The XLR Programming Language

by Christophe de Dinechin

Taodyne SAS, 2229 Route des Cretes, Sophia Antipolis, France

1 Introduction

XLR is a dynamic language based on meta-programming, i.e. programs that manipulate programs. It is designed to enable a natural syntax similar to imperative languages descending from Algol (e.g. C, Pascal, Ada, Modula), while preserving the power and expressiveness of homoiconic languages descending from Lisp (i.e. languages where programs are data).

Benefits Thanks to this unique approach, XLR makes it easy to write simple programs, yet there is no upper limit, since you can extend the language to suit your own needs.

In order to keep programs easy to write, the XLR syntax is very simple. It will seem very natural to most programmers, except for what it's missing : XLR makes little use of parentheses and other punctuation characters. Instead, the syntax is based on indentation, and there was a conscious design decision to only keep characters that had an active role in the meaning of the program, as opposed to a purely syntactic role. As a result, XLR programs look a little like pseudo-code, except of course that they can be compiled and run.

This simplicity also makes meta-programming not just possible, but easy. All XLR programs can be represented with just 8 data types. Meta-programming is the key to extensibility. As your needs grow, you won't find yourself limited by XLR as you would with many other programming languages. Instead of wishing you had this or that feature in the language, you can simply add it. Better yet, the process of extending the language is so simple that you can now consider language notations or compilation techniques that are useful only in a particular context. In short, with XLR, creating your own *domain-specific languages* (DSLs) is just part of normal, everyday programming.

Examples The key characteristics of XLR are best illustrated with a few short examples, going from simple programming to more advanced functional-style programming to simple meta-programming.

Figure 1 illustrates the definition of the factorial function:

```
// Declaration of the factorial notation
0! -> 1
N! -> N * (N-1)!
```

Figure 1. Declaration of the factorial function

Figure 2 illustrates functions usually known as *map*, *reduce* and *filter*, which take other functions as arguments. In XLR, *map*, *reduce* and *filter* operations all use an infix `with` notatoin with slightly different forms for the parameters :

```
// Map: Computing the factorial of the first 10 integers
// The result is 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880
(N->N!) with 0..9

// Reduce: Compute the sum of the first 5 factorials, i.e. 409114
(X,Y -> X+Y) with (N->N!) with 0..5

// Filter: Displaying the factorials that are multiples of 3
// The result is 6, 24, 120, 720, 5040, 40320, 362880
(N when N mod 3 = 0) with (N->N!) with 0..9
```

Figure 2. Map, reduce and filter

Figure 3 illustrates the XLR definition of the `if-then-else` statement, which will serve as our first introduction to meta-programming. Here, we tell the compiler how to transform a particular form of the source code (the if-then-else statement). Note how this transformation uses the same `->` notation we used to declare a factorial function in Figure 1. This shows how, in XLR, meta-programming integrates transparently with regular programming.

```
// Declaration of if-then-else
if true then TrueClause else FalseClause   -> TrueClause
if false then TrueClause else FalseClause  -> FalseClause
```

Figure 3. Declaration of if-then-else

The next sections will clarify how these operations work.

2 Syntax

XLR source text is encoded using UTF-8. Source code is parsed into a an abstract tree known as *XL0*. XL0 trees consist of four literal node types (integer, real, text and symbol) and four structured node types (prefix, postfix, infix and block). Note that line breaks most often parse as infix operators, and that indentation most often parses as blocks.

The precedence of operators is given by the `xl.syntax` configuration file, and can also be changed dynamically in the source code. This is detailed in Section 2.6.

2.1 Spaces and indentation

Spaces and tabs are not significant, but may be required to separate operator or name symbols. For example, there is no difference between `A B` (one space) and `A  B` (four spaces), but it is different from `AB` (zero space).

Spaces and tabs are significant at the beginning of lines. XLR will use them to determine the level of indentation from which it derives program structures (off-side rule), as illustrated in Figure 4. Only space or tabs can be used for indentation, but not both in a same source file.

```
if A < 3 then
    write "A is too small"
else
    write "A is too big"
```

Figure 4. Off-side rule: Using indentation to mark program structure.

2.2 Comments and spaces

Comments are section of the source text which is not parsed. The ignored source text begins with a comment separator, and finishes with a comment terminator.

With the default configuration[1], comments are similar to C++ comments: they begin with `/*` and finish with `*/` or they begin with `//` and finish at the end of line. This is illustrated in Figure 5.

```
// This is a single-line comment
/* This particular comment
   can be placed on multiple lines */
```

Figure 5. Single-line and multi-line comments

1. In other words, with the `xl.syntax` configuration file as shipped with XLR.

2.3 Literals

Four literal node types represent atomic values, i.e. values which cannot be decomposed into smaller units from an XLR point of view. They are:

1. Integer constants
2. Real constants
3. Text literals
4. Symbols and names

Integer constants Integer constants[2] such as `123` consist of one or more digits (`0123456789`) interpreted as unsigned radix-10 values. Note that `-3` is not an integer literal but a prefix `-` preceding the integer literal. The constant is defined by the longest possible sequence of digits.

Integer constants can be expressed in any radix between 2 and 36. Such constants begin with a radix-10 integer specifying the radix, followed by a hash `#` sign, followed by valid digits in the given radix. For instance, `2#1001` represents the same integer constant as `9`. If the radix is larger than 10, letters are used to represent digits following `9`. For example, `16#FF` is the hexadecimal equivalent of `255`.

The underscore character `_` can be used to separate digits, but do not change the value being represented. For example `1_000_000` is a more legible way to write `1000000`, and `16#FFFF_FFFF` is the same as `16#FFFFFFFF`. Underscore characters can only separate digits, i.e. `1__3`, `_3` or `3_` are not valid integer constants.

```
12
1_000_000
16#FFFF_FFFF
2#1001_1001_1001_1001
```

Figure 6. Valid integer constants

Real constants Real constants such as `3.14` consist of one or more digits (`0123456789`), followed by a single dot `.` followed by one or more digits (`0123456789`). Note that there must be at least one digit after the dot, i.e. `1.` is not a valid real constant.

Real constants can have a radix and use underscores to separate digits like integer constants. For example `2#1.1` is the same as `1.5` and `3.141_592_653` is an approximation of π.

Optionally, a real constant can 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 `0123456789`. For example, `1.0e-3` is the same as `0.001` and `1.0E3` is the same as `1000.0`.

The exponent value is always given in radix-10, and indicates a power of the given radix. For example, `2#1.0e3` represents 2^3, in other words it is the same as `8.0`.

The hash sign is 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`.

```
1.0
3.1415_9265_3589_7932
2#1.0000_0001#e-128
```

Figure 7. Valid real constants

Text literals Text is any valid UTF-8 sequence of printable or space characters surrounded by text delimiters, such as `"Hello Möndé"`. Except for line-terminating characters, the behavior when a text sequence contains control characters or invalid UTF-8 sequences is unspecified. However, implementations are encouraged to preserve the contents of such sequences.

2. At the moment, XL uses the largest native integer type on the machine (generally 64-bit) in its internal representations. The scanner detects overflow in integer constants.

The base text delimiters are the single quote `'` and the double quote `"`. They can be used to enclose any text that doesn't contain a line-terminating character. The same delimiter must be used at the beginning and at the end of the text. For example, `"Shouldn't break"` is a valid text surrounded by double quotes, and `'He said "Hi"'` is a valid text surrounded by single quotes.

In text surrounded by base delimiters, the delimiter can be inserted by doubling it. For instance, except for the delimiter, `'Shouldn''t break'` and `"He said ""Hi"""` are equivalent to the two previous examples.

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 `>>`.

When long text contains multiple lines of text, indentation is ignored up to the indentation level of the first character in the long text. For example, Figure 9 illustrates how long text indent is eliminated from the text being read.

```
"Hello World"
'Où Toto élabora ce plan çi'
<<This text spans
multiple lines>>
```

Figure 8. Valid text constants

Source code	Resulting text

`<< Long text can`
`  contain indentation`
`or not,`
`  it's up to you>>`
	`Long text can`
	`contain indentation`
	`or not,`
	`  it's up to you`

Figure 9. Long text and indentation

BUG: Indentation is not ignored entirely as it should in long text.

In general, comparing text literals involves only comparison of their value, not delimiters.

Name and operator symbols Names begin with an alphabetic character `A..Z` or `a..z` or any non-ASCII UTF-8 character, followed by the longuest possible sequence of alphabetic characters, digits or underscores. Two consecutive underscore characters are not allowed. Thus, `Marylin_Monroe`, `élaböràtion` or `j1` are valid XLR names, whereas `A-1`, `1cm` or `A__2` are not.

Operator symbols, or *operators*, begin with an ASCII punctuation character[3] which does not act as a special delimiter for text, comments or blocks. For example, `+` or `->` are operator symbols. An operator includes more than one punctuation character only if it has been declared in the syntax configuration file. For example, unless the symbol `%,` has been declared in the syntax, `3%,4%` will contain two operator symbols `%` and `,` instead of a single `%,` operator.

A special name, the empty name, exists only as a child of empty blocks such as `()`.

After parsing, operator and name symbols are treated identically. During parsing, they are treated identically except in the expression versus statement rule.

```
x
X12_after_transformation
α_times_π
+
-->
<<<>>>
```

Figure 10. Examples of valid operator and name symbols

3. Non-ASCII punctuation characters or digits are considered as alphabetic.

2.4 Structured nodes

Four structured node types represent combinations of nodes. They are:

1. Infix nodes, representing operations such as `A+B` or `A and B`
2. Prefix nodes, representing operations such as `+3` or `sin x`
3. Postfix nodes, representing operations such as `3%` or `3 cm`
4. Blocks, representing grouping such as `(A+B)` or `{lathe;rinse;repeat}`

Infix nodes An infix node has two children, one on the left, one on the right, separated by a name or operator symbol.

Infix nodes are used to separate statements with semi-colons `;` or line breaks `NEWLINE`.

Prefix and postfix nodes Prefix and postfix nodes have two children, one on the left, one on the right, without any separator between them. The only difference is in what is considered the "operation" and what is considered the "operand". 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.

Prefix nodes are used for functions. The default for a name or operator symbol that is not explicitly declared in the `xl.syntax` file or configured is to be treated as a prefix function. For example, `sin` in the expression `sin x` is treated as a function.

Block nodes Block nodes have one child bracketed by two delimiters.

The default configuration recognizes the following pairs as block delimiters:

- Parentheses, as in `(A)`
- Brackets, as in `[A]`
- Curly braces, as in `{A}`
- Indentation, as shown surrounding the `write` statements in Figure 4.

2.5 Parsing rules

The XLR parser only needs a small number of rules to parse any text:

1. Precedence
2. Associativity
3. Infix versus prefix versus postfix
4. Expression versus statement

These rules are detailed below.

Precedence Infix, prefix, postfix and block symbols are ranked according to their *precedence*, represented as a non-negative integer. The precedence can be given by the syntax configuration file, `xl.syntax`, or by special notations in the source code, as detailed in Section 2.6.

Symbols with higher precedence associate before symbols with lower precedence. For instance, if the symbol `*` has infix precedence value `300` and symbol `+` has infix precedence value `290`, then the expression `2+3*5` will parse as an infix `+` whose right child is an infix `*`.

The same symbol may receive a different precedence as an infix, as a prefix and as a postfix operator. For example, if the precedence of `-` as an infix is `290` and the precedence of `-` as a prefix is `390`, then the expression `3 - -5` will parse as an infix `-` with a prefix `-` as a right child.

The precedence associated to blocks is used to define the precedence of the resulting expression. This precedence is used primarily in the "expression versus statement" rule.

Associativity Infix operators can associate to their left or to their right.

The addition operator is traditionally left-associative, meaning that in `A+B+C`, `A` and `B` associate before `C`. The outer infix node has an infix node as its left child (with `A` and `B` as their children) and `C` as its right child.

Conversely, the semi-colon in XLR is right-associative, meaning that `A;B;C` is an infix node with an infix as the right child and `A` as the left child.

Operators with left and right associativity cannot have the same precedence. To ensure this, XLR gives an even precedence to left-associative operators, and an odd precedence to right-associative operators. For example, the precedence of `+` in the default configuration is `290` (left-associative), whereas the precedence of `^` is `395` (right-associative).

Infix versus Prefix versus Postfix During parsing, XLR needs to resolve ambiguities between infix and prefix symbols. For example, in `-A + B`, the minus sign `-` is a prefix, whereas the plus sign `+` is an infix. Similarly, in `A and not B`, the `and` word is infix, whereas the `not` word is prefix. The problem is exactly similar for names and operator symbols.

XLR resolves this ambiguity as follows[4]:

- 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.
- If the symbol has an infix precedence and a prefix precedence, and either a space following it, or no space preceding it, then it is an infix: the minus sign `-` in `A - B` is an infix, but the same character is a prefix in `A -B`.
- 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 `3+sin x` is a prefix.

In the first, second and last case, a symbol may be identified as a prefix without being given an explicit precedence. Such symbols are called *default prefix*. They receive a particular precedence known as *function precedence*.

Expression versus statement Another ambiguity is related to the way humans read text. In `write sin x, sin y`, most humans will read this as a `write` instruction taking two arguments. This is however not entirely logical: if `write` takes two arguments, then why shouldn't `sin` also take two arguments? In other words, why should this example parse as `write(sin(x),sin(y))` and not as `write(sin(x,sin(y)))`?

The reason is that we tend to make a distinction between "statements" and "expressions". This is not a distinction that is very relevant to computers, but one that exists in most natural languages, which distinguish whole sentences as opposed to subject or complement.

XLR resolves the ambiguity by implementing a similar distinction. The boundary is a particular infix precedence, called *statement precedence*. Intuitively, infix operators with a lower precedence separate statements, whereas infix operators with a higher precedence separate expressions. For example, the semi-colon `;` or `else` separate statements, whereas `+` or `and` separate instructions.

More precisely:

- If a block's precedence is above statement precedence, its content begins as an expression, otherwise it begins as a statement: `3` in `(3)` 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. The name `B` in `A+B` is an expression, but it is a statement in `A;B`.
- A similar rule applies after prefix nodes: `{X} write A, B` will give two arguments to `write`, whereas in `(x->x+1) sin x, y` the `sin` function only receives a single argument.
- A default prefix begins a statement if it's a name, an expression if it's a symbol: the name `write` in `write X` begins a statement, the symbol `+` in `+3` begins an expression.

2.6 Syntax configuration

The default XLR syntax configuration file looks like Figure 11.

4. All the examples are given based on the default `xl.syntax` configuration file.

```
INFIX
        11      NEWLINE
        21      ;
        25      ->
        30      loop when
        35      else into
        40      then require ensure
        41      with
        100     STATEMENT
        110     is as  while until
        120     written
        130     where
        200     DEFAULT
        210     ,
        221     := += -= *= /= ^= |= &=
        230     return
        240     and or xor
        250     in at
        251     of to
        260     ..
        270     = < > <= >= <>
        280     & |
        290     + -
        300     * / mod rem
        395     ^
        510     .
        520     :

PREFIX
        30      data
        120     exit loop
        125     case if return while until yield transform
        320     not in out constant variable const var
        340     @
        380     ! ~
        390     - + * /
        400     FUNCTION
        410     function procedure to type iterator
        500     &

POSTFIX
        400     ! ? % cm inch mm pt px
BLOCK
        5       INDENT UNINDENT
        21      '{' '}'
        400     '(' ')' '[' ']'

TEXT
        "<<" ">>"

COMMENT
        "//" NEWLINE
        "/*" "*/"
```

Figure 11. Default syntax configuration file

Spaces and indentation are not significant in a syntax configuration file. Lexical elements are

identical to those of XLR, as detailed in Section 2.3.

The significant elements are integer constants, names, symbols and text. Integer constants are interpreted as the precedence of names and symbols that follow them. Name and symbols can be given either with lexical names and symbols, or with text. A few names are reserved for use as keywords in the syntax configuration file:

- `INFIX` begins a section declaring infix symbols and precedence. In this section:
  - `NEWLINE` identifies line break characters in the source code
  - `STATEMENT` identifies the precedence of statements
  - `DEFAULT` identifies the precedence for symbols not otherwise given a precedence. This precedence should be unique.
- `PREFIX` begins a section declaring prefix symbols and precedence. In this section:
  - `FUNCTION` identifies the precedence for default prefix symbols, i.e. symbols identified as prefix that are not otherwise given a precedence.
- `POSTFIX` begins a section declaring postfix symbols and precedence.
- `BLOCK` begins a section declaring block delimiters and precedence. In this section:
  - `INDENT` and `UNINDENT` are used to mark indentation and unindentation.
- `TEXT` begins a section declaring delimiters for long text.
- `COMMENT` begins a section declaring delimiters for comments. In this section:
  - `NEWLINE` identifies line breaks

Syntax information can also be provided in the source code using the `syntax` name followed by a block, as illustrated in Figure 12.

```
// Declare infix 'weight' operator
syntax (INFIX 350 weight)
A weight B -> A = B

// Declare postfix 'apples' and 'kg'
syntax
    POSTFIX 390 apples kg
X kg -> X * 1000
N apples -> N * 0.250 kg

// Combine the notations
if 6 apples weight 1.5 kg then
    write "Success!"
```

Figure 12. Use of the `syntax` specification in a source file

3 Semantics

The semantics of XLR is based entirely on the rewrite of XL0 abstract syntax trees. The rewrite rules are based on a very limited number set of operators. The rewrites operations combine to result in the execution of an XLR program.

3.1 Operators

XLR defines a small number of operators:

- Rewrite declarations are used to declare operations. They roughly play the role of functions and operator declarations in other programming languages.

- Data declarations identify data structures in the program.
- Type declarations define the type of variables.
- Guards limit the validity of rewrite or data declarations.
- Assignment allow a variable value to change over time.
- Sequence operators indicate the order in which computations must be performed.

Rewrite declarations The infix `->` operator declares a tree rewrite. Figure 13 repeats the code in Figure 3 illustrating how rewrite declarations can be used to define the traditional `if-then-else` statement.

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

Figure 13. Examples of tree rewrites

The tree on the left of the `->` operator is called the *pattern*. The tree on the right is called the *implementation* of the pattern. The rewrite declaration indicates that a tree that matches the pattern should be rewritten using the implementation.

The pattern contains *constant* and *variable* symbols and names:

- Infix symbols and names are constant
- Block-delimiting symbols and names are constant
- A name on the left of a prefix is a constant
- A name on the right of a postfix is a constant
- All other names are variable

Figure 14 highlight in blue italic type all variable symbols in the declarations of Figure 13.

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

Figure 14. Examples of tree rewrites

Constant symbol and names form the structure of the pattern, whereas variable names form the parts of the pattern which can match other trees. For example, to match the pattern in Figure 13, the `if`, `then` and `else` words must match exactly, but `TrueClause` may match any tree, like for example `write "Hello"`.

A special case is a rewrite declaration where the pattern consists of a single name.

- If the rewrite declaration is the only child of a block, the pattern name is variable, and the result is called a *universal rewrite rule*, i.e. a rule that applies to any tree. For example `(x->x+1)` is a universal rewrite rule that adds one to any input tree.
- Otherwise, the name is constant, and the rewrite is said to be a *named tree*.

Data declaration The `data` prefix declares tree structures that need not be rewritten further. For instance, Figure 15 declares that `1,3,4` should not be evaluated further, because it is made of infix `,` trees which are declared as `data`.

```
data a, b
```

Figure 15. Declaring a comma-separated list

The tree following a data declaration is a pattern, with constant and variable symbols like for rewrite declarations. In Figure 16, the names `x` and `y` are variable, but the name `complex` is constant because it is a prefix. Assuming that addition is implemented for integer values, `complex(3+4, 5+6)` will evaluate as `complex(7,11)` but no further. The declaration in Figure 16 can therefore be interpreted as declaring the `complex` data type.

```
data complex(x,y)
```

Figure 16. Declaring a `complex` data type

Type declaration The infix colon `:` operator in a rewrite or data patten introduces a *type declaration*. It indicates that the left node has the type indicated by the right node. Types in XLR are explained in Section 3.4.

Figure 17 shows examples of type declarations. The first line declares a `complex` type where the coordinates must be `real`. The second line declares an addition between a `real` number called `a` and an `integer` number called `b` implementing by calling the `integer_to_real` function on `b` (which presumably promotes the integer to a real number).

```
data complex(x:real, y:real)
a:real+b:integer -> a + integer_to_real b
```

Figure 17. Simple type declarations

REVISIT: Role of type declarations outside of a pattern, if any.

Guards The infix `when` operator in a rewrite or data pattern introduces a *guard*, i.e. a boolean condition that must be true for the pattern to apply.

Figure 18 shows an improved definition of the factorial function which only applies for non-negative values. Note that the definition does not give a type to the variable `N`, making it possible for `N` to be for example a `big_integer`.

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

Figure 18. Guard limit the validity of operations

TODO: Guards are not currently implemented.

Assignment The assignment operator `:=` binds the variable on its left to the tree on its right. The variable must not already be bound to a named tree. In other words, it is not legal to write `A:=5` following `A->3`.

If the left side of the assignment is a simple name, no evaluation of the tree on the right is done as a result of the assignment. However, if the variable name is evaluated for any reason, the variable will subsequently be bound to the evaluated result.

An assignment is valid even if the variable has not previously been bound. Assignments declare the variable being assigned to in the code following the assignment.

If the left side of an assignment is a type declaration, the assigned value is evaluated and the result compared against the type. Furthermore, any subsequent assignment to the same variable will also evaluate the assigned value and compare it against the type. If the type doesn't match, the assignment will be equivalent to evaluating `type_error X`, where `X` is the tree for the assignment.

If the left side of an assignment is not a name nor a type declaration, the assignment expression follows the normal evaluation rules described in Section 3.2.

Using an assignment in an expression is equivalent to using the value bound to the variable after the assignment. For instance, `sin(x:=f(0))` is equivalent to `x:=f(0)` followed by `sin(x)`.

TODO: Assignment is not currently implemented

REVISIT: What if the left-hand side of assignment is a rewrite? Consider this:

```
array 0 -> A
array 1 -> B
array X := 3
```

Figure 19. Assignment to an array

Sequences The infix line-break `NEWLINE` and semi-colon `;` operators are used to introduce a sequence between statements. They ensure that the left node is evaluated entirely before the evaluation of the right node begins.

Figure 20 for instance guarantees that the code will first `write "A"`, then `write "B"`, then write the result of the sum `f(100)+f(200)`. However, the implementation is entirely free to compute `f(100)` and `f(200)` in any order, including in parallel.

```
write "A"; write "B"
write f(100)+f(200)
```

Figure 20. Code writing `A`, then `B`, then `f(100)+f(200)`

Note: tasking and threading operations are not defined yet, but they need not be defined in the core language as described in this document (any more than I/O operations).

3.2 Evaluation

Except for special forms, the evaluation of XLR trees is performed as follows:

1. The tree to evaluate, `T`, is matched against the available data and rewrite pattern. `3*4+5` will match `A*B+C` as well as `A+B` (since the outermost tree is an infix `+` as in `A+B`).
2. Possible matches are tested in *scoping order* (defined below) against the tree to evaluate. The first matching tree is selected. For example, in Figure 1, `(N-1)!` will be matched against the rules `0!` and `N!` in this order.
3. Nodes in each candidate pattern `P` are compared to the tree `T` as follows:
   - Constant symbols or names in `P` are compared to the corresponding element in `T` and must match exactly. For example, the `+` symbol in pattern `A+B` will match the plus `+` symbol in expression `3*4+5` but not the minus `-` symbol in `3-5`.
   - Variables names in `P` that are not bound to any value and are not part of a type declaration are bound to the corresponding fragment of the tree in `T`. For example, for the expression `3!`, the variable `N` in Figure 1 will be bound to `3`.
   - Variable names in `P` that are bound to a value are compared to the corresponding tree fragment in `T` after evaluation. For instance, if `true` is bound at the point of the declaration in Figure 13, the test `if A<3 then X else Y` requires the evaluation of the expression `A<3`, and the result will be compared against `true`.
   - This rule applies even if the binding occured in the same pattern. For example, the pattern `A+A` will match `3+3` but not `3+4`, because `A` is first bound to `3` and then cannot match `4`. The pattern `A+A` will also match `(3-1)+(4-2)`: although `A` may first be bound to the unevaluated value `3-1`, verifying if the second `A` matches requires evaluation both `A` and the test value.
   - Type declarations in `P` match if the result of evaluating the corresponding fragment in `T` has the declared type. In that case, the variable being declared is be bound to the evaluated value.
   - Constant values (integer, real and text) in `P` are compared to the corresponding fragment of `T` after evaluation. For example, in Figure 1, when the expression `(N-1)!` is compared against `0!`, the expression `(N-1)` is evaluated in order to be compared to `0`.
   - Infix, prefix and postfix nodes in `P` are compared to the matching node in `T` by comparing their children in depth-first, left to right order.

   The comparison process may cause fragments of the tree to be evaluated. Each fragment is evaluated at most once for the process of evaluating the tree `T`. Once the fragment has been evaluated, the evaluated value will be used in any subsequent comparison or variable binding. For example, when computing `F(3)!`, the evaluation of `F(3)` is required in order to compare to `0!`, guaranteeing that `N` in `N!` will be bound to the evaluated value if `F(3)` is not equal to `0`.

4. If there is no match found between any pattern `P` and the tree to evaluate `T`:
   - Terminal nodes (integer, real, text, names and symbols) evaluate as themselves.
   - Other nodes `X` evaluate as `evaluation_error X`, which is supposed to implement error handling during evaluation.
     TODO: Implemented as `error "No form matches $1", X` today, which doesn't allow fine-grained error checking.
     REVISIT: There are a few other options:
     - Evaluate children
     - Evaluate children and retry top-level evaluation (unbounded?)
     - Return the tree unchanged
5. Otherwise, variables in the first matching pattern are bound to the corresponding fragment of the tree to evaluate. If the fragment was evaluated (including as required for comparison with an earlier pattern), then the variable is bound with the evaluated version. If the fragment was never evaluated, the variable is bound with the tree fragment.
6. The implementation corresponding to the pattern in the previous step is evaluated.

Special forms Some forms have a special meaning and are evaluated specially:

1. A terminal node (integer, real, type, name) evaluates as itself, unless there is an explicit rewrite rule for it[5].
2. A rewrite rule evaluates as itself.
3. A data declaration evaluates as itself
4. An assignment binds the variable and evaluates as the named variable after assignment
5. A prefix with a universal rewrite rule on the left and a tree on the right binds the tree on the right to the pattern variable and evaluates the implementation. For instance, the prefix `(x->x+1) 3` evaluates as `4`. TODO: Not implemented yet
6. A sequence evaluates the left tree, then evaluates the right tree, and evaluates as the result of the latter evaluation.

3.3 Variable binding and scoping

Binding a variable means that the variable references a particular tree. The binding is valid for a particular *scope*.

- For patterns, the scope of a binding includes tree nodes that follow the first occurence of the name in the source code, i.e. they would be found later in a left-to-right depth-first traversal.
- For rewrite declarations, the scope of the binding includes the pattern as described above as well as the implementation.
- For data declarations, the scope of the binding includes the pattern as described above.
- For assignments, the scope begins at the assignment and finishes at the end of the innermost block in which the assignment first happens.

3.4 Types

Types are expressions that appear on the right of the colon operator `:` in type declarations. In XLR, a type identifies the *shape* of a tree. A value is said to *belong* to a type if it matches the shape defined by the type.

5. There several use cases for allowing rewrite rules for integer, real or text constants, notably to implement data maps such as `(1->0; 0->1)`.

REVISIT: The type system is still largely unimplemented.

Predefined types The following predefined types are defined like the built-in operations described in Section 3.5:

- `integer` matches integer constants
- `real` matches real constants
- `text` matches text constants
- `symbol` matches names and operator symbols
- `name_symbol` matches names
- `infix` matches infix trees
- `prefix` matches prefix trees
- `postfix` matches postfix trees
- `block` matches block trees
- `tree` matches any tree

Note that `tree` is treated specially in type declarations in that they do not require evaluation of the corresponding tree. In other words, `x:tree` is equivalent to `x` in a pattern.

Value types Integer, real and text values can be used as types that have a single element, the exact value. For example, `x:37` is a type declaration which only matches when `x` is bound to the `integer` value `37`. This is mostly useful in structure and union types.

Block type If `T` is a type, then any block with `T` as a child such as `(T)` is the same type. Using the notation `(T)` may be required for precedence reasons.

Union type If `T1` and `T2` are types, then `T1|T2` is a type that matches any value belonging either to `T1` or to `T2`. For example, `x:(0|1|2|3)` indicates that `x` can be any of the four values.

Structure types If `P` is pattern (in the same sense as for rewrite rules), `type(P)` is a type matching the pattern. For example, `type(X,Y)` matches `3,5` or `A+3,write X`. Note that `type(X|Y)` matches an infix tree with the `|` name, whereas `X|Y` matches type `X` or type `Y`.

Named types A complex type expression can be bound to a name, which can then be used as a replacement for the entire type expression. For example, one can declare a `number` type that accepts `integer` and `real` numbers using code similar to Figure 21:

```
number -> integer | real
increment x:number -> x+1
```

Figure 21. Binding a type expression to a name

Rewrite types The infix `->` operator can be used in type expressions to denote rewrites that perform a particular kind of tree transformation. Figure 22 illustrates this usage:

```
adder : (number + number -> number)
```

Figure 22. Declaration of a rewrite type

REVISIT: Not sure about the semantics of rewrite types.

3.5 Built-in operations

A number of operations are defined by the core run-time of the language, and appear in a scope that precedes any XLR program.

This section decsribes the minimum list of operations available in any XLR program. Operator priorities are defined by the `xl.syntax` file in Figure 11. All operations listed in this section may be implemented specially in the compiler, or using regular rewrite rules defined in a particular file called `builtins.xl` that is loaded by XLR before evaluating any program, or a combination of both.

Arithmetic Arithmetic operators for `integer` and `real` values are listed in Table 2, where `x` and `y` denote integer or real values. Arithmetic operators take arguments of the same type and return an argument of the same type. In addition, the power operator `^` can take a first `real` argument and an `integer` second argument.

Form	Description
`x+y`	Addition
`x-y`	Subtraction
`x*y`	Multiplication
`x/y`	Division
`x rem y`	Remainder
`x mod y`	Modulo
`x^y`	Power
`-x`	Negation
`x%`	Percentage

Table 1. Arithmetic operations

Comparison Comparison operators can take `integer`, `real` or `text` argument, both arguments being of the same type, and return a `boolean` argument, which can be either `true` or `false`. Text is compared using the lexicographic order[6].

Form	Description
`x=y`	Equal
`x<>y`	Not equal
`x<y`	Less-than
`x>y`	Greater than
`x<=y`	Less or equal
`x>=y`	Greater or equal

Table 2. Comparisons

Bitwise arithmetic Bitwise operators operate on the binary representation of `integer` values, treating each bit indivudally.

Form	Description
`x<<y`	Shift `x` left by `y` bits
`x>>y`	Shift `x` right by `y` bits
`x and y`	Bitwise and
`x or y`	Bitwise or
`x xor y`	Bitwise exclusive or
`not x`	Bitwise complement

Table 3. Bitwise arithmetic operations

Boolean operations Boolean operators operate on the names `true` and `false`.

6. There is currently no locale-dependent text comparison.

Form	Description
`x=y`	Equal
`x<>y`	Not equal
`x and y`	Logical and
`x or y`	Logical or
`x xor y`	Logical exclusive or
`not x`	Logical not

Table 4. Boolean operations

Mathematical functions Mathematical functions operate on `real` numbers. The `random` function can also take two `integer` arguments, in which case it returns an `integer` value.

Form	Description
`sqrt x`	Square root
`sin x`	Sine
`cos x`	Cosine
`tan x`	Tangent
`asin x`	Arc-sine
`acos x`	Arc-cosine
`atan x`	Arc-tangent
`atan(y,x)`	Coordinates arc-tangent (`atan2` in C)
`exp x`	Exponential
`expm1 x`	Exponential minus one
`log x`	Natural logarithm
`log2 x`	Base 2 logarithm
`log10 x`	Base 10 logarithm
`log1p x`	Log of one plus x
`pi`	Numerical constant π
`random`	A random value between 0 and 1
`random x,y`	A random value between `x` and `y`

Table 5. Mathematical operations

Text functions Text functions operate on `text` values.

Form	Description
`x&y`	Concatenation
`length x`	Length of the text
`x at y`	Character at position `y` in `x`

Table 6. Text operations

Conversions Conversions operations transform data from one type to another.

Form	Description
`real x:integer`	Convert integer to real
`real x:text`	Convert text to real
`integer x:real`	Convert real to integer
`integer x:text`	Convert text to real
`text x:integer`	Convert integer to text
`text x:real`	Convert real to text

Table 7. Conversions

Date and time Date and time functions manipulates time. Time is expressed with an integer representing a number of seconds since a time origin. Except for `system_time` which never takes an argument, the functions can either take an explicit time represented as an `integer` as returned by `system_time`, or apply to the current time in the current time zone.

Form	Description
`hours`	Hours
`minutes`	Minutes
`seconds`	Seconds
`year`	Year
`month`	Month
`day`	Day of the month
`week_day`	Day of the week
`year_day`	Day of the year
`system_time`	Current time in seconds

Table 8. Date and time

Tree operations Tree operations are intended to manipulate trees.

Form	Description
`identity x`	Returns `x`
`do x`	Forces evaluation of `x`
`x with y`	Map, Reduce or Filter
`x at y`	Find element at index `y` in `x`
`x..y`	Create a list of elements between `x` and `y`
`x,y`	Used to create data lists
`()`	The empty list

Table 9. Tree operations

The map, reduce and filter operations act on data in an infix-separated list `y`. The convention is to use comma-separated lists, such as `1,3,5,6`, but other infix separators can be used.

- Map: If `x` is a name or a universal rewrite rule, a list is built by mapping the name to each element of the list in turn. For example, `sin with (1,3)` returns the list `sin 1, sin 3`, and `(x->x+1) with (2,4)` returns the list `3,5`.
- Reduce: If `x` is a rewrite rule with an infix on the left, the infix is considered as the separator for the list, and list elements in `y` are reduced by applying the rewrite rule to successive elements. For example, `(x,y->x+y) with (1,2,3)` returns `6`.
- Filter: If `x` is a guard, a new list is built by filtering elements of `y` matching the guard condition. For example, `(x when x < 0) with (1,2,-3,2,-1)` returns `-3,-1`.

The notation `x..y` is a shortcut for the comma-separated list of elements composed of elements between `x` and `y` inclusive. For example, `1..3` is the list `1,2,3`.

Data loading operations Data loading operations read a data file and convert it to an XLR parse tree suitable for XLR data manipulation functions. The arguments are:

- The file `f` is specified by its name as `text`.
- A prefix name `p` can be specified as `text`.
  - If the text is empty, the data is returned as `NEWLINE` separated rows of comma-separated elements.
  - Otherwise, each row is prefixed with the specified prefix.
- The field separator `fs` and row separator `rs` are `text` describing characters to be interepreted as separating fields and rows respectively.

Form	Description
`load f`	Load a tree as an XLR program
`load_csv f, p`	Load comma-separated data
`load_tsv f, p`	Load tab-separated data
`load_data f, p, fs, rs`	Load field-separated data

Table 10. Data loading operations

3.6 Lazy evaluation

Form parameters are evaluated lazily, meaning that the evaluation of one fragment of the form does not imply the evaluation of another. This makes it possible to evaluate infinite lists.

```
data x,y
integers_starting_from N:integer -> N, integers_starting_from (N+1)
```

Figure 23. Lazy evaluation of an infinite list

3.7 Controlled compilation

A special form, `compile`, is used to tell the compiler how to compile its argument. This makes it possible to implement special optimization for often-used forms.

```
compile {if Condition then TrueForm else FalseForm} ->
    generate_if_then_else Condition, TrueForm, FalseForm
```

Figure 24. Controlled compilation

Controlled compilation depends on low-level compilation primitives provided by the LLVM infrastructure[7], and assumes a good understanding of LLVM basic operations. Table 11 shows the correspondance between LLVM primitives and primitives that can be used during controlled compilation.

XLR Form	LLVM Entity	Description
`llvm_value x`	`Value *`	The machine value associated to tree `x`
`llvm_type x`	`Type *`	The type associated to tree `x`
`llvm_struct x`	`StructType *`	The structure type for signature `x`
`llvm_function_type x`	`FunctionType *`	The function type for signature `x`
`llvm_function x`	`Function *`	The machine function associated to `x`
`llvm_global x`	`GlobalValue *`	The global value identifying tree `x`
`llvm_bb n`	`BasicBlock *`	A basic block with name `n`
`llvm_type`		

Table 11. LLVM operations

3.8 Type inference

4 Example code

5 Implementation notes

This section describes the implementation as published at `http://xlr.sourceforge.net`.

7. For details, refer to `http://llvm.org`.

5.1 Tree representation

The tree representation is performed by the `Tree` class, with one subclass per node type: `Integer`, `Real`, `Text`, `Name`, `Infix`, `Prefix`, `Postfix` and `Block`.

The `Tree` structure has template members `GetInfo` and `SetInfo` that make it possible to associate arbitrary data to a tree. Data is stored there using a class deriving from `Info`.

The rule of thumb is that `Tree` only contains members for data that is used in the evaluation of any tree. Other data is stored using `Info` entries.

Currently, data that is directly associated to the `Tree` includes:

- The `tag` field stores the kind of the tree as well as its position in the source code.
- The `info` field is a linked list of `Info` entries.

5.2 Evaluation of trees

Trees are evaluated in a given *context*, representing the evaluation environment. The context contains a lexical (static) and stack (dynamic) part.

1. The *lexical context* represents the declarations that precede the tree being evaluated in the source code. It can be determined statically.
2. The *dynamic context* represents the declarations that were introduced as part of earlier evaluation, i.e. in the "call stack".

A context is represented by a tree holding the declarations, along with associated code.

5.3 Tree position

The position held in the `tag` field is character-precise. To save space, it counts the number of characters since the begining of compilation in a single integer value.

The `Positions` class defined in `scanner.h` maps this count to the more practical file-line-column positioning. This process is relatively slow, but this is acceptable when emitting error messages.

5.4 Actions on trees

Recursive operations on tree are performed by the `Action` class. This class implements virtual functions for each tree type called `DoInteger`, `DoReal`, `DoText` and so on.

5.5 Symbols

The XL runtime environment maintains symbol tables which form a hierarchy. Each symbol table has a (possibly `NULL`) parent, and contains two kinds of symbols: names and rewrites.

- Names are associated directly with a tree value. For example, `X->0` will associate the value `0` to name `X`.
- Rewrites are used for more complex tree rewrites, e.g. `X+Y->add X,Y`.

5.6 Evaluating trees

A tree is evaluated as follows:

1. Evaluation of a tree is performed by `xl_evaluate()` in `runtime.cpp`.
2. This function checks the stack depth to report infinite recursion.
3. If `code` is `NULL`, then the tree is compiled first.
4. Then, evaluation is performed by calling `code` with the tree as argument.

The signature for `code` is a function taking a `Tree` pointer and returning a `Tree` pointer.

5.7 Code generation for trees

Evaluation functions are functions with the signature shown in Figure 25:

```
Tree * (*eval_fn) (eval_fn eval, Tree *self)
```

Figure 25. Signature for rewrite code with two variables.

Unfortunately, the signature in Figure 25 is not valid in C or C++, so we need a lot of casting to achieve the desired effect.

In general, the `code` for a tree takes the tree as input, and returns the evaluated value.

However, there are a few important exceptions to this rule:

Right side of a rewrite If the tree is on the right of a rewrite (i.e. the right of an infix `->` operator), then `code` will take additional input trees as arguments. Specifically, there will be one additional parameter in the code per variable in the rewrite rule pattern.

For example, if a rewrite is `X+Y->foo X,Y`, then the `code` field for `foo X,Y` will have `X` as its second argument and `Y` as its third argument, as shown in Figure 25.

In that case, the input tree for the actual expression being rewritten remains passed as the first argument, generally denoted as `self`.

Closures If a tree is passed as a `tree` argument to a function, then it is encapsulated in a *closure*. The intent is to capture the environment that the passed tree depends on. Therefore, the associated `code` will take additional arguments representing all the captured values. For instance, a closure for `write X,Y` that captures variables `X` and `Y` will have the signature shown in Figure 26:

```
Tree * (*code) (Tree *self, Tree *X, Tree *Y)
```

Figure 26. Signature for rewrite code with two variables.

At runtime, the closure is represented by a prefix tree with the original tree on the left, and the captured values cascading on the right. For consistency, the captured values are always on the left of a `Prefix` tree. The rightmost child of the rightmost `Prefix` is set to an arbitrary, unused value (specifically, `false`).

Closures are built by the function `xl_new_closure`, which is generally invoked from generated code. Their `code` field is set to a function that reads all the arguments from the tree and invokes the code with the additional arguments.

For example, `do` takes a `tree` argument. When evaluating `do write X,Y`, the tree given as an argument to `do` depends on variable `X` and `Y`, which may not be visible in the body of `do`. These variables are therefore captured in the closure. If its values of `X` and `Y` are `42` and `Universe`, then `do` receives a closure for `write X,Y` with arguments `42` and `Universe`.

5.8 Tail recursion

5.9 Partial recompilation