolean are predefined in **π**:

```
boolean ≔
   "true" | "false"
   ⇒ data
```

All other more complex data patterns like lists, matrices, maps and tables can then be defined on top of these basic data patterns.

Ontological (Meta-)Patterns Type hierarchies can be seen as ontologies having a type-entity and a subtype-relation. Consequently, all patterns, including, e.g., instruction patterns, can be classified by their type:

```
integer_unequality_comparison ≔
   integer:a ("≠" | "!=") integer:b
   ⇒ comparison_operator

comparison_operator ⇒ boolean_operator
```

The is-operator pattern can be used to find out if a pattern is a sub-pattern of another pattern:

```
is ≔ type "is" type
   ⇒ boolean_operator
```

The top-pattern of the ontology in **π** is the symbol-pattern. It has two immediate children: the instruction pattern and the expression pattern. Patterns having side-effects and control-flow patterns should be defined as instructions; patterns calculating a resulting symbol and data patterns should be defined as expressions.

Communication Patterns (I/O-Patterns) In the world of π, there is nothing but symbols and patterns. So, the communication mechanism will comprise the standard means to emit and receive symbols:

```
print ≔ "print" "(" symbol ")" ⇒ instruction
```

(World-)Knowledge-Patterns The world surrounding a system does contain a lot of information and knowledge that is in most cases relevant to the system. Among these "world"-patterns are those that define the current date, time and location of the system:

```
current_date ≔ "current_date" ⇒ date
```

Context (Meta-)Patterns We have used references to define the meaning of patterns. References, in π, like everything else (reflection-completeness), are predefined patterns:

```
explicit_reference ≔
   name [ "[" integer:index "]" ]
   [ "." reference ]
```

Interpreter (Reflective) (Meta-)Patterns We have already mentioned some of the patterns demanded by the principle of reflection completeness in the section on reflection. Other patterns in the same context are, for example:

```
parse_instruction ≔
   "parse" "(" symbol ")"
   ⇒ parsed_symbol
```

A parse-tree is encoded by the parsed-symbol pattern: a parse-tree could be seen as the original input symbol with (a lot of) additional information, especially the linkage between the read characters and the patterns they belong to. The interpret-pattern is a user-defined pattern:

```
interpret ≔
   "interpret" "(" symbol ")" ⇒ symbol
   → evaluate ( parse ( symbol ) )
```

3. Implementation

We now describe an implementation of the π-language. We emphasize once more the strict separation of the (semantics of the) π-language and the implementation of an interpreter for this language. The π-language does explicitly not include any technical features of any underlying system, for instance, something like a processor-ticks pattern or memory-management patterns; actually, it is a pure symbol manipulation mechanism.

Our interpreter is completely implemented in Java 6. Of course, the central and most important element of the implementation of a syntactically extensible language is the parser. For our purpose, we re-discovered a type of parsers which were used since the 70s almost exclusively in computational linguistics but not in programming: the chart parsers.

3.1. The Parser

Our parser is based on the modified Earley-parsing algorithm [2] with several little additional improvements from our side. The Earley-parser has three main advantages in the context of using it for a pattern language:

- It updates relatively fast on a change of the grammar rules while still having acceptable parsing speed; this is crucial for a syntactically extensible language.
- Secondly, it can parse any context free grammar; therefore programmers do not have to modify their pattern syntax in order to comply with the requirements of a specific grammar, as, e.g., the restriction to LL-grammars concerning ANTLR.
- Thirdly, the Earley-parser returns all possible readings of an input sentence; thus, the dispatcher can then decide in a later stage which reading is the one in the respective context (see the following section).

The algorithm has a complexity of $O(n^3)$ concerning the length of the input (O(n) in case of a LR(k)-grammar). However, a complete description of the algorithm is by far beyond the scope of this article.

We have furthermore developed an "Earley-parser generator", i.e., a small framework to instantiate a clean Earley-parser with predefined rules (the ones for the predefined patterns) written in EBNF.

3.2. Dispatch

In case that an input-symbol is homonymous, i.e., the parser returns several parse trees, the parse trees are compared then if they are just partially homonymous or differ in other more complex ways. In the latter case, as described in the section 2.4.1, the input is rejected. In the former case, the type-distance of all sub-symbols to the slots is calculated from the parse tree and they are compared pairwise as if in a lattice.

3.3. Evaluation

In a bootstrapping-process the predefined patterns are loaded in several steps, considering the dependencies between them as some predefined patterns are already based on other more basic predefined patterns, e.g., the predefined integer-sum pattern uses the integer pattern which in turn uses the digit pattern. The evaluation of predefined patterns is straightforward: the meaning of these patterns is defined by pure Java-code which is then executed. The evaluation of user-defined-patterns, on the other hand, is realized by evaluating the meaning-symbol of the respective pattern.

For example, the meaning of the following pattern is evaluated by evaluating the integer-multiplication symbol:

```
square ≔ integer:i "^2"
  ⇒ integer → i * i
```

Of course, as π is semantically grounded, every evaluation of a user-defined pattern will end up in the evaluation

of predefined patterns. In both cases, the evaluation of predefined and of user-defined patterns, the parse-tree of the symbol to evaluate is provided as a context for the resolving of the parameter references. For instance, the input-symbol 3^2 would result in the following parse-tree provided to the square pattern (we use an XML-style pretty-print for the parse trees):

```
<square-symbol>
   <integer-symbol name="i">
      <non-zero-digit name="first_digit"
       literal="3">
   </integer-symbol>
   <literal="^">
   <literal="2">
</square-symbol>
```

The reference "i" could now be resolved by looking it up in the parse-tree. **π** provides dynamic type checking as the resulting symbols of all patterns are checked if they match the indicated type; for example, the **π**-interpreter would try to parse the result of the square-pattern as an integer symbol. If this failed, the interpreter would throw a type error.

3.4. Implementation Status

The current implementation of the **π**-interpreter has some minor restrictions in implementing the **π**-language. These are as follows:

- In the current implementation the reference pattern differs slightly from the way as described here; all references start with a "$". The usage of an ordinary name pattern causes the Earley-parser to generate too many possible results, which could easily be sorted out at a later stage, but the sheer creation of these results temporarily consumes too much memory; presently, we think about working directly on the parse-chart in order to tackle that issue.
- Nested pattern declarations as described in section 2.3.1 ("Type Safety and Static Semantic") are currently disabled since the parser would as well take too many resources to process these (anyway deprecated) constructions; however, we think about introducing a kind of *lazy parsing*, i.e. a multi-stage-parsing which would then be controlled directly by the **π**-language, respectively by the programmer.

4. Evaluation

"It is good to design a thing, but it can be far better (and far harder) to design a pattern. Best of all is to know when to use a pattern." — Christopher Alexander

We evaluate the expressivity of **π** by revisiting several concepts of current programming languages, at the same time shedding light on these concepts from a **π** perspective. The evaluation is structured along the level of abstraction the analyzed patterns have, starting from simple notation-definition and ending with full programming languages realized in **π**. We limit our set of examples here to a few representative ones from each category. The interested reader can find more on our website (pi-programming.org).

4.1. Use-Case I: Language Constructs

"My point is that a good programmer in these times does not just write programs. A good programmer builds a working vocabulary. In other words, a good programmer does language design, though not from scratch, but by building on the frame of a base language." — Guy Steele

Language constructs are usually introduced by version changes of programming languages as, for example, the for-each loop and variadic methods came to Java; or they are introduced by completely new languages, as, e.g., closures came with LISP. In **π**, new language constructs are introduced by new patterns built on top of the core language, including expressions, instructions and data patterns.

4.1..1 Expression Patterns

Most contemporary programming languages lack of convenient notation for specific problem domains, for instance, for mathematical or technical notations. Newer languages such as Fortress [19] address this issue.

The definition of symbolic aliases for functions are a major motivation for syntactic extension (see the related work). In **π**, for instance, a square-root-pattern could be defined as follows:

```
"√" %W- number ⇒ float
   → square_root (number)
```

The square-root symbol could be used then in expressions ("ℕ" is predefined as a synonym for the positive-integer pattern):

```
if (n ∈ ℕ ∧ n ≥ 0) return √n;
```

Another example: in a lot of programming languages, ordered sequences of numbers have to be expressed in the following bloated way:

```
(i >= 10) && (i <= 20) && (j > 20) && (j < 40)
```

With the help of a user-defined operator-chain pattern the same expression can be written in the following intuitive way:

```
10 ≤ i ≤ 20 < j < 40
```

4.1..2 Control Structure Patterns

Control structures such as loops can be defined as patterns:

```
control_structure ⇒ instruction

loop ⇒ control_structure
```

The most basic of all loop-constructs is the one that performs a given action a fix number of times:

```
do { print ("hello!"); } (10) times
```

The corresponding pattern looks as follows (the execute pattern is a synonym for the evaluate pattern):

```
do_times_loop ≔
   "do" instruction "(" integer:times ")" "times"
   ⇒ loop →
{
   for (int i = 1; i ≤ times; i++)
      execute (instruction);
}
```

4.2. Use-Case II: Meta-Constructs

"Meta means that you step back from your own place. What you used to do is now what you see. What you were is now what you act on. Verbs turn to nouns. What you used to think of as a pattern is now treated as a thing to put in the slot of an other pattern. A meta foo is a foo in whose slots you can put foos." — Guy Steele

With the term "meta-construct", respectively *meta-pattern*, we denote constructs whose main purpose is in defining other patterns or helping with that. Every semantically extensible programming language must have concepts for defining the entities of the language. In the case of Java, for instance, these are, among others, class- and method-declarations. By now, in **π** we have seen only one construct to define other patterns: the pattern-declaration pattern. Many other constructs could be defined on top of that.

We can define alternative – possibly reduced – notations for repeating pattern declarations, e.g., for the declaration of "functions" in **π**:

```
declaration ⇒ instruction
```

Functions – being so to speak the first real abstraction in programming – are an ever occurring pattern in programming borrowed from mathematics. This kind of abstraction can be defined in **π**, as well, in the following in the widespread C-style (we use the "»" and "«" as quotation marks (CPS) so that the upper quotation character can be used in a readable form within the declaration-string):

```
function_declaration ⇒ declaration


variable_name ⇒ name
```

```
c_style_function_declaration ≔
   type name
   "("(type variable_name):first
      {"," type variable_name}:parameters ")"
   block:meaning
   ⇒ function_declaration →
{
   character_string pattern_string =
      »"« + name + »" "("«;

   if (present (first))
      pattern_string += slot(first.type) +
         """:"" first.variable_name;

   if present (parameters)
      for (int p=0; p≤size_of(parameter)-1; p++)
         pattern_string += ""","" +
            slot(parameters[p].type) +
            """:"" parameters[p].variable_name;

   pattern_string += """)"" "⇒" type "→" meaning;

   declare_pattern (pattern)pattern_string ;
}
```

Outgoing from the function-declaration, a pattern-declaration instruction is assembled on a string-basis. This looks similar to "macro"-programming; however, in **π**, there is no "macro-expansion" but just the usual evaluation of patterns: the assembled declaration string will be parsed as such and immediately executed.

We can now express the simple max-operator pattern we have previously defined in a more convenient way ("int" is defined as another synonym for "integer"):

```
int max(int a, int b)
{
   return a > b ? a : b;
}
```

4.3. Use-Case III: Libraries and Frameworks

In **π**, a library or a framework – we use these terms here synonymously as both are extensible – is a considerable set of interacting patterns serving a common purpose, i.e., a domain specific language. The advantage of **π** in library design is that the syntax of the entities in the library can be developed according to the purpose of the library, for example, in the domains of logging, error-handling, test-driven-development, software metrics, GUI-design, web-development or data-access.

A direct embedding of SQL-instructions would be very beneficial in a programming language as a lot of applications require persistency. We exemplary define here the sql-command "insert" assuming that the connection to the database is realized by a pattern `send_sql_command(database, sql_command)` and that the patterns "sql-value", "sql-column-name" and "sql-table-name" are already defined (the check-pattern stops the execution in case of an error; the this pattern is a reference pattern referencing the symbol itself, in this case, the whole sql-insert-statement):

```
sql_instruction ⇒ instruction
```

```
sql_insert_statement ≔
   "INSERT" "INTO" sql_table_name
   "(" sql_column_name
      {"," sql_column_name}:column_names
   ")" "VALUES" "("
      sql_value {"," sql_value):values
   ")" ";"
   ⇒ sql_instruction
   →
{
   check (size_of(column_names) = size_of(values),
      "Wrong number of values!");

   send_sql_command (current_database(), this);
}
```

If all CRUD-sql-commands were defined, we would be able to seamlessly integrate sql with **π**:

```
INSERT INTO people VALUES ("Alfred", "Wissel"),
   ("Julika", "Häuser");

for_each_in (SELECT * FROM people)
   print (current.forename);

UPDATE people SET surname = "Häuser"
   WHERE forename = "Alfred" AND surname =
"Wissel";

DELETE FROM people WHERE (forename ="Alfred");
```

The fact that sql-commands are elements of the language might help to reduce problems concerning sql-injection-attacks, too, as incoming character-strings denoting sql-commands can be parsed already as specific sql-command instead of generally interpreting the – possibly harmfully modified – commands.

4.4. Use-Case IV: Full Languages

"A language design can no longer be a thing. It must be a pattern – a pattern for growth – a pattern for growing the pattern for defining the patterns that programmers can use for their real work and their main goal." — Guy Steele

In **π**, the anyhow hard to define difference between a "language" and a "library" vanishes completely. Usually, "languages", in contrast to "libraries", define syntax in addition to the semantics they come with. As in **π** every library defines syntax, as well, every library is a language.

As a proof of concept of the expressiveness of **π** we define the λ-calculus in a straight-forward way, both semantically *and* syntactically. As mentioned before, the λ-calculus and **π** have a lot in common: the former is a *semantically minimal* programming language and **π** is a *semantically and syntactically minimal* programming language.

```
λ_calculus ⇒ language
```

The λ-calculus has three types of expressions:

```
λ_expression ≔
   λ_variable |
   λ_abstraction |
   λ_application
   ⇒ λ_calculus
```

A λ-variable is represented by a lowercase-name:

```
λ_variable ≔ [lowercase_letter]
```

The abstraction and the application look as follows:

```
v is a λ_variable;
e, e1 and e2 are λ_expressions;

λ_abstraction: (λv.e);

λ_application: (e1 e2) ≔
   { e1 is λ_abstraction :
      β_reduction(e1.expression, e1.variable,
e2),
     otherwise : this;
```

The above description is the original code-fragment in **π** which defines the λ-calculus by making use of a naturalistic-slot-definition pattern ("*x*, *y* and *z* are *v*") and a programming-by-example-definition pattern ("*name*: *expression*"). In addition to that, a case-pattern (user-defined) and a this-reference (CPS) are used. The complete implementation of the β-reduction-function can be found on our website.

Considering all properties of a pattern-language, we see a closer relation of **π** to languages like Scheme or Haskell. However, object-oriented languages can also be modeled with **π**. We currently work on a (prototypic) language incorporating some of the very basic features of Java, called πicoJava; more information can be found on our website.

4.5. Use-Case V: Meta-Languages

"In a way, a language design of the old school is a pattern for programs. But now we need to "go meta." We should now think of a language design as a pattern for language designs, a tool for making more tools of the same kind." — Guy Steele

The concept of patterns allows to interpret languages as sets of patterns. That way, with a *language workbench*, new languages could be created by composing features (occurring in different other languages). Languages, respectively language families, could, as well, be designed in a generic way by either leaving parts of the syntax open for exchange or parts of the semantics, for instance, in case that different implementations of a language for different machines should be developed – for example, one implementation for a multi-core system and another one for a micro-computer.

Languages could inherit from each other by sharing a common set of patterns (e.g. C++ "extends" C), possibly with the same syntax but with rewritten pattern-semantics. In this context, a whole language can be seen as a parametrized pattern. **π** could be used as a *super-language* for (some) of current programming languages and as a *hyper-language* in the sense of being a super-language with the additional capability to be syntactically extensible.

In this context, a pattern language behaves not only like a parser-generator but like a whole integrated *language generator*. Product line management could become a part of the language itself rather than being an outside mechanism.

5. Related Ideas

The idea of a pattern language touches several fields of contemporary programming which are best captured by the term "language oriented programming". This term has been initially described in [32] by Martin Ward. It splits the development process in three stages: the design of a problem oriented very high level programming language, the implementation of the system with this language and finally the implementation of a compiler for this language. This process should be performed recursively. Ward calls language oriented programming "middle out development" in contrast to top down or bottom up development and their combination in outside in development. Völter [31] adds interesting motivation for language driven development from a practical point of view.

In general, two movements can be noticed: the "old" movement in the 60s focussed mainly on macros as a mean of syntactic extension, the new movement, starting in this century is driven by DSLs, by providing extension tools for "big" languages like Java, and the desire for composing several languages in one, thereby creating a *superlanguage* (the term "superlanguage" was coined by Christopher Diggins [16]). Finally, term rewriting calculi and adaptive grammars, standing a bit aside, are more formal approaches to the same goal.

However, none of the works – even the closest ones about Katahdin [23] and XMF [22] sticking very strong to the object-oriented paradigm – draw the attention on perceiving programming itself as a process of designing and using patterns. **π** is a new language whose goal is not extension but which is built exclusively on the principle of patterns – realizing all features of a pure pattern language. **π** is *post-paradigmatic* in the sense that it does not favor any special programming paradigm (except for "pattern oriented programming" if itself interpreted as a paradigm). Only syntactic homogeneity and sometimes full reflectivity seem to be a widely accepted goals in the related work.

5.1. Syntactically Extensible Languages

Other used terms used in this field are *grammar-oriented programming* or *syntax-directed languages*. This kind of programming languages is closest related to the concept of a pattern language as the goal of XMF [22] is: "An idealized superlanguage provides control over all aspects of representation and execution. A superlanguage can be extended with new features that make it easy to represent concepts from the application that a customer would understand. [...] Each new feature that is added to a superlanguage has a description of how it should execute and how it integrates with other features.".

Katahdin [23] is an object-oriented imperative (super-)language; it is designed especially for the aspect of combining several programming languages in one. Katahdin, like **π**, supports the definition of syntax and semantics together as one unit. It hereby realizes the core of a pure pattern language, however, it is still attached completely to the object-oriented paradigm. Neither does it provide full reflectivity or meta-completeness. Despite that, the work on Katahdin is among the most interesting works in this field and the closest relative of a pattern-language.

XMF [22] is a syntactically extensible object-oriented language. XMF focusses very strong on the aspect of being a superlanguage for the use in multi-language projects. [15] describes how to syntactically extend Java on the basis of XMF. We find it disadvantageous that XMF forces the programmer to separately define syntax and semantics: syntax is defined in the Grammar-construct and semantics is implemented in a method "desugar" with a class implementing a "Performable"-element. Nevertheless, XMF is one of the most advanced approaches in the field of syntactically extensible languages.

Logix [25] – currently in alpha stadium – and the eXtensible Language (XL) [35] allow for the definition of user-defined syntax for operators (XL allows for a basic nesting by predefined "block"-symbols, too). Proof assistant systems like Isabelle [20] allow as well for syntactic extensions on the operator-level.

5.2. Language Design Tools

In contrast to syntactically extensible languages, tools for language design, mainly aim at extending existing languages with new syntax. They divide in two groups: general purpose compiler-compilers provide a framework for the design of new languages, possibly with an already written implementation of a widespread language like Java; extension tools / facilities on the other hand aim at extending a specific existing programming language. Both approaches can be used in the context of domain specific modeling. In a pattern language, there is no inherent difference between a library or a DSL as both syntax and semantics are defined in a library. A pattern language shares as well a common goal with model-driven development: both aim at giving the programmer a powerful abstraction mechanism; however, the way, this is realized is very different: whereas model driven development advocated on creating a hierarchy of ever more abstract (graphical) modeling languages, a pattern-language strives for meta-completeness and primarily keeps to textual modeling.

5.2..1 Extension Tools / Facilities

The Java Syntactic Extender (JSE) as described by Bachrach and Playford in [3] is a macro-facility for the Java language. Code used in macros is quoted with a special syntax of an opening "#{"-bracket and closing "}"-bracket. References to code-fragments to be inserted are then done by the identifier of the respective fragment preceded by a question mark. Code fragments are then evaluated in a multi-staged way. JSE provides only a limited syntax extension mechanism, for instance, it allows to extend only a bunch of surface syntactic nodes of Java. This is intended because the system focusses on the Java language and therefore considers a lot of usability issues.

In [9] on MetaBorg / Stratego Bravenboer and Visser describe how Java can be extended by domain specific languages. They provide examples for direct XML repre-

sentation and a GUI-language which are realized by a macro-like-rewriting of Java-code to Java-code. MetaBorg aims at bringing the concepts of APIs to programming languages.

Ometa/CLOS [33] is an object-oriented language for pattern matching and is similar to executable grammars like newspeak [8] – which is a general purpose language supporting the expression of parser combinators. Like [1] it is based on a parsing expression grammar (PEG) [18]. This is a relatively new grammar formalism which is similar to context-free grammars but supports syntax predicated and enforces – by rule prioritizing – a unique parsing result. We do not use a PEG as an ordering of patterns is not intended in our concept of a pattern language, all patterns are "equal" by default. Besides the higher expressivity of PEGs, the programmer has to take care of a lot of the syntactic details of the rules in a skillful way in order to guarantee the correct ordering of the rules.

[1] is the most recent work in the field and describes the introduction of macros to Fortress [19]. This work has a lot in common with Katahdin [23] and XMF [15]. As this work concentrates on the extension of an existing language rather than the definition of a new language it explicitly strives for syntactic homogeneity but does not aim at reflection- nor meta-completeness. However, the work describes an interesting approach of how to organize "grammars" in modules which may (multi-)inherit from each other.

5.2..2 Compiler Compilers

The Jakarta Tools Suite (JTS) [5] aims at creating domain-specific languages for existing programming languages. JTS consists of the tools "Jak" and "Bali". The former is a meta-programming extension for Java, the latter a tool for composing grammars.

JastAdd [21] is a compiler compiler based on aspect-oriented modules. JastAdd supports as well a rewriting of the AST for integrating new syntax into the language. The JastAdd Extensible Java Compiler [17] is a full implementation of Java in the JastAdd-framework.

Theoretically, every parser is syntactically extendable, it just has to be regenerated every time the grammar changes. In practice, for most predictive parsers, for instance the LL(k)-parsers generated by ANTLR, this is not realizable; they parse very fast though but take a long time for the generation of the automaton. "Rats!" [26] is a parser generator for Java which integrates, like the **π**-parser, lexing with parsing. It uses a parsing expression grammar.

5.3. Metaprogramming

"Metaprogramming" is a very extensively used term as it comprises macro-facilities, generative programming, multi-stage-programming, generic programming and meta-object-protocol implementations. We focus here on macro and multi-stage-programming as generic programming is widely known.

5.3..1 Macro Facilities

The term "macro" is used in various ways, as well. It mainly refers to textual macro-processors like the C-preprocessors or TeX or the syntactic macro-processors like the macro-facilities of LISP, Scheme or Dylan [4]. One of the earliest works on macros was [13] by T. E. Cheatham in 1966. Brabrand and Schwartzbach [7] give an excellent survey on macro languages comparing eight representative systems using 32 properties.

π is neither a lexical macro processor as it does not operate on a purely textual basis nor is it a syntactic macro processor since there is no "pre-evaluation"-phase or "macro expansion phase" as it is called concerning LISP. From the point of view of **π**, a "macro expansion" is like any other evaluation. **π** does not make a difference between "normal" data types and data which represents evaluateable code: *every* symbol is evaluateable.

However, conceptually, **π** has a lot in common with LISP: both are minimal languages, concerning syntax and semantics and concerning its main concepts: lists and patterns; actually, original LISP uses trees, lists are a special data-structure defined on trees, **π**, too, uses trees, namely, syntax-trees; both are homoiconic programming language, i.e. "code" is just a special form of data. **π** is a little bit like "LISP with syntax" but without a macro-expansion phase, therefore no problems concerning hygiene occur in **π**.

The metamorphic syntax macros [7] are designed to extend the syntax of a host-language, in the paper "<big-wig>" [6], an interactive-web-service-language. Brabrand and Schwartzbach make a difference between normal kind of macros and metamorphic macros (hence the name): the latter differ in that way from the former that they can be used in the definition of other macros. In the program, the macros are identified by their identifier. Ambiguities are resolved, among other strategies, by declaring greedy parsing as the standard parsing rule.

5.3..2 Multi-Stage Programming

MetaML [27] is a multi-stage programming language. MetaML supports "higher-stage expressions" where the stage of any piece of code is determined by the number of surrounding brackets. This way, code can be treated as usual data, created and executed in a later stage. A pattern language is only multi-stage in the sense that assembled symbols can be reinterpreted as instructions an then be immediately evaluated. However, the evaluation of whole code-pieces cannot be delayed to a later stage. MetaOCaml [28, 29] is another example of multi-stage programming.

5.4. Term Rewriting Calculi

The "recursive functions algorithmic language" (REFAL) described in [30] by Valentin Turchin is a functional programming language that directly implements term rewriting mechanisms by providing two basic mechanisms: pattern matching and substitution. In general, a Refal program consists of functions in arbitrary order (the starting function is marked with the keyword `$ENTRY`), which in turn consist of sentences. A sentence is the combination of a pattern and an expression, which will be returned as the function result if the pattern matches. So, a Refal function can be compared to a set of π-patterns. In addition to that, π is related to Refal insofar that the core principle of both languages is pattern matching; however, Refal does not aim at syntactic extensibility.

5.5. Adaptive Grammars

Adaptive grammars – also called modifiable, extensible, dynamic or adaptable grammars or dynamic syntax – are a formalisms based on the common grammar formalism but with the extension of dynamism: there are grammar rules that can change the rule set of the grammar during "parsing". This way, adaptive grammars even become turing-complete programming languages. Originally, adaptive grammars, were invented very early in 1963, see [12], afterwards reinvented several times. [14] gives a very good overview on these works.

The Universal Syntax and Semantics Analyzer (USSA) as described in [11] is a formalism and a parser and a more recent work in this field. The approach is based on a bottom-up modifiable grammar as described in [10]. In USSA, rules are declared in YACC-style and are organized in "clusters". A cluster is basically a set of rules which are then – at invocation of the cluster – added or removed from the set of current rules. Clusters can be used to define whole languages. In contrast to π, the semantics of USSA patterns is scattered within those patterns. It is hard to declare patterns in a consistent way since potential (syntactic) sub-patterns have to implement semantics, as well. Sub-patterns then communicate with their super-patterns by global attributes. The USSA stays very technical instead of exhibiting more the idea of patterns.

6. Future Work

Parser Performance The mentioned restrictions in the current implementation have to be relaxed and the performance of the parser should be addressed – in general, we see a great potential for optimizations here. Other parser and grammar formalisms should also be considered concerning their usefulness and applicability in the context of a pattern language.

Integration There are several ways of how π could integrate with existing languages: as a super-language, as a domain specific language for the definition of pattern within these languages or by directly importing source-files of these languages into the (Java-)π-interpreter.

Debugging Pattern languages require a special treatment of exceptions and errors because the parser has much more importance in a pattern language; however, it often does not provide sufficient information. So, there should be ways to generate useful hints from this side.

Text-Formatting In order to realize the full potential of a pattern language, IDEs should have an integrated support for formatted text editing like current text processors and elaborated editors do. In addition to that, an appropriate corresponding markup file format has to be developed.

Pattern-Sharing There should be a mechanism or a community platform to search, exchange and share patterns. This has to be done in a slightly different way than current – neither optimal – sharing of source code because patterns can only be identified by their name as it is hard to query on the syntax. An open pattern library would maybe be organized in an ontological way, having patterns be tagged or provided with additional descriptions.

7. Résumé and Conclusion

"Well—there may be one other way, which is to use a large, rich programming language that has grown in the course of tens or hundreds of years, that has all we need to say what we want to say, that we are taught as we grow up and take for granted. [...] But that is not where we are now. [...] I hope that we can, in this way or some other way, design a programming language where we don't seem to spend most of our time talking and writing in words of just one syllable." — Guy Steele

π fulfills all criteria of a pattern language: π is *fully semantically and syntactically extensible* in a *syntactically homogeneous* way because there is no inherent difference in the application of predefined and user-defined patterns. With its minimal approach π is *fully semantically grounded* – as every evaluation of a user-defined patterns ends up in the evaluation of a predefined pattern – and *reflection complete* as all functionality of the interpreter is accessible from within the language.

π is as well *meta-complete* as the reflection language is identical with the core language and *fully syntactically grounded* as even the syntax of patterns is represented as syntax-patterns.

This makes π *post-paradigmatic* in the sense that π directly realizes the process of abstraction in a macro-like fashion: stop copy & paste or using IDE-source-templates. Instead, start with an example, parameterize it and give it a unique syntax and name.

7.1. Benefits of a Pattern-Language

We think that several fields of software engineering would be positively affected by such a *pattern oriented design*, i.e. (symbolic) abstraction as a whole as the major design principle – thereby including other abstraction mechanisms, for example, functions:

Productivity / Expressibility / Intuitiveness In general, reducing code-redundancy by patterns for repeating tasks leads to a significant reduction of errors. As pattern-language increases the possibility for abstraction, programmers are enabled to directly express their ideas with the syntax they want to use; thus, much faster, direct and concise than without patterns. We think that this will compensate the extra work to learn a new syntax.

Understandability / Sustainability / Evolvability Most systems are designed for a long evolutionary existence. A program written in a notation suitable for its domain of usage is easier to read, maintain, adapt and enhance.

Structuring / Abstraction / Modularizability Patterns help in structuring the program or the problem domain; the declaration of a pattern is the manifestation of a new idea.

Learnability / Presentability Pattern-language-programmers would quickly learn the meaning of new features as they are defined on the basis of the already existing features. Code can be written and read in the familiar way.

For instance, in real mathematical notation instead of more or less convenient ASCII-mappings (as in the lower variant):

$$f(x) = x^2 + \sin x - \cos 2a_i$$

```
f(x) = x^2 + sin(x) - cos(2*a[i])
```

A pattern language could, as well, help in teaching programming languages theory to novice programmers as from the point of view of **π** programming languages can be regarded as a set of (changing) features.

Individuality / Freedom Programmers want to have freedom. This is a social argument rather than a technical one. However, programming *is* a social action, too. Programming is a form of expressing oneself in a creative act which is a great stimulus for general progress and personal satisfaction. Every programming language makes a trade-off between freedom and safety. **π** aims at providing as much of the former without completely sacrificing the latter.

Progress With a pattern-language language design becomes a community-process increasing the general progress in language design as syntax is then exposed to evolutionary mechanisms. From this evolution, not only a pattern-language could directly benefit but other programming languages, too, by incorporating new constructs evolving in and from the pattern-languages.

7.2. Our Plea

This work wants as well to be understood as advocating:

A Meta-Goal: a Renaissance of the Origins A pattern language might do a small contribution to what we would call the "renaissance of the origins": coming back to the roots and rethink some of the decisions made on the long way from the early days of programming to modern contemporary programming languages. During this long time, a lot of concessions had to be made concerning the trade off between possible programming language features and technical feasibility. These should be remembered as what they are: temporary trade-offs and by no means necessities arising from the nature of things. Some of these decisions might not any longer be completely right, for example, the decision for languages with a closed syntax.

In addition to that, we would like to shed light on some too familiar ideas of contemporary programming like classes, methods, aspects, control structures and others, thereby following the essence of science: *making familiar things unfamiliar*. A more fundamental research might as well reveal possibly forgotten ideas of the past still waiting for their time to come; sometimes one has to step back in order to jump forward.

A Meta-Goal: the Democratization of Language Development The extensions of languages are done in a half-transparent process – in the sense that user-participation is mainly restricted to making requests but final decisions are made by a small group of people in a standards committee. With a pattern language every programmer could take part in extending the language by the constructs she / he has in mind. The more restrictive a language / notation is, the less freedom a language allows, the more it reduces our creativity since right from the beginning our thoughts are forced into the corset of this specific notation and the less room is left for progress. Syntactic extensibility could lead to an open marketplace for syntax where languages syntactically advance on need and constructs are in competition with each other concerning their usefulness instead of being introduced – or not – in an authoritarian way. In any case, there will automatically develop common idioms, as programmers have a natural interest in their programs being readable by others; this is the same process of competing exclusiveness and universality as it is happening with the individual natural language(s) each of us uses.

A Plea for the Focus on Language Design There seems to be a trend to compensate deficiencies in language design with ever more elaborated programming tools. This "toolerism" has certain drawbacks: tools become out of date and incompatible with the language or with other tools. Programmers have to spend a lot of time to integrate the different tools and keep them up-to-date. One of the most prevalent arguments of those advocating tools is a circular argument: new languages would be disadvantageous as they would not any longer be compatible with the existing *tools*! We plea for a *language oriented design* instead of rather than subservience to tools, good tools will follow good languages automatically, we don't have to worry about that.

In addition to that, we plea for meta-complete languages instead of "meta-ization" creating cascades of ever more meta-levels or languages. We think that the existence of several meta-levels is not a sign of quality but rather the opposite as it shows that each level seems to suffer from a lack of expressivity. We think that the creation of new, pure

and consistently designed languages, making a tabula rasa should be preferred over overloading the languages currently en vogue until they finally go down with ever more features. This approach is not contradictory to a continuous evolution of languages: new languages – and new tools – derive from the experiences we have made with the former ones: "new" does not necessarily mean completely *different* but *better*.

A Plea for the Importance of Syntax In abstract programming language theory one could up to a certain limit (the readability for the analyst himself) neglect syntax and concentrate on semantics only. However, this has nothing to do with programming reality and leads to the widespread underestimation of the importance of syntax which is reflected in subtleties such as the emotional preference programmers have for a syntax she / he is familiar with: programmers do not think in calculi but in symbols and they want to use symbols for real programs, especially these symbols they are already used to – why else would programming languages syntactically refer so much to their predecessors, e.g. Java to C++/C? Why else would Microsoft provide their .NET-framework – one API, (semantically) one language – in different syntaxes? It is therefore the duty of software technology to provide programmers – especially those with customer contact – the tools they need to express themselves in the way *they* want to. Syntax *does* matter.

A Plea for more Risk and Dynamism **π** is a dynamic language. As the syntax is subject to dynamic modifications, it cannot be statically proven that a program is correct. Yet, dynamic programming languages have shown to be attractive for programmers. We are aware that **π** is a "dangerous tool". In practice, we think, a programmer would probably use a language (in form of a **π**-library) she/he is familiar with and add only several new patterns. An experienced language designer might use **π** in another, more advanced way. In our opinion, freedom in programming should not neglectfully sacrificed just for a higher safety. Natural language is the most powerful tool of communication we know and at the same time it is far from being open to formal proofs as ambiguities in natural language are resolved by further inquiries of the "programmed" person instead of using a (very restricted) unambiguous language in the first place. In the future we will therefore concentrate our research on how to give programmers (IDE-)support and intelligent feedback for pattern-programming. Based on their experience, programmers themselves will learn how to write programs which are adequate for them. So, with minimal personal insight and partial program analysis, it is possible to write type-safe **π**-programs. If we were afraid that our children died of a car accident, then we might consider to never let them drive a car, at all. Or we might think about giving them the best training and advice around to prepare them as good as we can for any eventuality. This is the concept that **π** follows. **π** does not want to put any restrictions on the programmer by force.

Our hope is that **π** will be used as an open artifact for studying the concept of patterns in programming and experimenting with many other new ideas and languages. In our opinion, there should be a common open source standard pattern language for the community, like, for instance, Haskell serves for functional programming. **π** could be a first source of inspiration for such a language. At least, **π** might serve as a basis for a lot of gedankenexperiments.

Acknowledgements

We are most indebted to Marc Wagner from Universität des Saarlandes who gave us the idea of using an Earley-parser for the realization of **π** as described in the work of John Aycock and R. Nigel Horspool [2]. **π** would not exist without the great help and technical knowledge of Felix Wolff who implemented the parser. The articles most inspiring to us concerning **π** were the works of Christopher Graham Seaton on "Katahdin" [23], of Ceteva, Inc. on "XMF" [22], of Jonathan Bachrach and Keith Playford on "The Java Syntactic Extender (JSE)" [3] and of Claus Brabrand and Michael I. Schwartzbach on "Growing Languages with Metamorphic Syntax Macros" [7]. Finally, we thank the members of our group, especially Andreas Sewe, Vaidas Gasunias, Tatjana Korbmacher and Tom Dinkelaker for their thorough feedback.

References

[1] Eric Allen et al. *Growing a Syntax* Sun Microsystems, FOOL, 2009

[2] John Aycock, R. Nigel Horspool *Practical Earley Parsing* University of Calgary, University of Victoria, Canada, The Computer Journal, Volume 45, Number 6, 2002

[3] Jonathan Bachrach, Massachussetts Institute of Technology *The Java Syntactic Extender (JSE)* Keith Playford, Functional Objects, Inc., OOPSLA, 2001

[4] Jonathan Bachrach *D-Expressions: Lisp Power, Dylan Style* Massachusetts Institute of Technology, USA, Keith Playford, Functional Objects Inc., Somerville, USA

[5] Don Batory, Bernie Lofaso, Yannis Smaragdakis *JTS: Tools for Implementing Domain-Specific Languages* The University of Texas at Austin, ICSR, 1998-06

[6] *<Bigwig>* http://www.brics.dk/bigwig, 2009-01-22

[7] Claus Brabrand, Michael I. Schwartzbach *Growing Languages with Metamorphic Syntax Macros*, PEPM, 2002

[8] Gilad Bracha *Executable Grammars in Newspeak* Cadence Design Systems, San Jose, California, USA, ENTCS, Volume 193, Pages 3-18, 2007-11

[9] Martin Bravenboer, Eelco Visser *Concrete Syntax for Objects* Universiteit Utrecht, The Netherlands, OOPSLA, 2004

[10] Boris Burshteyn *Generation and Recognition of Formal Languages by Modifiable Grammars* ACM SIGPLAN Notices, Volume 25, Number 12, Pages 45-53, 1990-12

[11] Boris Burshteyn *USSA–Universal Syntax and Semantics Analyzer* ACM SIGPLAN Notices, Volume 27, Number 1, Pages 42-60, 1992-01

[12] Alfonso Caracciolo di Forino *Some Remarks on the Syntax of Symbolic Programming Languages* Communication of the ACM, Volume 6, Number 8, Pages 456-460, 1963-08

[13] T. E. Cheatham *The introduction of definitional facilities into higher level programming languages* AFIPS, 1966-11

[14] Henning Christiansen *A Survey of Adaptable Grammars* Roskilde University Centre, SIGPLAN Notices, volume 25 number 11, pages 33-44, 1990-11

[15] Tony Clark *Beyond Annotations: A Proposal for Extensible Java (XJ)* Thames Valley University, United Kingdom, Paul Sammut, James Willans, Cetava Inc.

[16] Christopher Diggins *Superlanguages: Syntactic and Semantic Supersets of other Languages* 2008-03-12

[17] Torbjörn Ekman, Görel Hedin *The JastAdd Extensible Java Compiler* OOPSLA, 2007-10

[18] Bryan Ford *Parsing Expression Grammars: A Recognition Based Syntactic Foundation* POPL, 2004-01

[19] *The Fortress Language Specification* Sun Microsystems Inc., 2007

[20] *Isabelle* http://www.cl.cam.ac.uk/research/hvg/Isabelle 2009-03-16

[21] *JastAdd* http://jastadd.org, 2009-02-25

[22] Tony Clark, Paul Sammut, James Willans *Superlanguages – Developing Languages and Applications with XMF* Ceteva Inc., 2008

[23] Christopher Graham Seaton *A Programming Language Where the Syntax and Semantics Are Mutable at Runtime* Master's Thesis, University of Bristol, United Kingdom, 2007-05

[24] Bent Brrun Kristensen et al. *Abstraction mechanisms in the BETA programming language* Aalborg University Center, Aalborg, Denmark, POPL, 1983

[25] *Logix* http://www.livelogix.com/logix, 2009-03-02

[26] *Rats! – An Easily Extensible Parser Generator* http://www.cs.nyu.edu/rgrimm/xtc/rats.html, 2009-02-27

[27] Tim Sheard, Zino Benalssa, Matthleu Martel *Introduction to multistage Programming Using MetaML* Pacific Software Research Center, Oregon Graduate Institute of Science and Technology, 2000-02

[28] Walid Taha *A Gentle Introduction to Multi-stage Programming* Rice University, Houston, Texas, USA, DSPG, 2003

[29] Walid Taha *A Gentle Introduction to Multi-stage Programming, Part II* Rice University, Houston, USA, GTTSE, 2007

[30] Valentin F. Turchin et al. *Язык РЕФАЛ и его использование в задачах автоматизации программирования (English: The Language REFAL and its Application in the Automation of Programming)* Inter-University Conference on the Automation of Programming of Economical Calculations, Moscow, 1967

[31] Markus Völter *Architecture as Language: A story* InfoQ, 2009-01-28

[32] M. P. Ward *Language Oriented Programming* Computer Science Department, Durham, 2003-01

[33] Alessandro Warth *OMeta: an Object-Oriented Language for Pattern Matching* University of California, Los Angeles, USA, Ian Piumarta, Viewpoints Research Institute, Glendale, California, USA, Dynamic Languages Symposium, OOPSLA, 2007-10

[34] Daniel Weise, Roger Crew *Programmable Syntax Macros* Microsoft Research Laboratory, PLDI, 1993

[35] *XLR: Extensible Language and Runtime* http://xlr.sourceforge.net/concept/XL.html, 2009-02-19