XLR - eXtansible Language and Runtime

Benedikt Tuschter

Leopold-Franzens-University of Innsbruck

Instructor: Moser Georg

June 12, 2015

Overview

XL Extensible Language

XLR stands for "eXtensible Language and Runtime", it is a Dynamic Language based on meta-programming and makes it easy to write simple programs

[1] project side at http://xlr.sf.net

XL Extensible Language

XLR stands for "eXtensible Language and Runtime", it is a Dynamic Language based on meta-programming and makes it easy to write simple programs

XL is defined at four different Levels

- XL0 defines how an input text is transformed into a parse tree.

[1] project side at http://xlr.sf.net

XL Extensible Language

XLR stands for "eXtensible Language and Runtime", it is a Dynamic Language based on meta-programming and makes it easy to write simple programs

XL is defined at four different Levels

- XL0 defines how an input text is transformed into a parse tree.
- XL1 defines a base language with features comparable to C++.

[1] project side at http://xlr.sf.net

XL Extensible Language

XLR stands for "eXtensible Language and Runtime", it is a Dynamic Language based on meta-programming and makes it easy to write simple programs

XL is defined at four different Levels

- XL0 defines how an input text is transformed into a parse tree.
- XL1 defines a base language with features comparable to C++.
- XL2 defines the standard library, which includes common data types and operators.

[1] project side at http://xlr.sf.net

XL Extensible Language

XLR stands for "eXtensible Language and Runtime", it is a Dynamic Language based on meta-programming and makes it easy to write simple programs

XL is defined at four different Levels

- XL0 defines how an input text is transformed into a parse tree.
- XL1 defines a base language with features comparable to C++.
- XL2 defines the standard library, which includes common data types and operators.
- **XLR** defines a dynamic runtime for XL based on XL0.

[1] project side at http://xlr.sf.net

Syntax

- **Comments and Spaces**, like in C
- **Literals**: Integer, Real, Text, Symbols and names
- **Structured nodes**: infix, prefix, postfix, blocks

Examples

```
//integer and real
   2#1001     =    9
   3.14
   1.0e-3     = 0.001
/*separator*/
   1_000_000 = 1000000
```

```
A+B    //infix
3!     //postfix
-3     //prefix
(A+B) //block

"Hello World", <<multiple
lines>>
```

Syntax - Parsing rules

The XLR parser only needs a small number of rules to parse any code into XL0:

Syntax - Parsing rules

The XLR parser only needs a small number of rules to parse any code into XL0:

- **Precedence**
  structured nodes are ranked

Syntax - Parsing rules

The XLR parser only needs a small number of rules to parse any code into XL0:

- **Precedence**
  structured nodes are ranked
- **Associativity**
  infix operators can associate to their left or right child

Syntax - Parsing rules

The XLR parser only needs a small number of rules to parse any code into XL0:

- **Precedence**
  structured nodes are ranked
- **Associativity**
  infix operators can associate to their left or right child
- **Infix vs Prefix vs Postfix**
  resolve ambiguities between infix and prefix, i.e -A + B

Semantics - Tree rewrite operators

The infix operator $\rightarrow$ declares a tree rewrite. The tree on the left is called *Pattern* and on the right *Implementation*

- **Rewrite declarations** $\rightarrow$ declare operations (*pattern* $\rightarrow$*impl*)

Semantics - Tree rewrite operators

The infix operator $\rightarrow$ declares a tree rewrite. The tree on the left is called *Pattern* and on the right *Implementation*

- **Rewrite declarations** $\rightarrow$ declare operations (*pattern* $\rightarrow$*impl*)
- **Data declaration** $\rightarrow$ identify data structures (*data Pattern*)

Semantics - Tree rewrite operators

The infix operator $\rightarrow$ declares a tree rewrite. The tree on the left is called *Pattern* and on the right *Implementation*

- **Rewrite declarations** $\rightarrow$ declare operations (*pattern* $\rightarrow$*impl*)
- **Data declaration** $\rightarrow$ identify data structures (*data Pattern*)
- **Type declaration** $\rightarrow$ define type of variable (*Name:type*)

Semantics - Tree rewrite operators

The infix operator $\rightarrow$ declares a tree rewrite. The tree on the left is called *Pattern* and on the right *Implementation*

- **Rewrite declarations** $\rightarrow$ declare operations (*pattern* $\rightarrow$*impl*)
- **Data declaration** $\rightarrow$ identify data structures (*data Pattern*)
- **Type declaration** $\rightarrow$ define type of variable (*Name:type*)
- **Guards** $\rightarrow$ limit the validity of rewrite or data declaration (*when*)

Semantics - Tree rewrite operators

The infix operator $\rightarrow$ declares a tree rewrite. The tree on the left is called *Pattern* and on the right *Implementation*

- **Rewrite declarations** $\rightarrow$ declare operations (*pattern* $\rightarrow$*impl*)
- **Data declaration** $\rightarrow$ identify data structures (*data Pattern*)
- **Type declaration** $\rightarrow$ define type of variable (*Name:type*)
- **Guards** $\rightarrow$ limit the validity of rewrite or data declaration (*when*)
- **Sequence operators** $\rightarrow$ order of computations (*stm1 ; stm2*)

Semantics - Tree rewrite operators

The infix operator $\rightarrow$ declares a tree rewrite. The tree on the left is called *Pattern* and on the right *Implementation*

- **Rewrite declarations** $\rightarrow$ declare operations (*pattern* $\rightarrow$*impl*)
- **Data declaration** $\rightarrow$ identify data structures (*data Pattern*)
- **Type declaration** $\rightarrow$ define type of variable (*Name:type*)
- **Guards** $\rightarrow$ limit the validity of rewrite or data declaration (*when*)
- **Sequence operators** $\rightarrow$ order of computations (*stm1 ; stm2*)
- **Index operators** $\rightarrow$ perform "structures" or "arrays" (*A[]*) or *A.B*)

Semantics - Tree rewrite operators

The infix operator $\rightarrow$ declares a tree rewrite. The tree on the left is called *Pattern* and on the right *Implementation*

- **Rewrite declarations** $\rightarrow$ declare operations (*pattern* $\rightarrow$*impl*)
- **Data declaration** $\rightarrow$ identify data structures (*data Pattern*)
- **Type declaration** $\rightarrow$ define type of variable (*Name:type*)
- **Guards** $\rightarrow$ limit the validity of rewrite or data declaration (*when*)
- **Sequence operators** $\rightarrow$ order of computations (*stm1 ; stm2*)
- **Index operators** $\rightarrow$ perform "structures" or "arrays" (*A[]*) or *A.B*)
- **C Interface** $\rightarrow$ import and using c functions

Types

Types are expressions that appear on the right of the colon operator in type declarations. *Types* identifies the *shape* of a tree.

- **integer**
- **real**
- **text**
- **symbol**
- **name**
- **operator**
- **infix**
- **prefix**
- **postfix**
- **block**
- **tree**
- **boolean**

Types - Type definition

A *type definition* for type T is a special form of tree rewrite declaration where the pattern has the form type X.
Name → type Pattern

Example

```
/*Simple type declaration*/
complex → type (re:real; im:real)

Z1:complex+Z2:complex → (Z1.re+Z2.re; Z1.im+Z2.im)
/* Possible resulting bindings in the implementation of + */
(3.4;5.2)+(0.4;2.22)


Z1 ->                          Z2 ->
    re->3.4                        re->0.4
    im->5.2                        im->2.22
    (re; im)                       (re;im)
```

Types - Properties and data inheritance

Properties are rewrite declarations and start with **properties**
Declaration:

- *Name:Type* or
- *Name:Type:=DefaultValue*

Inheritance starts with **inherit**

Example

```
color → properties
      red   : real
      green : real
      blue  : real
      alpha : real := 1.0
```

```
rgb → properties
      red   : real
      green : real
      blue  : real
rgba → properties
      inherit rgb
      alpha : real
```

Library - Built-in Operations

A number of operations are defined by the core run-time of the language, and appear in the context used to evaluate any XLR program.

Examples

- **Arithmetic**: x+y, x-y, x*y, x/y, x rem y, x mod y, x^y, -x, x%, x!

Library - Built-in Operations

A number of operations are defined by the core run-time of the language, and appear in the context used to evaluate any XLR program.

Examples

- **Arithmetic**: x+y, x-y, x*y, x/y, x rem y, x mod y, x^y, -x, x%, x!
- **Comparison**: $x = y, x <> y, x < y, x > y, x <= y, x >= y$

Library - Built-in Operations

A number of operations are defined by the core run-time of the language, and appear in the context used to evaluate any XLR program.

Examples

- **Arithmetic**: x+y, x-y, x*y, x/y, x rem y, x mod y, x^y, -x, x%, x!
- **Comparison**: $x = y, x <> y, x < y, x > y, x <= y, x >= y$
- **Bitwise arithmetic**: x shl y, x shr y, x and y, x or y, x xor y, not x

Library - Built-in Operations

A number of operations are defined by the core run-time of the language, and appear in the context used to evaluate any XLR program.

Examples

- **Arithmetic**: x+y, x-y, x*y, x/y, x rem y, x mod y, x^y, -x, x%, x!
- **Comparison**: $x = y, x <> y, x < y, x > y, x <= y, x >= y$
- **Bitwise arithmetic**: x shl y, x shr y, x and y, x or y, x xor y, not x
- **Boolean operations**: $x = y$, $x <> y$, x and y, x or y, x xor y, not x

Library - Built-in Operations

A number of operations are defined by the core run-time of the language, and appear in the context used to evaluate any XLR program.

Examples

- **Arithmetic**: x+y, x-y, x*y, x/y, x rem y, x mod y, x^y, -x, x%, x!
- **Comparison**: $x = y, x <> y, x < y, x > y, x <= y, x >= y$
- **Bitwise arithmetic**: x shl y, x shr y, x and y, x or y, x xor y, not x
- **Boolean operations**: $x = y$, $x <> y$, x and y, x or y, x xor y, not x
- **Mathematical functions**: *sqrt* x, *sin* x, *log* x, *random* x, y , ...

Library - Built-in Operations

A number of operations are defined by the core run-time of the language, and appear in the context used to evaluate any XLR program.

Examples

- **Arithmetic**: x+y, x-y, x*y, x/y, x rem y, x mod y, x^y, -x, x%, x!
- **Comparison**: $x = y, x <> y, x < y, x > y, x <= y, x >= y$
- **Bitwise arithmetic**: x shl y, x shr y, x and y, x or y, x xor y, not x
- **Boolean operations**: $x = y$, $x <> y$, x and y, x or y, x xor y, not x
- **Mathematical functions**: *sqrt* x, *sin* x, *log* x, *random* x, y , ...
- **Text functions**: x&y, text_length x, ...

Library - Built-in Operations

A number of operations are defined by the core run-time of the language, and appear in the context used to evaluate any XLR program.

Examples

- **Arithmetic**: x+y, x-y, x*y, x/y, x rem y, x mod y, x^y, -x, x%, x!
- **Comparison**: $x = y, x <> y, x < y, x > y, x <= y, x >= y$
- **Bitwise arithmetic**: x shl y, x shr y, x and y, x or y, x xor y, not x
- **Boolean operations**: $x = y$, $x <> y$, x and y, x or y, x xor y, not x
- **Mathematical functions**: *sqrt* x, *sin* x, *log* x, *random* x, y , ...
- **Text functions**: x&y, text_length x, ...
- **Conversions**: real x:integer, integer x:text, ...

Library - Built-in Operations

A number of operations are defined by the core run-time of the language, and appear in the context used to evaluate any XLR program.

Examples

- **Arithmetic**: x+y, x-y, x*y, x/y, x rem y, x mod y, x^y, -x, x%, x!
- **Comparison**: $x = y, x <> y, x < y, x > y, x <= y, x >= y$
- **Bitwise arithmetic**: x shl y, x shr y, x and y, x or y, x xor y, not x
- **Boolean operations**: $x = y$, $x <> y$, x and y, x or y, x xor y, not x
- **Mathematical functions**: *sqrt* x, *sin* x, *log* x, *random* x, y , ...
- **Text functions**: x&y, text_length x, ...
- **Conversions**: real x:integer, integer x:text, ...
- **Date and Time**: system_time, hours, minutes, ...

Library - Built-in Operations

A number of operations are defined by the core run-time of the language, and appear in the context used to evaluate any XLR program.

Examples

- **Arithmetic**: x+y, x-y, x*y, x/y, x rem y, x mod y, x^y, -x, x%, x!
- **Comparison**: $x = y, x <> y, x < y, x > y, x <= y, x >= y$
- **Bitwise arithmetic**: x shl y, x shr y, x and y, x or y, x xor y, not x
- **Boolean operations**: $x = y$, $x <> y$, x and y, x or y, x xor y, not x
- **Mathematical functions**: *sqrt* x, *sin* x, *log* x, *random* x, y , ...
- **Text functions**: x&y, text_length x, ...
- **Conversions**: real x:integer, integer x:text, ...
- **Date and Time**: system_time, hours, minutes, ...
- **Tree operations**: identity x, do x, child x, ...

Library - Built-in Operations

A number of operations are defined by the core run-time of the language, and appear in the context used to evaluate any XLR program.

Examples

- **Arithmetic**: x+y, x-y, x*y, x/y, x rem y, x mod y, x^y, -x, x%, x!
- **Comparison**: $x = y, x <> y, x < y, x > y, x <= y, x >= y$
- **Bitwise arithmetic**: x shl y, x shr y, x and y, x or y, x xor y, not x
- **Boolean operations**: $x = y$, $x <> y$, x and y, x or y, x xor y, not x
- **Mathematical functions**: $sqrt\ x, sin\ x, log\ x, random\ x, y$, ...
- **Text functions**: x&y, text_length x, ...
- **Conversions**: real x:integer, integer x:text, ...
- **Date and Time**: system_time, hours, minutes, ...
- **Tree operations**: identity x, do x, child x, ...
- **List operations**: nil, length L, head L, tail L, map, filter, ...

Library - Control structures

The XLR Lib provides a number of loop constructs.

- **Loops**: while, until, for

Examples

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

/*for-Loop*/
for Variable in Low:integer..High:integer loop Body →
   Index : integer := Low
   while Index < High Loop
     Variable := Index
     Body
     Index := +1
```

Simple Example

Examples

```
//XLR
color → properties
        red   : real
        green : real
        blue  : real
        alpha : real := 1.0


complex → type (re:real;
                im:real)


Z1:complex+Z2:complex →
          (Z1.re+Z2.re;
           Z1.im+Z2.im)
```

```
//HASKELL
 data Color = Color
     { red   :: Double
     , green :: Double
     , blue  :: Double
     , alpha :: Double
     }

data Complex = Complex
          (Double,Double)
          deriving (Show)

instance Num Complex where
  Complex (re1, im1)  +
  Complex (re2, im2) =
  Complex (re1+re2,
           im1+im2)
```

Simple Example

Examples

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

```
//HASKELL
fac :: Double -> Double
  fac 0 = 1
  fac x = x * fac (x-1)
```

Conclusion

- exploring new ideas
- flexible
- procedural and functional approach
- aspects of object-orientated programming (*inheritance*)
- concept programming

Thanks for your attention !