Refal: the Language for Processing XML Documents*

Valentin F. Turchin
SuperCompilers LLC
e-mail: VTurchin@BellAtlantic.net

1 Wanted: a Metalanguage

The last few years witnessed the emergence of a new class of formal languages: markup languages, such as HTML and XML. A text in such a language is a document structured in a certain way according to the rules of the language. The common feature of markup languages is the use of tags, which allows to treat a string of characters as a tree-like structure. The non-algorithmic nature of markup languages calls for a language (a *metalanguage* with respect to markup languages) for writing programs which would parse, check and, generally, make whatever use of the marked-up document was intended by its author.

We need a high-level programming language, convenient for working with markup languages, but universal in the sense that it is also convenient for general symbol manipulation. Further narrowing of the class of admissible data and algorithms is undesirable, because the software dealing with texts in markup languages will, no doubt, grow in volume, diversity and sophistication.

In addition to being universal, the language must be conceptually simple and easy for an unsophisticated user—at least, as long as the user wants something relatively simple.

2 Refal

We believe that the language Refal [1] is the best choice for such a metalanguage. Refal (REcursive Functions Algorithmic Language) was conceived by the present author and implemented in the early 1970s in the Soviet Union, where it became widely known and used in the 1970s and 1980s for writing various translators and converters, as well as programs of Artificial Intelligence type. By the 1990s the existing implementations of Refal became obsolete, and no new implementations appeared, mainly because of the general crisis in the countries of the former Soviet Union. But Refal is a simple language, and the methods of its efficient implementation are well researched, so it would not be difficult to make a state of the art Refal System implemented in Java.

Refal is a purely functional language based on pattern matching. This means that:

- Computation is always an evaluation of some function call.
- A function is defined by a number of *sentences* (rules, or equations), where in the left side we see a pattern of the argument (i. e. an argument which may be only partially defined), whereas in the right side we see an expression which must be substituted for the function call in one step of computation. Presented in this way the program looks declarative: it is much easier to read

*This text has been initially published on the Web in HTML form. Reformatting by hand into LaTeX, as well as correcting some typos, and adding a couple of comments was done by Boyko Bantchev (boykobb@gmail.com)

and handle than a program presented as a sequence of commands (instructions) which include jumps of the execution point from one place to another.

3 Refal Compared to Other Languages

For some time after its inception, Refal was unique in combining these two features. In the 1980s a number of similar functional languages appeared, such as ML, Haskell, Miranda, etc. Comparing Refal with these languages we note, first of all, that it is the simplest language in the family. One of the design goals of Refal was to minimize the number of basic concepts which the user will have to understand and remember. In particular, variables in Refal can be only of three basic data types (see below), and you cannot define new types. To distinguish between different classes of values, tags can be used. Thus, if `x` is a variable whose value defines a dog, then we keep it as `(Dog x)` throughout the program. In some contexts this is even more convenient than having type declarations as a separate part of the program, because it keeps type names close to values and variables. Then type-related conditions and operations may get their visually clear definitions in terms of patterns.

The most important difference between Refal and all known to us functional languages is the fundamental data type used for manipulation of symbolic information. Refal uses *R-expressions* (Refal expressions); other languages use *lists*, or *S-expressions*.

Lists were first introduced in the language Lisp and became widely used in theoretical computer science. A list is a slightly restricted S-expression, while S-expression is a binary tree. Lisp's lists are not exactly lists as we understand it intuitively. When we speak of a list of three elements A, B, and C, we think of it as a chain $A - B - C$, which you can read from left to right or right to left. Not so with lists of computer scientists. Here a list is a skew binary tree where the list's elements make up the tree's rightmost path from the root. Symbolically, the list above must be seen as $A \rightarrow B \rightarrow C$. You cannot read it from right to left. To locate the last element you must start from the root A and pass the whole list. To sum up, such widely used data structure as string cannot be adequately represented in terms of Lisp's lists.

In Refal we allow a genuine concatenation of elements, which creates strings. This is achieved by representing the data in the computer with references in two directions: $A \leftrightarrow B \leftrightarrow C$. Therefore, the user can see his data as printed on paper or shown on the screen (i. e. ABC in our example), because whatever he can do on paper, he also can do in the computer.

Besides concatenation, Refal allows embracing an expression in parentheses. This creates tree-like structures with an arbitrary number of children of a node. For instance, `(A B (C) D)` may be understood as a tree where the root has four children: `A`, `B`, `(C)`, and `D`; `(C)` is a node with one child, `C`, and `A`, `B`, `C`, `D` have no children; they are called *leaves*.

The syntax of Refal expression is very simple. A *symbol* is a unit which is not decomposed into parts. It is a leaf when an expression is seen as a tree. A *term* is either a symbol or an expression in parentheses. An *expression* is a string of symbols and terms, possibly empty. Accordingly, there are three syntax types of variables: symbol variable (it has prefix `s.`), term variable (prefix `t.`), and expression variable (prefix `e.`).

It must be noted that parentheses in a Refal program are not characters, but special symbols that are represented in the computer in such a manner that the location of one parenthesis contains a reference to its pair, so that we can both enter parentheses, and jump over the parenthesized subexpressions. A parenthesis which is a character appears in quotes, as all other characters, so that it cannot be taken for a "genuine" Refal parenthesis.

4 Two small Programs in Refal

As an example of a simple Refal program let us define function `Palindrome` as a predicate telling us whether a string of letters is a palindrome (i. e. reads the same from left to right and right to left):

```
Palindrome  {
  s.1 e.x s.1 = <Palindrome e.x>;
  s.1         = True;
              = True;
  e.x         = False;
}
```

A call of function `F` with the argument `Arg` is written `<F Arg>`. The pattern of the argument in the first sentence, if put in words, is: the argument starts and ends with the same symbol. Note that this pattern is not translatable into LISP or any language based on lists. To define this function in LISP, the argument must be first reversed and then compared with the original list. This is clearly much less efficient than our program (which may give answer in one step).

The second and the third sentences tell us that a string consisting of one symbol or an empty string (it is allowed in REFAL) are palindromes. The sentences are tried in the order in which they are listed: this transforms a declaration into an algorithm. Hence the last sentence in the definition of `Palindrome` must be read: if none of the above sentences is applicable, then the answer is `False`. The last sentence here is always applicable, because `e.x` stands for an arbitrary argument.

Consider a simple example of a text editing problem. We want a function `Fab` which changes every `a` to `b`:

```
Fab  {
  'a' e.x   = 'b' <Fab e.x>;
  s.1 e.x   = s.1 <Fab e.x>;
  (e.y) e.x = (<Fab e.y>) <Fab e.x>;
            = ;
}
```

With this definition, the replacement of `a` by `b` takes place throughout the whole tree, i. e. on every level of the parenthesis structure. If we want to make it only on the top level, without entering parentheses, we rewrite the third sentence as:

```
(e.y) e.x = (e.y) <Fab e.x>
```

5 Mapping XML on R-expressions

The first markup language, HTML (HyperText Markup Language)[1] emerged in the context of the Internet, and it has been used in this capacity ever since. In a modified and generalized form it gave birth to XML (eXtensible Markup Language),[2] and it is believed to become soon, if not have already become, the universal standard for representing structured documents of any kind. So, we concentrate on XML.

To separate a substructure from the rest, XML uses two markers: <*tag* ...> on its left end and </*tag*> on the right side, where *tag* is a string of characters. The dots indicate that some optional information about the substructure may be put there. Substructures may be nested or concatenated like characters in a string.

The REFAL programmer immediately recognizes (and welcomes!) the familiar structure of a REFAL expression, where <*tag* ...> plays the role of a left parenthesis, and </*tag*> that of the right one. To use REFAL on XML, we first convert the input text into a REFAL expression. This is a simple one-pass procedure, an important part of which is transformation of tags into REFAL parentheses paired according to the well-known simple rules. This makes a tree from a string. Thus, the input

[1] HTML (1993) is not really the first markup language. Many different such languages have been in existence since at least the early 1960s. SGML (1986), of which HTML is an *application*, and GML, from which SGML itself descends, are but two examples. –B.B.

[2] Actually, XML is neither a modification or a generalization of HTML, and does not spring from it in any way. It is a *profile* (subset) of SGML, and is created for a very different purpose from that of HTML. –B.B.

transformation also plays the role of parsing and may give out error messages. The details of the input transformation may vary, depending on the preferences of the user. The general principle is: *organize semantically significant substructures by using parentheses.*

As an example, consider the following text in XML borrowed from [2]:

```
<Recipe>
<Name>Lime Jello Marshmallow Cottage Cheese Surprise</Name>
<Description>
My grandma's favorite (may she rest in peace).
</Description>
<Ingredients>
<Ingredient>
<Qty unit="box">1</Qty>
<Item>lime gelatin</Item>
</Ingredient>
<Ingredient>
<Qty unit="g">500</Qty>
<Item>multicolored tiny marshmallows</Item>
</Ingredient>
<Ingredient>
<Qty unit="ml">500</Qty>
<Item>Cottage cheese</Item>
</Ingredient>
<Ingredient>
<Qty unit="dash"/>
<Item optional="1">Tabasco sauce</Item>
</Ingredient>
</Ingredients>
<Instructions>
<Step>
Prepare lime gelatin according to package instructions
</Step>
And so on...
</Instructions>
</Recipe>
```

Listing 1. An XML document

After transformation into a Refal form it becomes:

```
((Recipe)
((Name) 'Lime Jello Marshmallow Cottage Cheese Surprise')
((Description) "My grandma's favorite (may she rest in peace).")
((Ingredients)
((Ingredient)
((Qty Unit Is "box") 1)
((Item) 'lime gelatin')
)
((Ingredient)
((Qty Unit Is "g") 500)
((Item) 'multicolored tiny marshmallows')
)
```

```
((Ingredient)
((Qty Unit Is "ml") 500)
((Item) 'Cottage cheese')
)
((Ingredient)
((Qty Unit Is "dash"))
((Item Optional Is "1") 'Tabasco sauce')
)
)
((Instructions)
((Step)
'Prepare lime gelatin according to package instructions...'
)
'<!-- and so on -->'
)
)
```

Listing 2. The document from Listing 1 formatted as an R-expression

We have used the following rules of transformation (presented rather informally and incompletely).

- An XML term of the form <*tag property-list*> *content* </*tag*> becomes the Refal term ((*tag* [*property-list*]) [*content*]), where square brackets indicate that the inside is subject to further transformation. Symbolic names, like *tag*, are distinguished in Refal by starting with a capital letter. Strings are delimited by single or double quotes.
- Property-list in XML is a list of properties, each of the form *property-name* = "*property-value*" where *property-name* is a symbolic name, and *property-value* is a string of characters. In Refal it becomes: *property-name* `Is ("`*property-value*`")`. We enclose *property-value* in parentheses for easier parsing. Property-lists may be empty.
- Strings in XML enter documents in their "natural form", without quotes. In R-expressions we enclose them in quotes. Blanks may be freely used for better readability.

One can see that the result of the input transformation, which will be dealt with by Refal programs, is very close to the original XML text, and is no less readable. The inverse transformation of a qualifying R-expression into an XML document is as simple and fast as the direct transformation.

As an example of processing XML in Refal consider this task: compile the list of the tags in a given document for which a certain property named `Property` has the value `True`. Let the name of the function which does the job be `Taglist`. Here is the program for it:

```
Taglist  {
  ((s.tag e.1 Property Is (True) e.2) e.x) = s.tag <Continue e.x>;
  ((s.tag e.properties) e.x)               = <Continue e.x>;
}

Continue  {
  e.x (e.term) e.y = <Taglist (e.term)> <Continue e.y>;
  e.x              = ;
}
```

The pattern in the left side of the first sentence is a combination of two patterns: the general pattern of an XML term: `((s.tag e.properties) e.x)` and the pattern `e.1 Property Is (True) e.2`, which is substituted for the variable `e.properties`. This pattern, in its turn, is of the form: `e.1` *something* `e.2`, which is satisfied when *something* is found in any place in a string, because the free variables `e.1` and `e.2` may be of arbitrary length. Putting all this together, we define the left side as

an XML term which in its property field has a substring: `Property Is (True)`. If this is the case, the right side instructs us to put out the name of the tag `s.tag`, and continue search over the inside part `e.x` of the term.

The second sentence of `Taglist` takes over when the first is not applicable, that is when `Property` is not `True`, or does not exist at all. In this case `s.tag` is ignored, and the search is continued. The function `Continue` looks for the first term in the argument, sends it to `Taglist` for processing, and recursively calls itself to continue.

6 Comparison with XSL

XSL[3] is a declarative language invented recently with the goal of transforming XML documents into something else, and first of all into HTML. Various versions of this transformation define various styles of presentation of an "abstract" XML document in a concrete form defined in HTML. We continue with the example above borrowed from [2]. In Listing 3 we have the HTML file from which the XML file in Listing 1 was abstracted. Therefore, there must be an XML-to-HTML transformation which concretizes Listing 1 as Listing 3. This transformation can be defined in XSL or in Refal (or in any other language). If it is in Refal, then Listing 1 must be first converted into its Refal form in Listing 2 (the output is in regular HTML). The XSL text is fully declarative, not a program, but an input for some interpreter. It is presented in Listing 4. The Refal text is a program ready for execution; see it in Listing 5.

```
<HTML>
<HEAD>
<TITLE>Lime Jello Marshmallow Cottage Cheese Surprise</TITLE>
</HEAD>
<BODY>
<H3>Lime Jello Marshmallow Cottage Cheese Surprise</H3>
My grandma's favorite (may she rest in peace).
<H4>Ingredients</H4>
<TABLE BORDER="1">
<TR
BGCOLOR="#308030"><TH>Qty</TH><TH>Units</TH><TH>Item</TH></TR>
<TR><TD>1</TD><TD>box</TD><TD>lime
gelatin</TD></TR>
<TR><TD>500 </TD><TD>g</TD><TD>multicolored
tiny marshmallows</TD></TR>
<TR><TD>500 </TD><TD>ml</TD><TD>Cottage
cheese</TD></TR>
<TR><TD></TD><TD>dash</TD><TD>Tabasco
sauce(optional)</TD></TR>
</Table>
<P>
<H4>Instructions</H4>
<OL>
<LI>Prepare lime gelatin according to package instructions...</LI>
<!-- and so on -->
</BODY>
</HTML>
```

Listing 3. The HTML file from XML-to-HTML transformation

[3]What is really at issue is XSLT: the transformation language within the family of languages called XSL. –B.B.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<xsl:template match="/Recipe">
<HTML>
<HEAD>
<TITLE>
<xsl:value-of select="Name"/>
</TITLE>
</HEAD>
<BODY>
<H3>
<xsl:value-of select="Name"/>
</H3>
<U>
<xsl:value-of select="Description"/>
</U>
<xsl:apply-templates/>
</BODY>
</HTML>
</xsl:template>
<!-- Format ingredients -->
<xsl:template match="Ingredients">
<H4>Ingredients</H4>
<TABLE BORDER="1">
<TR
BGCOLOR="#308030"><TH>Qty</TH><TH>Units</TH><TH>Item</TH></TR>
<xsl:for-each select="Ingredient">
<TR>
<!-- handle empty Qty elements separately -->
<xsl:if test='Qty[not(.="")]' >
<TD><xsl:value-of select="Qty"/></TD>
</xsl:if>
<xsl:if test='Qty[.=""]' >
<TD BGCOLOR="#404040"> </TD>
</xsl:if>
<TD><xsl:value-of select="Qty/@unit"/></TD>
<TD><xsl:value-of select="Item"/>
<xsl:if test='Item/@optional="1"'>
<SPAN> -- <em><u>optional</u></em></SPAN>
</xsl:if>
</TD>
</TR>
</xsl:for-each>
</TABLE>
</xsl:template>
<!-- Format instructions -->
<xsl:template match="Instructions">
<H4>Instructions</H4>
<OL>
<xsl:apply-templates select="Step"/>
```

```
</OL>
</xsl:template>
<xsl:template match="Step">
<LI><xsl:value-of select="."/></LI>
</xsl:template>
<!-- ignore all not matched -->
<xsl:template match="*" priority="-1"/>
</xsl:stylesheet>
```

Listing 4. The XSL file defining the XML-to-HTML transformation

```
* Convert Xml file to Html file
Xh {
((Recipe)((Name)e.name) ((Description) e.descr) e.on) =
'<HTML>'
'<HEAD>'
'<TITLE>' e.name '</TITLE>'
'</HEAD>'
'<BODY>'
'<H3>'e.name'</H3>'
e.descr
<Xh e.on>
'</BODY>'
'</HTML>' ;
((Ingredients) e.table) e.on =
'<H4>Ingredients</H4>'
'<TABLE BORDER="1">'
'<TR BGCOLOR="#308030"><TH>Qty</TH><TH>Units</TH><TH>Item</TH></TR>'
<Table e.table>
'</Table>'
<Xh e.on> ;
((Instructions) e.steps) e.on =
'<P>'
'<H4>Instructions</H4>'
'<OL>'
<Xh e.steps>
<Xh e.on>;
((Step) e.step) e.on =
'<LI>' e.step '</LI>'
e.on ;
= ; }
Table {
((Ingredient) ((Qty Unit Is e.unit) e.qty)
((Item e.option) e.item) ) e.on =
'<TR><TD>'e.qty'</TD>'
'<TD>'e.unit'</TD>'
'<TD>'e.item<Option e.option>'</TD>'
'</TR>'
<Table e.on>;
= ; }
Option {
```

```
Optional Is "1" = '(optional)';
= ; }
```

Listing 5. The Refal program `Xh` defining the XML-to-HTML transformation

The Refal program is shorter than XSL, and quite transparent for anybody who is familiar with basics of Refal. Let us review the structure of the program and the way it works. The function which does the job is `Xh` in Listing 5.

The left sides of Refal sentences are elements of the input XML file (Listing 2), while the right sides are the corresponding elements of the output HTML file (Listing 3). Since a program must be applicable to more that one fixed input, the left sides may include free variables, i. e. are *patterns*. These variables are used in the construction of the right sides. Information flow is from left to right.

Now look at Listing 2. This is the input XML file transformed into the Refal form. The tag `Recipe` is associated with the whole recipe, so the pattern in the first sentence of `Xh` in Listing 5 starts with `((Recipe) ....`

The next two terms are tagged as `Name` and `Description`. They show the strings associated with these tags:

```
((Name) 'Lime Jello Marshmallow Cottage Cheese Surprise')
((Description) "My grandma's favorite (may she rest in peace).")
```

In Listing 5 we generalize this as

```
((Name)e.name) ((Description) e.descr)
```

where `e.name` and `e.decr` are free varibles. When Listing 2 is on the input, they take the values:

```
e.name = 'Lime ... etc.'
e.descr = "My grandma's ... etc."
```

Now it is time to start the construction of the right side. We consult the desired output in Listing 3 for the standard beginning of the HTML format, and abstract from specific values of variables:

```
((Recipe)((Name)e.name) ((Description) e.descr) e.on) =
'<HTML>'
'<HEAD>'
'<TITLE>' e.name '</TITLE>'
'</HEAD>'
'<BODY>'
'<H3>'e.name'</H3>'
e.descr
```

Here `e.name` is used twice: as the title and as the beginning of the `BODY`.

Did you notice the variable `e.on` followed by a closing parenthesis in the left side? It covers the remaining part of the contents of `Recipe`, which is still in the input. Therefore, we recursively apply function `Xh` to it, in order to continue translation. The right parenthesis reminds us that the opening brackets `<BODY>` and `<HTML>` must be closed by `</BODY>` and `</HTML>`, respectively, after the computation of `Xh` on `e.on` is completed:

```
<Xh e.on>
'</BODY>'
'</HTML>' ;
```

The semicolon shows the end of the right side of the first sentence.

The next not yet used portion of the input in Listing 2 has the tag `Ingredients`. The structure of its contents is clearly seen due to the positioning of paired parentheses one under the other. It consists of four terms intended to be displayed by HTML in the form of a table with four lines. They are marked up by the tag `Ingredient` and all have the same structure, differing only in parameters. Therefore, we need not define the construction of the right side for each of the four lines; we can define

a generalized "abstract" line with free variables, and use it with various parameters, i. e. values of the variables, which the Refal machine will take from the input.

The left side of the second sentence has the already familiar structure:

```
((Ingredients) e.table) e.on =
```

Two variables will be defined when the input will be matched to this pattern: `e.table`, which is the four lines of the table, and `e.on`—the remaining part of the input to which the function `Xh` will be applied after the completion of the table.

But before including in the right side the four lines we are required to include the printout `Ingredients` and the HTML element `TABLE` which contains the record of the names of the columns in the table:

```
'<H4>Ingredients</H4>'
'<TABLE BORDER="1">'
'<TR BGCOLOR="#308030"><TH>Qty</TH><TH>Units</TH><TH>Item</TH></TR>'
<blockquote>
  <blockquote>
    Now we take out the creation of the table as a job
    for a new function
    Table, and do not forget to close the TABLE element:
    <Table e.table>
  </blockquote>
</blockquote>
'</Table>'
```

The definition of `Table` proceeds along the same lines as with `Xh`. Inspect the first line in the input (Listing 2):

```
((Ingredient)
((Qty Unit Is "box") 1)
((Item) 'lime gelatin')
)
```

Generalize it by replacing parameters `"box"`, `1`, and `'lime gelatin'` by `e.unit`, `e.qty` and `e.item`, respectively, and do not forget to put `e.on` for further lines—you have the left side of the first sentence of function `Table`.

```
((Ingredient) ((Qty Unit Is e.unit) e.qty)
((Item e.option) e.item) ) e.on =
```

There is a small detail though. In the properties field of the tag `Item` there is an option of inserting the string: `'Optional Is "1"'`. This indicates that after the name of the Item in the HTML output we must add: `'(optional)'`. To provide for this we put `e.option` after the tag `Item` in the left side above, and put the function call `<Option e.option>` in the corresponding place of the right side. Function `Option` is simple: if the string `'Optional Is "1"'` is in the input, then `'(optional)'` will be added right after the `Item` (see the first sentence of function `Option`; if `e.option` is empty, the addition is empty. This feature is actually used in our input, namely in the third line of the table.

The right side from `<TR>` to `</TR>` is a table record (line) in the HTML format. Function `Table` continues working on the remaining lines in `e.on`. If `e.on` is empty, the construction of the table is completed, which is ensured by the second sentence.

The remaining part of the definition of function `Xh` describes the beginning of a list of recipe instructions. We leave it to the reader to make sense of it.

7 Supercompilation

One more reason to use Refal is the availability of a *supercompiler* for Refal programs.

A supercompiler is a computer program for transformation of computer programs with the objective of making them work faster. Supercompilation is a global transformation; it can radically change the overall structure of the original program by tracing the possible generalized histories of computation in the original program and compiling an equivalent program, reducing in the process the redundancy that could be present in the original program. The most common type of redundancy is when the program treats some data as variable, but is actually called with a fixed value of this variable. A subtler case is when two or more variables are known to have the same value (which is, however, unknown). Finally, every composition of functions creates some redundancy because the outer function is defined for any argument, while actually working with the value of inner function only. There are interesting examples of successful supercompilation in all these cases. A special (easier) case of supercompilation is known as *partial evaluation*.

The concept of a supercompiler is a product of cybernetic thinking. A program is seen as a machine. To make sense of it, one must observe its operation. Thus instead of transforming the program step-by-step, as other optimization techniques do, the supercompiler runs the original program in a general form, with unknown values of variables and creates a graph of states and transitions between possible configurations of the computing system. This process (called *driving*) can usually go on indefinitely. To make it finite, the supercompiler performs the process of generalization on the system configurations in such a manner that it finally comes to a set of generalized configurations, called *basic*, in terms of which the behavior of the system can be expressed. The set of basic configurations with transitions between them becomes the flow chart of the transformed program. The original program is thrown away unchanged.

The most obvious application of supercompilation is in the use of interpreters of any kind. A function $L(v, t)$, where v is a list of variables, and t is a text in some language (call it also L), is an interpreter of L if operations on the variables in v are defined by consulting the text t. When we define function $L(v, t)$, we do that for an arbitrary t, but when we use it we have some specific T for t. Interpretation, as we know, is costly in terms of required time. A supercompiler transforms $L(v, T)$ into an efficient, specialized for $t = T$, function $LT(v)$. In our experience with supercompilation of interpreters, the typical speedup factor is around 40. The use of interpreting programs is limited to such situations where the loss of factor 40 in speed is acceptable. The availability of a supercompiler removes these limits and allows the programmer to use interpretation freely.

In the context of XML we can, in addition to direct use of Refal as above, define several variations of style languages specialized for different types of users. Because of specialization, using such a language will be much easier than using a single general language, such as XSL. (For example, a language might be reduced to a simple table form). We define these languages through interpretation in Refal, and use the supercompiler to obtain efficient programs.

8 Conclusion

We argue that Refal is the best language for processing documents in XML, HTML, and similar formats, because it has the following unique combination of features:

- Refal is a functional language. A program in Refal defines an algorithm, but looks as (and is, indeed) a declaration of certain relations between function calls.
- The choice of a relation to use is governed by pattern matching, which is very convenient visually, and allows to distinguish easily between different cases of arguments.
- Unlike other known to us functional languages, Refal has R-expressions as its basic data structure. It can represent both tree-like structures, and strings of symbols or terms which can be processed from left to right and right to left. The basic data structure of XML is a special case of R-expression. This choice of the creators of XML is not accidental. R-expressions are not only convenient; they are familiar to everybody as expressions of school algebra, which also are R-expressions.

- REFAL has a short list of basic concepts and a simple syntax. Hence it is fast to learn.
- REFAL is universal as a language for processing symbolic information. It can stay as the only needed programming language at all stages of the work with XML documents: from simplest one-to-one substitutions to sophisticated Artificial Intelligence systems.
- A supercompiler for REFAL is available.

As of now, REFAL is implemented in C. It is highly desirable to make it available in the context of JAVA.

Acknowledgments

It was Andrei V. Klimov who turned my attention to XML and noticed the close similarity between its data structure and that of REFAL. This paper was discussed with him, as well as with Arkady V. Klimov. Thanks to both.

References

[1] Valentin F. Turchin. *Refal-5: Programming Guide & Reference Manual*, New England Publishing Co., Holyoke, 1989.

[2] Mark Johnson. *XML for the Absolute Beginner*, `http://www.javaworld.com/javaworld/jw-04-1999/jw-04-xml_p.htm`.