Seminar Report

XLR

Benedikt Tuschter
benedikt.tuschter@student.uibk.ac.at

31 July 2015

Supervisors:
Manuel Schneckenreither, Assoc. Prof. Dr. Georg Moser

Abstract

This document is a short overview of XLR - Extensible Language and runtime. It is a dynamic language based on meta-programming, i.e. programs that manipulate other programs. The Syntax is similar to other imperative languages (e.g. C or Pascal) and has the power and expressiveness of homoiconic languages descending from Lisp (i.e languages where programs are data). Concept programming is the underlying design philosophy behind XLR. The core idea is very simple: programming is the art of transforming ideas (i.e. concepts that belong to concept space) into artefacts such as programs or data structure (i.e. code that belongs to code space). The document shows the usage of XLR and commonalities to other languages. This is not a full documentation, but it helps to get started with this programming language.

Contents

1 Introduction **1**

2 Syntax **1**
2.1 Spaces, Indentation and Comments 2
2.2 Literals 2
2.2.1 Integer constants 2
2.2.2 Real constants 3
2.2.3 Text literals, names and operator symbols 3
2.3 Structured nodes 4
2.4 Parsing rules 4
2.4.1 Precedence 4
2.4.2 Associativity 5
2.4.3 Infix versus Prefix versus Postfix 5
2.4.4 Expression versus statement 5

3 Language semantics **6**
3.1 Rewrite declarations 6
3.2 Data declaration 7
3.3 Type declaration 7
3.4 Assignment, Guards, Sequences, Index operators 7

4 XL Library **8**
4.1 Built-in operations 9
4.2 Control structures 9

5 Conclusio **10**

Bibliography **11**

1 Introduction

XL stands for 'Extensible Language'. It's a programming language designed to address the challenges faced by programmers today. It addresses these challenges not by adding language features, but by making it easy and safe for any programmer to add features themselves. Today, programmers have to deal with exponentially-growing program complexity, because the complexity indirectly follows Moore's law. But humans do not follow a law in common of Moore's law, so tools to bridge the gab with higher and higher levels of abstractions are needed. XLR was designed to allow programmers control over their naturally growing language.
The XLR syntax and semantics are very simple and easy to write and read. Programs in XLR look a bit like pseudo code for programmers, except that they can be compiled an run. Therefore it only needs 8 data types which form an *abstract syntax tree* (AST). Because of the importance of meta-programming, the data structure representing programs is very simple and practical. Extensibility is a key function of XLR, and so it is possible and just part of everyday programming to create *domain-specific languages* in a simple way. To understand how XLR programs execute, it also needs understanding about ASTs. Conceptually, an XLR program is a transformation of ASTs following a number of tree rewrite rules.
XL is defined at four different levels. The first level is *XL0*. It is the basic representation of XL programs and defines how an input text is transformed into a parse tree. The second level *XL1* defines a base language with features comparable with C++. It is a statically compiled imperative language based on XL0 with novel generic capabilities. XL2 is the third level and a complete language including XL1 with a library and a runtime. It defines the standard library, which includes common data types and operators. And the last level *XLR* defines a dynamic runtime for XL based on XL0. [1]
XL is available as a free software project under the GNU General Public License and the main project site is *http://xlr.sf.net*.

2 Syntax

XLR source code is parsed into an abstract syntax tree format known as XL0. These trees consist of four literal (integer, real, text and symbol) and four structured (prefix, postfix, infix and block) node types.The precedence of operators is either given by the *xl.syntax* configuration file or in the source code by using the syntax statements. These both methods are called *syntax configuration*.
XLR is very easy to read and write as the following listing shows:

```
0! -> 1
N! -> N * (N-1)!
```

Listing: Factorial in XLR

The easy syntax will help you to write shorter and more beautiful code.

2.1 Spaces, Indentation and Comments

Spaces and tabs are not significant except to separate operator or name symbols. For example, there is no difference if between **A** and **B** are two or three spaces, but both are different to from **AB** (zero space). XLR will use spaces and tabs at the beginning of lines to determine the level of indentation from which it derives program structures (off-side rule). They both can be used for indentation, but cannot be mixed in a single source file. In other words, if the first indented line uses spaces, all other indentation must be done using spaces, and similarly for tabs.

```
if A < 10 then
    write "A is smaller 10"
else
    write "A is not smaller 10"
```

Listing: Off-side rule

As used in other Languages, *Comments* are used for documentation purpose and play no role in the execution of the program. Comments begin with a comment separator and end with a comment terminator. Comments are similar to C++ comments, they begin with /* and finish with */ or they begin with // and finish with the end of the line, as shown in the next listing:

```
/* multi line
comment */

// single-line comment
```

Listing: Comments

Comments are recorded as attachments in XL0 and can be used by a documentation generator to construct documentation automatically.

2.2 Literals

Four literal node types represent atomic values, i.e. which cannot be decomposed into smaller units from an XLR point of view. These literal node types are *Integer* constants, *Real* constants, *Text* literals, *Symbols* and *Names*.

2.2.1 Integer constants

Integer constants consist of one or more digits from 0 to 9 interpreted as radix-10 values. The constant is defined by the longest possible sequence of digits. Note, -9 is not an integer literal, but an prefix preceding the literal. These integer literals can be expressed in any radix between 2 and 36. They begin with a radix-10 integer, followed by a hash sign an valid digits in the given radix. For example, 2#1001 represents the same

integer as 9. There is also the underscore character, which separate digits for a better representation, but does not change the value. The following listing shows some usages of integer constants:

```
13
3_000_000
16#FFFF_FFFF
2#1010_1010_1010_1010
```

Listing: Valid integer constants

2.2.2 Real constants

Real constants look very similar to most of the other languages, they consist of one or more digits between 0 and 9, followed by a dot and again followed by one or more digits between 0 and 9. Note, there must be at least one digit after the dot, otherwise it's not a valid real constant. Real constants can have a radix and underscores like integer constants. It can also have an exponent, which consists of an optional hash sign #, followed by the character e or E, followed by optional plus + or minus - sign, followed by one or more decimal digits from 0 to 9. For example 1.0e-3 is the same as 0.001. The exponent is always given in radix-10 and indicates a power of the given radix. The hash sign # is only required for any radix greater than 14, since in that case the character e or E is also a valid digit. For instance, 16#1.0E1 is approximately the same as 1.05493, whereas 16#1.0#E1 is the same as 16.0.

```
2.0
3.1451_9256_3598_7923
2#1.0000_0001#e-127
```

Listing: Valid real constants

2.2.3 Text literals, names and operator symbols

Text is any valid UTF-8 sequence of printable or space characters surrounded by single quotes ' or double quotes ". They can be used to enclose any text that doesnâĂŹt contain a line-terminating character. The same delimiter must be used at the beginning and at the end of the text. For example, "Hello World" is a valid text surrounded by double quotes, and She said "Hey"âĂŹ is a valid text surrounded by single quotes. Other text delimiters can be specified, which can be used to delimit text that may include line breaks. Such text is called long text . With the default configuration, long text can be delimited with « and ».

```
"Hello XLR"
'Hello XLR again'
<< text that spans
multiple lines>>
```

Listing: Valid text constants

Names begin with an alphanumeric character from A to Z or a to z, followed by the longest possible sequence of alphanumeric characters, digits or underscores. Note, two consecutive underscore characters are not allowed. Operator *symbol*, or *operators* begin with an ASCII punctuation character which does not act as a special delimiter for text, comments or blocks.

```
y
Y13_before_transformation
+
-->
<<<>>>
```

Listing: Examples of valid operator and name symbols

2.3 Structured nodes

There exist four structured node types which represent combinations of nodes. These nodes are called *Infix*, *Prefix*, *Postfix* and *Blocks*. We call a node Infix, if the operator is between its operands (i.e. A+B). If the operator is before or after its operand, than we call it a Prefix (i.e. +4) or a Postfix (i.e. cos x). A Block means, that the operators surround their operand (i.e. (A+B)). An infix node has one children on its left, and one on its right, separated by a name or operator symbol. Prefix and postfix nodes have also two children, but without any separator between them. The only difference is in what is considered the âĂIJoperationâĂİ and what is considered the âĂIJoperandâĂŹâĂŹ. For a prefix node, the operation is on the left and the operand on the right, whereas for a postfix node, the operation is on the right and the operand on the left. Typically prefix nodes are used for functions. For example, *sin x* is defined as a function in the *xl.syntax* file. *Bock* nodes have one child bracketed by two delimiters. Following pairs are recognized as block delimiters: Parenthesis (A), Brackets [] and Curly braces A.

2.4 Parsing rules

There are only four rules to parse any XLR source code into XL0. These rules are called *precedence*, *associativity*, *Infix versus prefix versus postfix* and *expression versus statement* and are detailed below.

2.4.1 Precedence

Infix, prefix, postfix and block symbols are ranked according to their precedence, represented as a non-negative integer. The precedence is specified either in the *xl.syntax* configuration file or through *syntax* statements in the source code. That means symbols with higher precedence associate before symbols with lower precedence. If we assume that the symbol * has infix precedence value 200 and symbol + has infix precedence value

190, then the expression 3+4*6 will parse as an infix + whose right child is an infix *. Note, the same symbol may receive different precedence values for prefix, infix or postfix operators. For example, if the precedence of - as an infix is 190 and the precedence of - as a prefix is 290, then the expression 4 - -6 will parse as an infix - with a prefix - as a right child. The precedence of blocks is used to define the precedence of the resulting expression.

2.4.2 Associativity

Infix operators can associate to their left or to their right. For Example, the addition operator is traditionally left-associative, that means that A and B associate before C, of we have as an expression A+B+C. In other words, the outer infix + node in A+B+C has an infix + node as its left child with A and B as children, and C as its right child. An example for a right-associative symbol is the semi-colon, meaning that A;B;C is an infix node with an infix as the right child and A as the left child. To ensure ambiguity, left-associative operators have even precedence values and right-associative operators have odd precedence values.

2.4.3 Infix versus Prefix versus Postfix

During parsing, XLR needs to know if an operator is treated as an infix, prefix or postfix. Therefor XLR has some rules to resolve this ambiguity [2]:

- The first symbol in a statement or in a block is a prefix: and in (and x) is a prefix.
- A symbol on the right of an infix symbol is a prefix: and in A+and B is a prefix
- Otherwise, if the symbol has an infix precedence but no prefix precedence, then it is interpreted as an infix: and in A and B is an infix.
- Otherwise, if the symbol has a postfix precedence, then it is a postfix: % in 3% is a postfix.
- Otherwise, the symbol is a prefix: sin in write sin x is a prefix.

Symbols without being given an explicit precedence are called *default prefix*. They receive a particular precedence known as *function precedence*, identified by FUNCTION in the syntax configuration.

2.4.4 Expression versus statement

Also a big problem is how humans read text. For Example, humans would think that *write cos x, cos y* is a write instruction with two arguments, but that is not logical. If it would parse like that, then why shouldn't *cos* also take two arguments and produce *write(cos(x, cos(x)))* as the parsing result ? XLR resolves this ambiguity by implementing a distinction which also exists in many natural languages. The boundary is a particular infix precedence, called statement precedence, denoted as *STATEMENT* in the syntax

configuration. In detail, that means that if a blocks precedence is less than statement precedence, the following content begins with an expression, otherwise it begins with a statement. For Example, 4 in (4) is an expression, write in write is a statement. Right after an infix symbol with a precedence lower than statement precedence, we are in a statement, otherwise we are in an expression. That means A in B+A is an expression, whereas A in B;A is a statement. A complementary rule applies after prefix nodes. optimize write A,B gives two arguments to write, whereas in (x-¿x+1) sin x,y the sin function only receives a single argument. Usually if a prefix is a name, it begins a statement otherwise an expression, if it is a symbol. the name write in write X begins a statement, the symbol + in +3 begins an expression.

3 Language semantics

The semantics of XLR is based entirely on the rewrite of XL0 abstract syntax trees. Tree rewrite operations define the execution of XLR programs, also called *evaluation*. There is only a small set of tree rewrite operators with special meaning for the XLR compiler. There exists *rewrite declaration*, which are used to declare operations, *data declarations*, which identify data structures in the program and *type declarations*, which define the type of variables. Moreover we have *guards* and *assignments*. Guards limit the validity of rewrite or data declarations and assignments change the value associated to a binding. Furthermore *sequence operators* indicate the order in which computations must be performed and *index operators*, which perform particular kinds of tree rewrites similar in usage to 'structure' or 'arrays' in other programming languages.

3.1 Rewrite declarations

The infix → operator declares a tree rewrite. The following listing shows the **if-then-else** statement defined by the rewrite operator.

```
if true then TrueClause else FalseClause -> TrueClause
if false then TrueClause else FalseClause -> FalseClause
```

Listing: If-then-else rewrite declaration

The left tree of the → operator is called the *pattern* and the tree on the right is called the *implementation*. This declaration means that the pattern should be rewritten using the implementation. The pattern contains constant and variable symbols and names [2]:

- Infix symbols and names are constant, like + in A+B.
- Block-delimiting symbols and names are constant, like [and] in [A].
- A name on the left of a prefix is a constant, like sin in sin X.
- A name on the right of a postfix is a constant, like cm in X cm.

- A name alone on the left of a rewrite is a constant, like X in X→0.
- Operators are constant, like + in X and +Y.
- All other names are variable.

To form the structure of a pattern, we use constants and names, whereas variables are used to form the parts of a pattern which can match other trees. The names are called parameters and the tree they match are called arguments. But there is also a special case for rewrite declarations, for example **X→0** binds X to a value 0. If we want to create a function which increments its input, we can rewrite **X:real→X+1**. This does not declare the variable X, but an anonymous function. Rewrite declarations roughly play the role of functions, macro or operator declaration in other programming languages.

3.2 Data declaration

Rewrites, that do not need to be rewritten any further start with the prefix *data*. For Example, if we declare *data complex(a, b)*, than complex(4+5, 6+7) evaluates to complex(9, 13) but no further. This declaration can be interpreted as declaring a complex data type. If there is any tree matching such a pattern, this tree will not be rewritten further. Data declarations only limit the rewrite of the tree specified by the pattern, but not the evaluation of pattern variables.

3.3 Type declaration

An *type declaration* is an infix colon : operator in a rewrite or data pattern with a name on the left and a type on the right. A *return type declaration* is an infix as in a rewrite pattern with a pattern on the left and a type on the right. With a return type declaration a return type can be specified. The following listing shows examples of type declarations:

```
polynom X:real
Z:real
N:integer as real -> (X–Z)^N
```

Listing: Type declaration

For Example to match the pattern of polynom, the arguments X and Z must be real and the value returned by polynom will also belong to real. So type declaration enables *overloading*, that means to have multiple functions or operators with similar structure but different types of parameters, like in other languages, e.g. Java or C.

3.4 Assignment, Guards, Sequences, Index operators

The assignment operator := binds the reference on its left to the value of the tree on its right. Local and global variables can be assigned. A new binding in the local scope is created if the left side of an assignment is a type declaration and has a return type declaration associated with it.

```
// Assigns to global Y
assigns_to_global -> Y := 1
// Global Y
Y := 0
// Assign to local Y
assigns_new -> Y:integer := 1
```

Listing: Assignments

A *guard* is similar to a boolean condition in other languages, that means that the pattern applies only if the guard is true. The infix *when* operator introduces a guard. The following listing shows how a guard limit the validity of operations.

```
0!                          -> 1
N! when N > 0 -> N * (N-1)!
```

Listing: Guard

The infix line-break NEWLINE and semi-colon ; operators are used to introduce a sequence between statements. Declarations and statements are items from sequences and include rewrite declarations.

Index operators perform like arrays or structures such as in other languages. The notation A[B] and A.B are used as index operators to refer to individual items in a collection. This data structures can be extended on the fly and also can include other kinds of rewrite, for example functions, thus object-oriented data-structures are enabled.

```
MyData ->
  Name -> "dataname"
  Value -> 2.36
  1 -> "first"
  2 -> "second"
  3 -> "third"
```

Listing: Guard

4 XL Library

The XLR language is very simply, with a strong focus on how to extend it rather than on built-in features. The XL Library offers a number of operation and makes it easy to extend the language. The following sections evidence some basic operations available in any XLR program. As described before, operator priorities are defined by the *xl.syntax* file, whereas regular rewrite rules for arithmetic or comparison where defined in the *builtins.xl* file, which is loaded by XLR before evaluating any program.

4.1 Built-in operations

The XL Library contains basic operations like arithmetic, comparison, text functions conversions and many more.
Arithmetic Arithmetic operators are mostly used for integer and real values. The operators like addition or division take two arguments and return an argument of the same type. *Comparison* operators return true or false and can take integer, real and text as argument. There also exist *bitwise* and *boolean* operators which we know from other languages like C or Fortran. The Library also includes *mathematical* functions like known from other languages, *text functions*, *conversions* and *tree operations*. By convention, XLR use comma-separated lists, such as 1,3,4,5, and therefore offers functions, for example known from the language OCaml, like *map*, *reduce*, *head*, *tail* any some more. These are some examples:

```
x+y                 Addition
x mod y             Modulo
x!                  Factorial
x<=y                Less or euqal
not x               Logical not
sqrt x              Square root
real x:integer      Convert integer to real
```

Listing: Built-in operations

These operators are really easy to read and write and for programmers well known code snippets. This example shows how similar XL language semantic is with better known languages like Java or C++.

4.2 Control structures

The XLR Library offers *infinite*, *conditional* and *controlled* loops. The following example shows a *while* loop, which runs while a given condition is true.

```
while Condition loop Body ->
   if Condition then
      Body
      while Condition loop Body
```

Listing: While loop

A very complex type of loop is the *for* loop. It exists in multiple variants. The simplest for-loop is shown in the listing below.

```
for Variable in Low:integer..High:integer loop Body ->
    Index : integer := Low
    while Index < High loop
        Variable := Index
        Body
        Index := Index + 1
```

It is also possible to create custom loops to explore other data structures.

5 Conclusio

With XL you have an experimental tool for exploring new ideas in programming. It gives you procedural approach with generics reminiscent of C++ and a functional approach with dynamic compilation reminiscent of Haskell.
With the flexibility of high order functions and dynamic compilation, XLR is a very powerful tool. It helps you to program effective code in a very short time. Especially C++ and Haskell developers feel very familiar with it. It helps you to write shorter and more beautiful code.
XLR is designed to grow with you, that means that your ideas have no limits. It's extensible feature is easy, save and fun.

References

[1] C. de Dinechin. Eliminating newspeak in programming languages. `http://xlr.sourceforge.net/newspeak`, 2010. Zugegriffen am : 25.07.2015.

[2] C. de Dinechin. The xlr programming language. `http://xlr.sourceforge.net/xlr-ref`, 2010. Zugegriffen am : 27.07.2015.