XL

An extensible programming language

Christophe de Dinechin

XL is an extensible programming language, designed to accomodate a variety of programming needs with ease.

Being *extensible* means that the language is designed to make it very easy for programmers to adapt the language to suit their needs, for example by adding new programming constructs. In XL, extending the language is a routine operation, much like adding a function or creating a class in more traditional programming languages. This extensibility is demonstrated by the fact that operations that are built-in in other programming languages, such as integer arithmetic, basic types or loops, are part of the standard library in XL.

As a consequence of this extensibility, XL is intended to be suitable for programming tasks ranging from the simplest to the most complex, from documents and application scripting, as illustrated by Tao3D, to compilers, as illustrated by the XL2 self-compiling compiler to distributed programming, as illustrated by ELFE.

> XL is a work in progress. Even if there are some bits and pieces that happen to already work, and even if there were fully functioning releases like the XL version used in Tao3D in the past, XL is being totally reworked, and the compiler in this repository is presently not suitable for any serious programming. Examples given below may sometimes simply not work. Take it as a painful reminder that the work is far from finished, and, who knows, as an idea for a contribution. See HISTORY for how we came to the present mess. The README gives a quick overview of the language.

Chapter 1. Introduction to XL

Extensible? What does that mean for a programming language? For XL, it really means three things:

1. XL has a method to extend the language with any kind of feature, not just functions or data types, but also programming constructs, optimizations, domain-specific notations, and more. Actually, all this is done with a single operator, `is`, called the *definition operator*.
2. As a validation of the concept, most features that are built-in in other programming languages, like the `while` loop, or integer arithmetic, are *constructed* in XL. Specifically, they are provided by the standard library, using techniques that any programmer can use in their program. This, obviously, means that programmers can add their own loops, or their own machine-level data types, and even extend existing ones.
3. XL provides complete control over the program translation process. This means that libraries exist or can be written to make XL at least as good as C for low-level bit-twiddling, at least as good as C++ for generic algorithms, at least as good as Ada for tasking, at least as good as Fortran for numerical algorithms, at least as good as Java for distributed programming, and so on.

This may all seem too good to be true. This document explains how the magic happens. But first of all, one thing that really matters: XL is supposed to be *simple*. Let's start with a few well-known examples to prove this.

1.1. Two basic examples

It is practically compulsory to begin the presentation of any programming language with a "Hello World" example, immediately followed by a a recursive definition of the factorial function. Let's follow this long honored tradition.

1.1.1. Hello World

In XL, a program that prints `Hello World` on the terminal console output will look like this:

```
use XL.CONSOLE.TEXT_IO
print "Hello World"
```

The first line *imports* the `XL.CONSOLE.TEXT_IO` module. The program can then use the `print` function from that module to write the text on the terminal console.

Why do we need the `use` statement? There is a general rule in XL that you only pay for things that you use. Not all programs will use a terminal console, so the corresponding functions must be explicitly imported into a program. It is possible that some systems, like embedded systems, don't even have a terminal console. On such a system, the corresponding module would not be available, and the program would properly fail to compile.

What is more interesting, though, is the definition of `print`. That definition is discussed below, and you will see that it is quite simple, in particular when compared with similar input/output operations in languages such as C++.

Another interesting, if slightly more complicated version of "Hello World" is one written in the Tao3D dialect of XL that produces this result:

▶ https://www.youtube.com/watch?v=6WIMWlUZJvs *(YouTube video)*

Helo World in Tao3D

Example 1. Source code for the Tao3D "Hello World"

The source code for this example can be found below. The Tao3D dialect of XL still uses `->` instead of `is` as the definition operator. Apart from that change, the following code is valid XL for the language described in this document.

```
color "white"
milkyway 10000
rotatez -23
earth 400
hello_world 440

milkyway R ->
// ----------------------------------------------------------------------------
//    Draw the Milky Way
// ----------------------------------------------------------------------------
    locally
        texture_wrap true, true
        texture_transform {scale 5, 5, 5}
        texture "milkyway.jpg"
        rotatey 0.02 * page_time + 100
        scale 1, -1, 1
        sphere R


earth R ->
// ----------------------------------------------------------------------------
//    Draw Earth
// ----------------------------------------------------------------------------
    locally
        texture "earth.bmp"
        texture_wrap true, true
        rotatey 5 * page_time + 250
        sphere 0, 0, 0, R


hello_world R ->
// ----------------------------------------------------------------------------
//    Draw "hello world" text
// ----------------------------------------------------------------------------
    locally
        frame_texture 1900, 600,
            color 1, 1, 1, 1
            reset_transform
            // If font Arial Unicode installed, it will be used.
            // Otherwise, unifont will be used (unifont is packaged
            // with Tao presentations).
            font "Arial Unicode MS", "unifont", 72
            move_to -800, -9, 0
            text "Hello World! or Καλημέρα κ□σμε; or □□□□□ □□"
        rotatey -11 * page_time + 180
        color 20% , 20% , 20% , 70%
        sphere 0, 0, 0, R - 30
        color 100% , 90% , 20% , 90%
        sphere 0, 0, 0, R
```

1.1.2. Factorial

A program computing the factorial of numbers between 1 and 5, and then showing them on the console, can be written as follows:

```
use IO = XL.CONSOLE.TEXT_IO

0! is 1
N! is N * (N-1)!

for I in 1..5 loop
    IO.print "The factorial of ", I, " is ", I!
```

We have used an alternative form of the `use` statement, where the imported module is given a local nick-name, `IO`. This form is useful when it's important to avoid the risk of name collisions between modules. In that case, the programmer need to refer to the `print` function of the module as `IO.print`.

The definition of the factorial function shows how expressive XL is, making it possible to use the well-known notation for the factorial function. The definition consists in two parts:

- the special case of the factorial of `0` is defined as follows:

  ```
  0! is 1
  ```

- the general case is defined as follows, and involves a recursion in the form of the `(N-1)!` expression:

  ```
  N! is N * (N-1)!
  ```

That definition would not detect a problem with something like `-3!`. The second form would match, and presumably enter an infinite recursion that would exhaust available stack space. It is possible to fix that problem by indicating that the definition only works for positive numbers:

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

Writing the code that way will ensure that there is a compile-time error for code like `-3!`, because there is no definition that matches.

1.2. One operator to rule them all

XL has a single fundamental operator, `is`, called the *definition operator*. It is an infix operator with a pattern on the left and an implementation on the right. In other words, the pattern for the infix `is` is `Pattern is Implementation`, where `Pattern` is a program pattern, like `X+Y`, and `Implementation` is an implementation for that pattern, for example `Add X, Y`. This operator can also be read as *transforms into*, i.e. it transforms the code that is on the left into the code that is on the right.

This single operator can be used to define all kinds of entities.

Example 2. Simple variables or constants

```
pi              is      3.1415926
```

Example 3. Lists or data structures

```
funny_words     is      "xylophage", "zygomatic", "barfitude"
identity_matrix is
    [ [1, 0, 0],
      [0, 1, 0],
      [0, 0, 1] ]
```

Example 4. Functions

```
abs X:number    is      if X < 0 then -X else X
```

Example 5. Operators

```
X ≠ Y           is      (not X = Y)
```

Example 6. Specializations for particular inputs

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

Example 7. Notations using arbitrary combinations of operators

```
A in B..C       is      A >= B and A <= C
```

Example 8. Optimizations using specializations

```
X * 1           is      X
X + 0           is      X
```

Example 9. Program structures

```
loop Body       is      { Body; loop Body }     // Define an infnite loop
```

Example 10. Types

```
complex         is      polar or cartesian
cartesian       is      type cartesian(re:number, im:number)
polar           is      type polar(mod:number, arg:number)
```

types in XL indicate the shape of parse trees. In other words, the `cartesian` type above will match any parse tree that takes the shape of the word `cartesian` followed by two numbers, like for example `cartesian(1,5)`.

Example 11. Higher-order functions, i.e. functions that return functions

```
adder N         is      { lambda X is N + X }
add3            is      ( adder 3 )

 // This will compute 8
 add3 5
```

The notation `lambda X`, which can also be written `\X`, is inspired by lambda calculus. It makes it possible to create patterns that match entire expressions. In other words, `X is 0` defines a name, and only the expression `X` matches that definition, whereas `\X is 0` defines a "catch-all" pattern that will match `35` or `"ABC"`. This *lambda notation* can be used to build something that behaves almost exactly like an *anonymous function* in functional languages, although the way it actually works internally is still based on pattern matching.

> The current implementations of XL special-case single-defintion contexts, and `lambda` can be omitted in that case. In a normal context, `X is Y` defines a name `X`, but it did not seem very useful to have single-definition contexts defining only a name. The above example could have been written as:
>
> ```
> adder N is (X is N + X)
> ```
>
> However, this is not consistent with the rest of the language, and `lambda` will be required in future implementations.

Example 12. Maps that associate a key to a value

```
my_map is
    0 is 4
    1 is 0
    8 is "World"
    27 is 32
    lambda N when N < 45 is N + 1

// The following is "World"
my_map 8

// The following is 32
my_map[27]

// The following is 45
my_map (44)
```

This provides a functionality roughly equivalent to `std::map` in C++. However, it's really nothing more than a regular function with a number of special cases. The compiler can optimize special kinds of mapping to provide an efficient implementation, for example if all the indexes are contiguous integers.

Example 13. Templates (C++ terminology) or generic code (Ada terminology)

```
// An (inefficient) implementation of a generic 1-based array type
array[1] of T is type
    Value : T
    1 is Value
array[N] of T when N > 1 is type
    Head  : array[N-1] of T
    Tail  : T
    lambda I when I<N is Head[I]
    lambda I when I=N is Tail

A : array[5] of integer
for I in 1..5 loop
    A[I] := I * I
```

Example 14. Variadic functions

```
min X, Y    is { Z is min Y; if X < Z then X else Z }
min X       is X

// Computes 4
min 7, 42, 20, 8, 4, 5, 30
```

In short, the single `is` operator covers all the kinds of declarations that are found in other languages, using a single, easy to read syntax.

1.3. The standard library

Each programming language offers a specific set of features, which are characteristic of that language. Most languages offer integer arithmetic, floating-point arithmetic, comparisons, boolean logic, text manipulation (often called "*strings*"), but also programming constructs such as loops, tests, and so on.

XL provides most features programmers are used to, but they are defined in the XL *standard library*, not by the compiler. The standard library is guaranteed to be present in all implementations and behave identically. However, it is written using only tools that are available to a regular developer, not just to compiler writers.

1.3.1. Usual programming features

Definitions in the standard library include common fixtures of programming that are built-in in other languages, in particular well-known programming constructs such as loops, tests, and so on.

For example, the *if statement* in XL is defined in the standard library as follows:

```
if [[true]]  then TrueClause else FalseClause   is TrueClause   ①
if [[false]] then TrueClause else FalseClause   is FalseClause
if [[true]]  then TrueClause                    is TrueClause
if [[false]] then TrueClause                    is false
```

① A value between two square brackets, as in `[[true]]` and `[[false]]`, is called a metabox. It indicates that the pattern must match the actual values in the metabox. In other words, `foo true is ...` defines a pattern with a formal parameter named `true`, whereas `foo [[true]] is ...` defines a pattern which only matches when the argument is equal to constant `true`.

Similarly, the `while` loop is defined as follows:

```
while Condition loop Body is
    if Condition then
        Body
        while Condition loop Body
```

With the definitions above, programmers can then use `if` and `while` in their programs much like they would in any other programming language, as in the following code that verifies the Syracuse conjecture:

```
while N <> 1 loop
    if N mod 2 = 0 then
        N /= 2
    else
        N := N * 3 + 1
    print N
```

1.3.2. The next natural evolutionary step

Moving features to a library is a natural evolution for programming languages. Consider for example the case of text I/O operations. They used to be built-in for early languages such as BASIC's `PRINT` or Pascal's `WriteLn`, but they moved to the library in later languages such as C with `printf`. As a result, C has a much wider variety of I/O functions. The same observation can be made on text manipulation and math functions, which were all built-in in BASIC, but all implemented as library functions in C. For tasking, Ada has built-in construct, C has the `pthread` library. And so on.

Yet, while C moved a very large number of things to libraries, it still did not go all the way. The meaning of `x+1` in C is defined strictly by the compiler. So is the meaning of `x/3`, even if some implementations that lack a hardware implementation of division have to make a call to a library function to actually implement that code.

C++ went one step further than C, allowing programmers to *overload* operators, i.e. redefine the meaning of an operation like `X+1`, but only for custom data types, and only for already existing operators. In C++, a programmer cannot *create* the *spaceship operator* `<=>` using the standard language mechanisms. It has to be implemented in the compiler. The spaceship operator has to be added to the language by compiler writers, and it takes a 35-pages article to discuss the implications. This takes time and a large effort, since all compiler writers must implement the same thing.

By contrast, all it takes in XL to implement `<=>` in a variant that always returns `-1`, `0` or `1` is the following:

```
syntax { INFIX 290 <=> }
X <=> Y     when X < Y  is -1
X <=> Y     when X = Y  is  0
X <=> Y     when X > Y  is  1
```

Similarly, C++ makes it extremely difficult to optimize away an expression like `X*0`, `X*1` or `X+0` using only standard programming techniques, whereas XL makes it extremely easy:

```
X*0     is 0
X*1     is X
X+0     is X
```

Finally, C++ also makes it very difficult to deal with expressions containing multiple operators. For example, many modern CPUs feature a form of fused multiply-add, which has benefits that include performance and precision. Yet C++ will not allow you to overload `X*Y+Z` to use this kind of operations. In XL, this is not a problem at all:

```
X*Y+Z   is FusedMultiplyAdd(X,Y,Z)
```

In other words, the XL approach represents the next logical evolutionary step for programming languages along a line already followed by highly-successful ancestors.

1.3.3. Benefits of moving features to a library

Putting basic features in the standard library, as opposed to keeping them in the compiler, has several benefits:

1. Flexibility: It is much easier to offer a large number of behaviors and to address special cases.
2. Clarity: The definition given in the library gives a very clear and machine-verifiable description of the operation.
3. Extensibility: If the library definition is not sufficient, it is possible to add what you need. It will behave exactly as what is in the library. If it proves useful enough, it may even make it to the standard library in a later iteration of the language.
4. Fixability: Built-in mechanisms, such as library versioning, make it possible to address bugs without breaking existing code, which can still use an earlier version of the library.

The XL standard library consists of a wide variety of modules. The top-level module is called `XL`, and sub-modules are categorized in a hierarchy. For example, if you need to perform computations on complex numbers, you would `use XL.MATH.COMPLEX` to load the complex numbers module

The library builtins is a list of definitions that are accessible to any XL program without any explicit `use` statement. This includes most features that you find in languages such as C, for example integer arithmetic or loops. Compiler options make it possible to load another file instead, or even to load no file at all, in which case you need to build everything from scratch.

1.3.4. The case of text input / output operations

Input/output operations (often abbreviated as I/O) are a fundamental brick in most programming languages. In general, I/O operations are somewhat complex. If you are curious, the source code for the venerable `printf` function in C is available online.

The implementation of text I/O in XL is comparatively very simple. The definition of `print` looks something like, where irrelevant implementation details were elided as ⋯:

```
write X:text            as mayfail      is ... ①
write X:integer         as mayfail      is ...
write X:real            as mayfail      is ...
write X:character       as mayfail      is ...
write [[true]]          as mayfail      is { write "true"  } ②
write [[false]]         as mayfail      is { write "false" }
write Head, Rest        as mayfail      is { write Head; write Rest }

print                   as mayfail      is { write SOME_NEWLINE_CHARACTER }
print Items             as mayfail      is { write Items; print }
```

① The mayfail type is used to represent the `nil or error` type, in other words it indicates that the function either returns nothing, or returns an error.

② The `[[true]]` notation is called a metabox, and indicates that we must match the value of the expression in the metabox, in that case, `true`.

This is an example of *variadic function definition* in XL. In other words, `print` can take a variable number of arguments, much like `printf` in C. You can write multiple comma-separated items in a `print`. For example, consider the following code:

```
print "The value of X is ", X, " and the value of Y is ", Y
```

That would first call the last definition of `print` with the following binding for the variable `Items`:

```
Items   is "The value of X is ", X, " and the value of Y is ", Y
```

This in turn is passed to `write`, and the definition that matches is `write Head, Rest` with the following bindings:

```
Head    is "The value of X is "
Rest    is X, " and the value of Y is ", Y
```

In that case, `write Head` will directly match `write X:text` and write some text on the console. On the other hand, `write Rest` will need to iterate once more through the `write Head, Rest` definition, this time with the following bindings:

```
Head    is X
Rest    is " and the value of Y is ", Y
```

The call to `write Head` will then match one of the implementations of `write`, depending on the actual type of `X`. For example, if `X` is an integer, then it will match with `write X:integer`. Then the last split occurs for `write Rest` with the following bindings:

```
Head    is " and the value of Y is "
Rest    is Y
```

For that last iteration, `write Head` will use the `write X:text` definition, and `write Rest` will use whatever definition of `write` matches the type of `Y`.

All this can be done at compile-time. The generated code can then be reused whenever the combination of argument types is the same. For example, if `X` and `Y` are `integer` values, the generated code could be used for the following code:

```
print "The sum is ", X+Y, " and the difference is ", X-Y
```

This is because the sequence of types is the same. Everything happens as if the above mechanism had created a series of additional definition that looks like:

```
print A:text, B:integer, C:text, D:integer is
    write A, B, C, D
    print

write A:text, B:integer, C:text, D:integer is
    write A
    write B, C, D

write B:integer, C:text, D:integer is
    write B
    write C, D

write C:text, D:integer is
    write C
    write D
```

All these definitions are then available as shortcuts whenever the compiler evaluates future function calls.

The `print` function as defined above is both type-safe and extensible, unlike similar facilities found for example in the C programming language.

It is type-safe because the compiler knows the type of each argument at every step, and can check that there is a matching `write` function.

It is extensible, because additional definitions of `write` will be considered when evaluating `write Items`. For example, if you add a `complex` type similar to the one defined by the standard library, all you need for that type to become "writable" is to add a definition of `write` that looks like:

```
write Z:complex     is write "(", Z.Re, ";", Z.Im, ")"
```

Unlike the C++ `iostream` facility, the XL compiler will naturally emit less code. In particular, it will need only one function call for every call to `print`, calling the generated function for the given combination of arguments. That function will in turn call other generated functions, but the code sequence corresponding to a particular sequence of arguments will be factored out between all the call sites, minimizing code bloat.

Additionally, the approach used in XL makes it possible to offer specific features for output lines, for example to ensure that a single line is always printed contiguously even in a multi-threaded scenario. Assuming a `single_thread` facility ensuring that the code is executed by at most one thread, creating a locked `print` is nothing more than:

```
locked_print Items is
    single_thread
         print Items
```

It is extremely difficult, if not impossible, to achieve a similar effect with C++ `iostream` or, more generally, with I/O facilities that perform one call per I/O item. That's because there is no way for the compiler to identify where the "line breaks" are in your code.

1.4. Efficient translation

Despite being very high-level, XL was designed so that efficient translation to machine code was possible, if sometimes challenging. In other words, XL is designed to be able to work as a *system language*, in the same vein as C, Ada or Rust, i.e. a language that can be used to program operating systems, system libraries, compilers or other low-level applications.

For that reason, nothing in the semantics of XL mandates complex behind-the-scene activites, like garbage collection, thread safety, or even memory management. As for other aspects of the language, any such activity has to be provided by the library. You only pay for it if you actually use it. In other words, the only reason you'd ever get garbage collection in an XL program is if you explicitly need it for your own application.

This philosophy sometimes requires the XL compiler to work extra hard in order to be more than minimally efficient. Consider for example the definition of the `while` loop given above:

```
while Condition loop Body is
    if Condition then
        Body
        while Condition loop Body
```

That definition can be used in your own code as follows:

```
while N <> 1 loop
    if N mod 2 = 0 then N /= 2 else N := N * 3 + 1
```

What happens is that the compiler looks at the code, and matches against the definitions at its disposal. The `while` loop in the code matches the form `while Condition loop Body`, provided you do the following bindings:

```
Conditions is N <> 1
Body is
   if N mod 2 = 0 then N /= 2 else N := N * 3 + 1
```

The definition for the `while Condition loop Body` form is then evaluated with the above bindings, in other words, the code below then needs to be evaluated:

```
    if Condition then
        Body
        while Condition loop Body
```

Conceptually, that is extremely simple. Getting this to work well is of course a little bit complicated. In particular, the definition ends with another reference to `while`. If the compiler naively generates a *function call* to implement a form like that, executing that code would likely run out of stack space for loops with a large number of iterations. A special optimization called *tail call elimination* is required to ensure the expected behavior, namely the generation of a machine branch instruction instead of a machine call instruction.

Furthermore, the reference implementation is just that, a reference. The compiler is perfectly allowed, even encouraged, to "cheat", i.e. to recognize common idioms, and efficiently translate them. One name, `builtin`, is reserved for that purpose. For example, the definition of integer addition may look like this:

```
X:integer + Y:integer as integer    is builtin Add
```

The left part of `is` here is perfectly standard XL. It tells the compiler that an expression like `X+Y` where both `X` and `Y` have the `integer` type will result in an `integer` value (that is the meaning of `as integer`). The implementation, however, is not given. Instead, the `builtin Add` tells the compiler that it has a cheat sheet for that operations, called `Add`. How this cheat sheet is actually implemented is not specified, and depends on the compiler.

1.5. Adding complex features

Features can be added to the language that go beyond a simple notation. This can also be done in XL, although this may require a little bit of additional work. This topic cannot be covered extensively here. Instead, examples from existing implementations will provide hints of how this can happen.

1.5.1. Reactive programming in Tao3D

Reactive programming is a form of programming designed to facilitate the propagation of changes in a program. It is particularly useful to react to changes in a user interface.

Tao3D added reactive programming to XL to deal with user-interface events, like mouse movements or keyboard input. This is achieved in Tao3D using a combination of *partial re-evaluation* of programs in response to *events* sent by functions that depend on user-interface state.

For example, consider the following Tao3D program to draw the hands of a clock (see complete YouTube tutorial for more details):

```
locally
    rotate_z -6 * minutes
    rectangle 0, 100, 15, 250

locally
    rotate_z -30 * hours
    rectangle 0, 50, 15, 150

locally
    color "red"
    rotate_z -6 * seconds
    rectangle 0, 80, 10, 200
```

The `locally` function controls the scope of partial re-evaluation. Time-based functions like `minutes`, `hours` or `seconds` return the minutes, hours and seconds of the current time, respectively, but also trigger a time event each time they change. For example, the `hours` function will trigger a time event every hour.

The `locally` function controls partial re-evaluation of the code within it, and caches all drawing-related information within it in a structure called a *layout*. There is also a top-level layout for anything created outside of a `locally`.

The first time the program is evaluated, three layouts are created by the three `locally` calls, and populated with three rectangles (one of them colored in red), which were rotated along the Z axis (perpendicular to the screen) by an amount depending on time. When, say, the `seconds` value changes, a time event is sent by `seconds`, which is intercepted by the enclosing `locally`, which then re-evaluated its contents, and then sends a redraw event to the enclosing layout. The two other layouts will use the cached graphics, without re-evaluating the code under `locally`.

All this can be implemented entirely within the constraints of the normal XL evaluation rules. In other words, the language did not have to be changed in order to implement Tao3D.

1.5.2. Declarative programming in Tao3D

Tao3D also demonstrates how a single language can be used to define documents in a way that feels declarative like a declarative language, i.e. similar to HTML, but still offers the power of imperative programming like JavaScript, as well as style sheets reminiscent of CSS. In other words, Tao3D does with a single language, XL, what HTML5 does with three.

For example, an interactive slide in Tao3D would be written using code like this (note that Tao3D uses `import` instead of `use`):

```
import Slides

slide "The XL programming language",
    * "Extensible"
    * "Powerful"
    * "Simple"
```

This can easily be mis-interpreted as being a mere markup language, something similar to markdown, which is one reason why I sometimes refer to XL as an *XML without the M*.

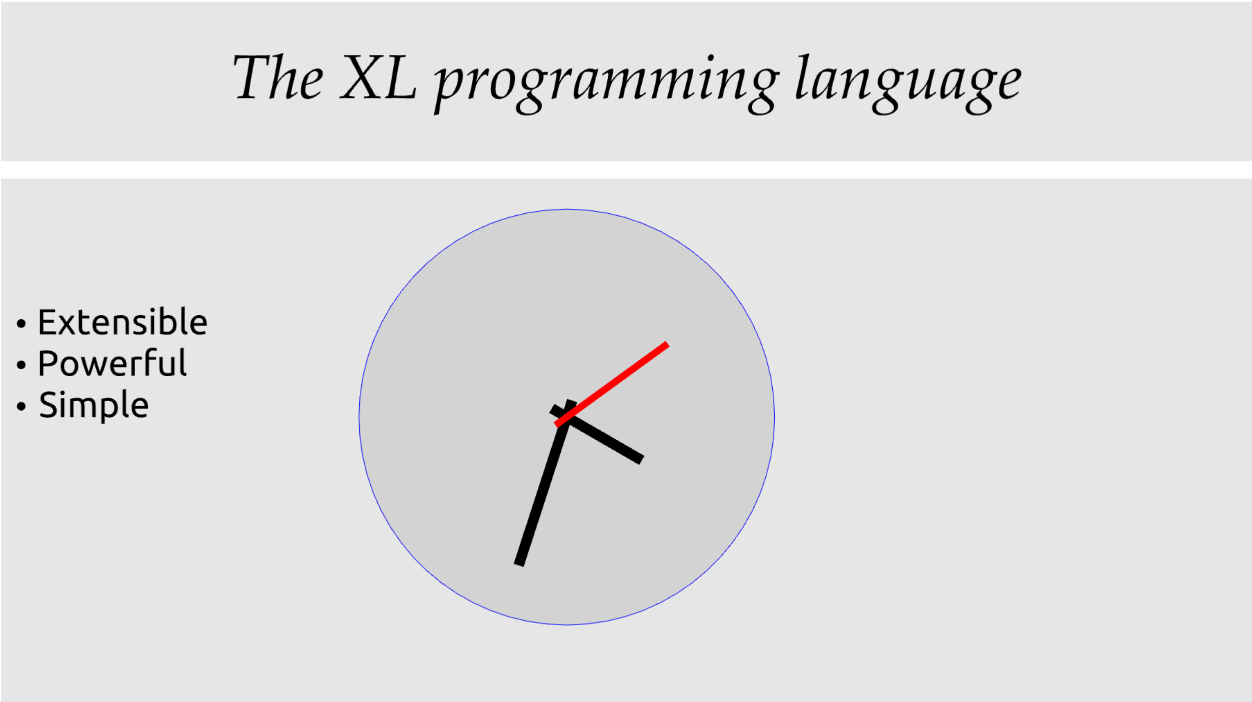
However, the true power of XL can more easily be shown by adding the clock defined previously, naming it `clock`, and then using it in the slide. This introduces the dynamic aspect that Javascript brings to HTML5.

```
import Slides

clock is
    locally
        line_color "blue"
        color "lightgray"
        circle 0, 0, 300

    locally
        rotate_z -6 * minutes
        rectangle 0, 100, 15, 250

    locally
        rotate_z -30 * hours
        rectangle 0, 50, 15, 150

    locally
        color "red"
        rotate_z -6 * seconds
        rectangle 0, 80, 10, 200

slide "The XL programming language",
    * "Extensible"
    * "Powerful"
    * "Simple"
    anchor
        translate_x 600
        clock
```

In order to illustrate how pattern matching provides a powerful method to define styles, one can add the following definition to the program in order to change the font for the titles (more specifically, to change the font for the "title" layouts of all themes and all slide masters):

```
theme_font Theme, Master, "title" is font "Palatino", 80, italic
```

The result of this program is an animated slide that looks like the following:



1.5.3. Distributed programming with ELFE

ELFE is another XL-based experiment targeting distributed programming, notably for the Internet of things. The idea was to use the homoiconic aspect of XL to evaluate parts of the program on different machines, by sending the relevant program fragments and the associated data over the wire for remote evaluation.

> ELFE is now integrated as part of XL, and the ELFE demos are stored in the demo directory of XL.

This was achieved by adding only four relatively simple XL functions:

- `tell` sends a program to another node in a "fire and forget" way, not expecting any response.
- `ask` evaluates a remote program that returns a value, and returns that value to the calling program.
- `invoke` evaluates a remote program, establishing a two-way communication with the remote that the remote can use with `reply`
- `reply` allows remote code within an `invoke` to evaluate code in its original caller's context, but with access to all the local variables declared by the remote.

Consider the following program:

```
WORKER_1 is "pi2.local"
WORKER_2 is "pi.local"

invoke WORKER_1,
   every 1.1s,
        rasp1_temp is
            ask WORKER_2,
                temperature
        send_temps rasp1_temp, temperature

   send_temps T1:real, T2:real is
       if abs(T1-T2) > 2.0 then
           reply
               show_temps T1, T2

show_temps T1:real, T2:real is
    print "Temperature on pi is ", T1, " and on pi2 ", T2, ". "
    if T1>T2 then
        print "Pi is hotter by ", T1-T2, " degrees"
    else
        print "Pi2 is hotter by ", T2-T1, " degrees"
```

This small program looks like a relatively simple control script. However, the way it runs is extremely interesting.

1. This single program actually runs on three different machines, the original controller, as well as two machines called `WORKER_1` and `WORKER_2`.
2. It still looks and feels like a single program. In particular, variables, values and function calls are passed around machines almost transparently. For example
   - the computation `T1-T2` in `send_temps` is performed on `WORKER_1`…
   - … using a value of `T1` that actually came from `WORKER_2` through the `ask` statement in `rasp1_temp`.
   - Whenever the `reply` code is executed, variable `T1` and `T2` live on `WORKER_1`…
   - … but within the `reply`, they are passed transparently as arguments in order to call `show_temps` on the controller.
3. Communication occurs primarily between `WORKER_1` and `WORKER_2`, which exchange a message every 1.1s. Communication with the controller only occurs if and when necessary. If the controller resides in Canada and the workers in Australia, this can save substantial networking costs.
4. A single `temperature` function, with an extremely simple implementation, provides an remarkably rich set of remotely-accessible features that might require a very complex API in other languages.

This last point is worth insisting on. The following program uses the same function to compute the minimum, maximum and average temperature on the remote node. Nothing was changed to the temperature API. The computations are performed efficiently by the remote node.

```
invoke "pi.local",
    min   is 100.0
    max   is 0.0
    sum   is 0.0
    count is 0

    compute_stats T:real is
        min   := min(T, min)
        max   := max(T, max)
        sum   := sum + T
        count := count + 1
        reply
            report_stats count, T, min, max, sum/count

    every 2.5s,
        compute_stats temperature

report_stats Count, T, Min, Max, Avg is
    print "Sample ", Count, " T=", T, " ",
          "Min=", Min, " Max=", Max, " Avg=", Avg
```

The definitions of `min`, `max`, `sum` and `count` would not be acceptable in the version of XL described in this document. You would need to write for example `min : real := 100` instead of `min is 100.0`, since `min is 100.0` would declare a constant.

To run the ELFE demos, you need to start an XL server on the machines called `pi.local` and `pi2.local`, using the `-remote` command-line option of XL:

```
% xl -remote
```

You can then run the program on a third machine with:

```
% xl 7-two-hops.xl
```

Like for Tao3D, the implementation of these functions is not very complicated, and more importantly, it did not require any kind of change to the basic XL evaluation rules. In other words, adding something as sophisticated as transparently distributed progrmming to XL can be done by practically any programmer, without changing the compiler.

Chapter 2. XL syntax

For programmers familiar with other programming languages, the syntax of XL may not seem very innovative at first, and that is intentional. Most programmers should be able to read and write correct XL code in a matter of minutes.

The first noticable thing is a disturbing lack of all these nice semi-random punctuation characters that have decorated programs since the dawn of computing and make most source code look like an ornate form of line noise to the uninitiated. Where are all the parentheses gone? Why this horrible lack of curly braces? How can you make sense of a program without a semi-colon to terminate or separate statements?

In reality, the difference between XL syntax and earlier programming languages is much more than skin deep. The syntax of XL is actually one of its most unique characteristics. The design of the XL syntax is essential to understand both the philosophy and implementation of the whole language.

2.1. Homoiconic representation of programs

XL is a homoiconic language, meaning that all XL programs are data and conversely. This makes it particularly easy for programs to manipulate programs, an approach sometimes referred to as *metaprogramming*. Metaprogramming is the foundation upon which the touted extensibility of XL is built.

2.1.1. Why Lisp remains so strong to this day

In that respect, XL is very much inspired by one of the earliest and most enduring high-level programming languages, Lisp. The earliest implementations of Lisp date back to 1958, yet that language remains surprisingly modern and flourishing today, unlike languages of that same era like Cobol or Fortran.

One reason for Lisp's endurance is the metaprogramming capabilities deriving from homoiconicity. If you want to add a feature to Lisp, all you need is to write a program that translates Lisp programs with the new feature into previous-generation Lisp programs. This kind of capability made it much easier to add object-oriented programming to Lisp than to languages like C: neither C++ nor Objective C were implemented as just another C library, and there was a reason for that. Unlike Lisp, C is not extensible.

Despite its strengths, Lisp remains confined to specific markets, in large part because to most programmers, the language remains surprisingly alien to this day, even garnering such infamous nicknames as "*Lots of Insipid and Stupid Parentheses*". As seen from a concept programming point of view, the underlying problem is that the Lisp syntax departs from the usual notations as used by human beings. For example, adding 1 and 2 is written `1+2` in XL, like in most programming languages, but `(+ 1 2)` in Lisp. In concept programming, this notational problem is called *syntactic noise*.

XL addresses this problem by putting human usability first. In that sense, it can be seen as an effort to make the power of Lisp more accessible. That being said, XL is quite a bit more than just Lisp with a new fancy and programmer-friendly syntax.

2.1.2. The XL parse tree

The XL syntax is much *simpler* than that of languages such as C, and arguably not really more complicated than the syntax of Lisp. The parser for XL is less than 800 lines of straightforward C++ code, and the scanner barely adds another 900 lines. By contrast, the C parser in GCC needs more than 20000 lines of code, which is about the size of a complete XL interpreter, and the C++ parser is over twice as much!

A key to keeping things really simple is that the XL syntax is *dynamic*. Available operators and their precedence are *configured* primarily through a syntax file. As a result, there are no hard-coded keywords or special operators in the XL compiler.

All XL programs can be represented with a very simple tree structure, called a *parse tree*. The XL parse tree contains *leaf nodes* that don't have any children, such as integer, real, text or symbol nodes, and *inner nodes* that have at least one child node, such as infix, prefix, postfix and block nodes. In general, when a node can have children, these children can be of any kind.

Leaf nodes contain values that are atomic as far as XL is concerned:

1. `integer` nodes represent non-negative whole numbers like `1234`, `2#1001` or `16#FFFE_FFFF`.
2. `real` nodes represent a floating-point approximation of real numbers like `1.234`, `1.5e-10` or `2#1.0001_0001#e24`.
3. `character` nodes represent individual characters, like `'A'`.
4. `text` nodes represent text values like `"Hello world"`
5. `name` nodes represent names like `JOHN_DOE`
6. `operator` nodes represent non-alphabetical operators like `<=>`.
7. `symbols` nodes regroup names, symbols and a special empty symbol used in the representation of empty blocks like `()`.
8. `data` nodes hold an arbitrary amount of binary data.

Inner nodes contains combinations of other XL nodes:

1. `infix` nodes represent two operands separated by a name or operator, like `A+B` or `X and Y`. Infix nodes with a "new line" name are used for separate program lines.
2. `prefix` nodes represent two nodes where the operand follows the operator, like `+A` or `sin X`.
3. `postfix` nodes represent two nodes where the operator follows the operand, like `3%` or `45km`.
4. `block` nodes represent a node surrounded by two delimiters, like `[a]`, `(a)`, `{a}`. Blocks are also used to represent indentation.
5. `parenthese_block` nodes are delimited with `(` and `)`.
6. `square_block` nodes are delimited with `[` and `]`.
7. `curly_block` nodes are delimited with `{` and `}`.
8. `indent_block` nodes are delimited by code indentation.

For example, let's consider the following code:

```
if X < 0 then
   print "The value of ", X, " is negative"
   X := -X
```

Assuming that this program is stored in a file called `program.xl`, the XL parse tree for this program can be obtained by using the following command:

```
% xl -parse program.xl -style debug -show
(infixthen
 (prefix
  if
  (infix<
   X
   0))
 (block indent
  (infix CR
   (prefix
    print
    (infix,
     "The value of "
     (infix,
      X
      " is negative"
     )))
   (infix:=
    X
    (prefix
     -
     X
    )))))
```

All of XL is built on this very simple data structure. Some choices, like having distinct `integer` and `real` node, were guided primarily by considerations beyond syntax, for example the need to be able to precisely define program evaluation or to represent distinct machine types.

The list of node types given above is what the current implementations of XL offer. Some changes may happen in the future, notably:

- Adding a "binary object" node type, which could be used to store binary data in the program. A possible syntax would be to prefix `bits` before a large integer value or file name:

```
bits 16#FF_00_FF_00_FF_FF_00_FF_00
bits "image.png"
```

- Finding a better representation for empty blocks such as `()`. In the current implementation, they are represented as a block with an "empty symbol" as a child. With this choice, the parse tree has no "null" node anywhere in the tree. However, this is not very satisfactory, since the empty symbol cannot exist anywhere else in the parse tree. Alternatives such as representing blocks as possibly empty sequences of items have proven even more complicated, since the representation of `[A,B,C]` becomes ambiguous (it could be a block containing three elements, or a block containing two elements, one of them being an infix, or any other combination), and proved more difficult to process in a generic way.
- Finding a more efficient representation for large sequences of items. Currently, they are represented by an unbalanced tree, i.e. a tree where one side is disproportionately larger than the other. So far, attempts at finding a better representation all had at least one severe drawback that precluded their use.

2.2. Leaf nodes

The leaf nodes in XL each have a uniquely identifable syntax. For example, simply by looking at the sequence of characters, we can tell that `42` is a whole number, `3.5` is a fractional number, `"ABC"` is a text value, `'a'` is a character value, `ABC` is a name, and `->` is an operator. This section describes the syntax for leaf nodes.

There is currently no provision in the compiler to add new kinds of leaf nodes. This is being considered, and would require a minimal addition to the syntax file. The primary implementation issue is that it would require the syntax of the syntax file to diverge from the XL syntax itself, since numbers or names in the syntax file have to be "hardcoded" somehow

2.2.1. Numbers

Numbers in XL begin with a digit, i.e. one of `0123456789`, possibly followed by other digits. For example, `0` and `42` are valid XL numbers. XL describes two kinds of numbers: *whole numbers*, which have no fractional part, and *fractional numbers*, which have a fractional part.

In the rest of the document, other terminologies, such as *integer* or *real* numbers may be applied for whole numbers and fractional numnbers respectively. This corresponds to numbers having been given a type for evaluation purpose. This is notably the case whenever a computer font is used, e.g. when we refer to `integer` or `real` values. Except as far as syntax is concerned, this document will very rarely talk about whole numbers or fractional numbers.

A single underscore `_` character can be used to separate digits, as in `1_000_000`, in order to increase readability. The following are not valid XL numbers: `_1` (leading underscore), `2_` (trailing underscore), `3__0` (two underscores). While this is not a requirement, it is considered good style to group digits in equal-sized chunks, for example `1_000_000` or `04_92_98_05_55`.

By default, numbers are written in base 10. Any other numerical base between 2 and 36 can be used, as well as base 64 using a special syntax. Based numbers can be written by following the base with the `#` sign. For example `8#76` is an octal representation of `62`. For bases between 11 and 36, letters `A` through `Z` or `a` through `z` represent digit values larger than 10, so that `A` is 10, `f` is 15, `Z` is 35. Case does not matter. For example, `16#FF` and `16#ff` are two valid hexadecimal representation of `255`. For base 64, Base64 encoding is used, and case matters. This is mostly indended for use in binary objects, i.e. after `bits`. For instance, `64#SGVsbG8h` is the base-64 encoding for the number with the same binary representation as the sequence of ASCII characters in `Hello!`.

For fractional numbers, a dot `.` is used as decimal separator, and must separate digits. For example, `0.2` and `2.0` are valid but, unlike in C, `.2` and `2.` are not numbers but a prefix and postifix `.` respectively. This is necessary to avoid ambiguities. Also, the standard library denotes ranges using an infix `..`, so `2..3` is an infix `..` with `2` and `3` as operands, representing the range between 2 and 3.

Numbers can contain an exponent, specified by the letter `e` or `E`. If the exponent is negative, then the number is parsed as a fractional number. Therefore, `1e3` is integer value 1000, but `1e-3` is the same as `0.001`. The exponent is always given in base 10, and it indicates an exponentiation in the given base, so that `2#1e8` is 2^8, in other words decimal value 256. For based numbers, the exponent may be preceded by a `#` sign, which is mandatory if `e` or `E` are valid digits in the base, as in `16#FF#e2` which is an hexadecimal representation of decimal value 65280.

There is an implementation-dependent limit for the maximum value a number can have. This limit cannot be less than 2^{64}-1 for whole numbers, and less than `9.99e99` for floating-point numbers.

If a value is preceded by a `+` or `-` sign, that sign is parsed as a prefix operator and not as part of the number. For example, `-2` is a prefix `-` with `2` as an argument.

The various syntactic possibilities for XL numbers are only for convenience, and are all strictly equivalent as far as program execution is concerned. In other words, a program may not behave differently if a constant is spelled as `16#FF_FF` or as `65535`.

One unsatisfactory aspect of XL number syntax is that it does not offer an obvious path to correctly represent "semantic" version numbers in the code. For example, a notation like `2.3.1` will parse as an infix `.` between real number `2.3` and integer `1`, making it indistinguishable from `2.30.1`.

> Computers cannot really represent mathematical numbers. For example, the set of natural numbers is infinite, so there is no such thing as "the largest natural number". Due to hardware limitations, there is however such a thing as the largest 64-bit unsigned number. Similarly, there is no way to accurately represent real numbers in a computer, but there are at least two widely used representations called floating-point and fixed-point.
>
> From a concept programming point of view, this is a blatant case of concept cast. A computer `integer` is not a mathematical *integer*, and a computer `real` is only a floating-point or fixed-point approximation of a true *real number*. In the rest of this document, we will ignore this distinction, and refer to a `real`, knowing full well that there is a "largest" `real` value and a limited number of digits.

2.2.2. Symbols

Names in XL begin with an letter, followed by letters or digits. For example, `MyName` and `A22` are valid XL names. A single underscore `_` can be used to separate two valid characters in a name. Therefore, `A_2` is a valid XL name, but `A__2` and `_A` are not.

> The current implementation reads its input in Unicode UTF-8 format, and makes crude attempts at accepting Unicode. This was good enough for Tao3D to deal with multi-lingual text, including in languages such as Hebrew or Arabic. However, that implementation is a bit naive with respect to distinguishing Unicode letters from non-letter characters. For example, `▯_2` or `étalon` are valid XL names, and this is intentional, but `⇒A2` is presently a valid XL name, and this is considered a bug.

Case and delimiters are not significant in XL, so that `JOE_DALTON` and `JoeDalton` are treated identically.

> For historical reasons, the current implementations are quite lacking in that respect, and will treat `V` and `v` differently. There is still an open debate about giving a semantic role to capitalization.

Operators begin with one of the ASCII punctuation characters:

```
! # $ % & ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~
```

Operators longer than one character must be specified in the XL syntax file. For example, the XL syntax file defines a `<=` operator, but no `<=>` operator. Consequently, the sequence `1 <=> 2` will be parsed as `(1 <= (> 2))`. In order to add this operator, it is necessary to extend the syntax using a `syntax` statement.

Names and operators are treated interchangeably by XL after the parsing phase, and are collectively called *symbols*.

2.2.3. Text

Text in XL is delimited with a pair of single or double quotes. Text can contain any printable character. For example, `"Hello World"` or `'ABC'` are valid text in XL. If the delimiter is needed in the text, it can be obtained by doubling it. For example, `"He said ""Hello"""` is text containing `He said "Hello"`.

Additionally, the XL syntax file can specify delimiters for "long" text. Long text can include line-terminating characters, and only terminates when the matching delimiter is reached. By default, `<<` and `>>` are long-text delimiters, so that the following is valid text:

```
MyLongText is <<
   This is a multi-line text
   that contains several lines
>>
```

Additional delimiters can be configured, and can be used to define specific types of text. For example, a program that often has to manipulate HTML data could allow `HTML` and `END_HTML` as delimiters, so that you could write:

```
MyHTML is HTML
    <p>This is some HTML text here</p>
END_HTML
```

RATIONALE The reason for a built-in format for text using single or double quotes is because the syntax file is read using the standard XL parser, and it needs text tokens in some specific cases that would otherwise parse incorrectly such as block or comment delimiters.

2.3. Inner nodes

The inner nodes are defined by the syntax file, which specifies their precedence and associativity.

2.3.1. Indentation and off-side rule

Indentation in XL is significant. XL follows the *off-side rule* to define program blocks. There is no need for keywords such as `begin` and `end`, nor for block delimiters such as `{` or `}`. However, `{` and `}` can be used as block delimiters when needed, for example to create a block on a single line. The code below shows two equivalent ways to write the same loop:

```
loop { Eat; Pray; Love }
loop
    Eat
    Pray
    Love
```

The two ways to write the loop above are not just functionally equivalent. They also share the same parse tree structure, the only difference being the operators being used. For example, `A;B` is an infix `;` with `A` on the left and `B` on the right, whereas individual lines are operands of an infix *new-line* operator. Similarly, `{A}` is a block containing `A`, and indentation is represented in the parse tree by a block delimited by *indent* and *outdent* invisible symbols.

The structure of the second loop from the previous listing can be shown by the XL compiler using the `-show` option, as illustrated below:

```
% xl -parse loop.xl -style debug -show
(prefix
 loop
 (block indent
  (infix CR
   Eat
   (infix CR
    Pray
    Love
   ))))
```

Indentation must use the same indentation character within a single file, either tab or space. In other words, either your whole file is indented with tabs, or it is indented with spaces, but it is a syntax error to mix both.

Indentation within a block must be consistent. For example, the following code will cause a syntax error because of the incorrect indentation of `Pray`:

```
loop
    Eat
   Pray
    Love
```

2.3.2. Operator precedence and associativity

The operators available for XL programmers are defined by the syntax file. The same rules apply for all symbols, i.e. for names or for operators. The table given in this file uses keywords such as `INFIX`, `PREFIX` and `POSTFIX` to indicate if an operator is an infix, a prefix, or a postfix respectively.

The table also gives operators a precedence. For example, the following segment in the `INFIX` portion of the table indicates that `*` and `/` have higher precedence than `+` and `-`, so that `X+Y*Z` will parse as `X+(Y*Z)`:

```
        21      -> is has
        310     + -
        320     * / mod rem
```

The precedence also indicates associativity for infix operators. Even precedences indicate left associativity, as for `+` and `*` above. This means that `X * Y * Z` parses as `(X * Y) * Z`. Conversely, right-associativity is indicated by an odd precedence, as is the case for `is`. This means that `X is Y is Z` parses as `X is (Y is Z)`.

Enforcing different precedences for left and right associativity guarantees that it's impossible for operators to have the same precedence, with some being left-associative and some being right-associative, which would cause parsing ambiguities.

The syntax file uses a few special names:

- `INFIX`, `PREFIX`, `POSTFIX` and `BLOCK` introduce sections that declare the operators of the respective types.
- `COMMENT` and `TEXT` specify delimiters for comments and long text respectively.
- `SYNTAX` introduces a child syntax. It is followed by the name of a syntax file, and then by an opening and closing symbol for that syntax.
- `BINARY` specifies the names that introduce binary data. The default syntax file uses `bits`. The syntax for binary data can take one of two forms: either a very large integer constant in big-endian format, as in `bits 16#000102030405060708090A0B0C0D0E0F`, or the name of a file, as in `bits "image.png"`.
- `NEWLINE` is used to represent the infix operators that separates individual source code lines.
- `STATEMENT` is the precedence that delimits expressions from statements. Any operator with a lower precedence belongs to a statement, like `if` or `loop`. Any operator with a higher precedence belongs to an expression, like `+` or `*`.
- `DEFAULT` is the default precedence for names and symbols. It is not very important in practice.
- `FUNCTION` is the precedence for names and symbols used as a prefix when they are not explicitly listed in the file. If you write `sin X` for example, the associated precedence will be that of `FUNCTION`.

2.3.3. Delimiters

Additional sections of the syntax file define delimiters for comment, block and text. Comment and text delimiters come in pairs.

The default syntax file specifies comments that follow the C/C++ convention, i.e. comments either start with `/*` and end with `*/` or start with `//` and end with a new line. The basic text separators (simple and double quotes) are not specified in the syntax file because they are used to parse the syntax file itself. The default syntax file adds `<<` and `>>` as separators for multi-line text..

Block separators come in pairs and have a priority. The special names `INDENT` and `UNINDENT` are used for the indentation block. The block priority is used to give the priority of the block in an expression, but also to determine if the block contains an expression or a statement.

In the default syntax file, indentation blocks and blocks delimited by curly braces `{ }` contain statements, whereas blocks delimited by parentheses `( )` or square brackets `[ ]` will contain expressions.

2.3.4. Child syntax

A syntax file can define a child syntax file, which overrides the syntax when a given symbol is found.

The default syntax file contains a child syntax named `C` which is activated between the `extern` name and a following semi-colon `;`. This is used to approximate C-style parsing for extern declarations, making it easier to reference C code from XL:

```
extern real sqrt(real);
```

The so-called "C syntax" in XL is only a very crude and limited approximation of the actual C syntax, which is only intended for relatively simple function declarations.

2.3.5. Extending the syntax

The `syntax` name followed by a block can be used to alter the default syntax provided by the syntax file. Within the block, operators can be defined and their precedence given using the same rules as in the syntax file.

For example, if you want to add the spaceship operator `<=>` in your program, and give the same precedence as `<=`, namely 290, you could write:

```
syntax
    INFIX 290 <=>
```

Extending the syntax is intended to also work also in a module. This means that an `use` statement can alter the syntax in your source code. This is, however, rarely recommended. Also, importing a syntax extension does not presently work.

2.4. Making the syntax easy for humans

XL contains a couple of tweaks designed specifically to make code easier to read or write by humans. When the human logic is subtle, so is the XL compiler parsing…

2.4.1. Expression vs. statement

This first tweak is intended to put in XL an implicit grammatical grouping that humans apparently do. Consider for example the following:

```
print sin X, cos Y
```

Most human beings parse this as `print (sin(X),cos(Y))`, i.e. we call `print` with two values resulting from evaluating `sin X` and `cos Y`.

This is, however, not entirely logical. If `print` takes comma-separated arguments, why wouldn't `sin` also take comma-separated arguments? In other words, why doesn't this parse as `print(sin(X, cos(Y))`?

This shows that humans have a notion of *expressions* vs. *statements*. Expressions such as `sin X` have higher priority than commas and require parentheses if you want multiple arguments. By contrast, statements such as `print` have lower priority, and will take comma-separated argument lists. An indent or `{ }` block begins a statement, whereas parentheses `()` or square brackets `[]` begin an expression.

There are rare cases where the default rule will not achieve the desired objective, and you will need additional parentheses. One important such case is what follow `is` if it is not a block. Consider the following declarations:

```
debug X     is write "X=", X
expm1 X     is exp X - 1
double X    is X; X
```

The first example parses as intended, as a statement. The second one, however, is not, despite being syntactically similar. On could want to see this parse as `(exp X) -1`, but in reality, it parses as `exp (X-1)` for the same reason that the line above parses as `write ("X=", X)`. Another issue occurs with the body of `double X`, because it actually only contains the first `X`. The `;` operator has lower precedence than `is`, which is useful for maps, but does not achieve the expected effect above.

The solution to these problems is use a block on the right of `is` in all these cases. The correct way to write the above code is therefore:

```
debug X     is { write "X=", X } ①
expm1 X     is ( exp X - 1 )  ②
double X    is { X; X } ③
```

① The curly braces indicate that we expect `write` to be a statement.

② The parentheses indicate that we expect `exp` to be an expression.

③ The curly braces ensure that we interpret the sequence as the body of `double X`.

> A quality implementation of XL should probably warn if a prefix is seen on the right of `is` and has an infix as an argument. Expressions such as `type X` or `foo(A,B,C)` do not present a risk, but expressions such as `foo A-1` do represent present a risk, and should always be written in a block.

2.4.2. infix vs. prefix

Another special rule is that XL will use the presence of a space on only one side of an operator to disambiguate between an infix or a prefix. For example:

```
write -A    // write (-A)
B - A       // (B - A)
```

Chapter 3. XL program evaluation

XL defines *program execution* primarily in terms of operations on the parse tree combined with operations on an implicit *context* that stores the program state. The context itself is also described in XL in order to define the expected result of evaluation.

For efficiency, actual implementations are unlikely to store everything as an actual parse tree, although there is an *interpreter* implementation that does exactly that. A compiler is more likely to optimize representations of both code and data, as long as that optimized representation ultimately respect the semantics described using the normal form for the parse tree.

3.1. Execution phases

Executing an XL program is the result of three phases,

1. A parsing phase where program source text is converted to a parse tree,
2. A declaration phase, where all declarations are stored in the context,
3. An evaluation phase, where statements other than declarations are processed in order.

The execution phases are designed so that in a very large number of cases, it is at least conceptually possible to do both the parsing and declaration phases ahead of time, and to generate machine code that can perform the evaluation phase using only representations of code and data optimized for the specific machine running the program. It should be possible to create an efficient ahead-of-time compiler for XL. Work is currently in progress to build one.

> Reasonably efficient compilers were produced for earlier generations of the language, notably as part of the Tao3D project. However, this earlier iteration of the language had a very weak type system that made advanced optimizations hard to achieve. This was actually a feature for Tao3D, which purposely disabled some optimizations in order to improve compilation speed, notably when the program structure did not change. The version of XL described in this document, however, has markedly evolved relative to what was implemented in Tao3D, with the hope that much better code quality can be achieved. This part has not been demonstrated yet.

3.1.1. Execution context

The execution of XL programs is defined by describing the evolution of a particular data structure called the *execution context*, or simply *context*, which stores all values accessible to the program at any given time.

That data structure is only intended to explain the effect of evaluating the program. It is not intended to be a model of how things are actually implemented. As a matter of fact, care was taken in the design of XL to allow standard compilation and optimization techniques to remain applicable, and to leave a lot of freedom regarding actual evaluation techniques.

In the examples below, `CONTEXT0`, `CONTEXT1`, … will denote pseudo-variables that describe the various currently visible execution contexts, following the language scoping rules. The most recent contexts will have higher numbers. In addition, `HIDDEN0`, `HIDDEN1`, … will represent pending execution contexts that are invisible to the currently executing code. These are also known as *activation records*. Entries in `HIDDEN` contexts are live, but invisible to the current code. By convention, `CONTEXT0` and `HIDDEN0` are not defined in the examples and are assumed to be inherited from earlier execution.

3.1.2. Parsing phase

The parsing phase reads source text and turns it into a parse tree using operator spelling and precedence information given in the syntax file. This results either in a parse-time error, or in a faithful representation of the source code as a parse tree data structure that can be used for program evaluation.

Since there is almost a complete equivalence between the parse tree and the source code, the rest of the document will, for convenience, represent a parse tree using a source code form. In the rare cases where additional information is necessary for understanding, it will be provided in the form of XL comments.

Beyond the creation of the parse tree, very little actual processing happens during parsing. There are, however, a few tasks that can only be performed during parsing:

1. Filtering out comments: Comments should not have an effect on the program, so they are simply eliminated during parsing.
2. Processing `syntax` statements: This must be done during parsing, because `syntax` is designed to modify the spelling and precedence of operators, and that information is used during the parsing phase.
3. Processing `use` statements: Since imported modules can contain `syntax` statements, they must at least partially be processed during parsing. Details about `use` statements are covered in the chapter about modules.
4. Identifying words that switch to a child syntax: symbols that activate a child syntax are recognized during parsing. This is the case for example with the `extern` name in the default syntax.
5. Identifying binary data: words such as `bits` marked as introducing `BINARY` data in the syntax file are treated specially during parsing, to generate parse tree nodes representing binary data. > NOTE: this is not currently implemented.

The need to process `use` statements during parsing means that it's not possible in XL to have computed `use` statements. The name of the module must always be evaluated at compile-time.

RATIONALE An alternative would have been to allow computed `use` statement, but disallow `syntax` in them. However, for convenience, `use` names look like `XL.CONSOLE.TEXT_IO` and not, say, `"xl/console/text_io.xs"`, so there is no obvious way to compute them anyway. If computed `use` statement ever become necessary, it will be easy enough to use the syntax `use "path"` for them.

Once parsing completes successfully, the parse tree can be handed to the declaration and evaluation phases. Parsing occurs for the *entire program*, including imported modules, before the other phases begin.

3.1.3. Sequences

Both declaration and evaluation phases will process *sequences*, which are one of:

- A block, in which case processing the sequence means processing the block's child

```
loop { print "Hello World" }
```

- An infix `NEWLINE`, semi-colon `;` or comma `,`, in which case the left and right operands of the infix are processed in that order. The comma is typically used in parameter lists and to separate expressions, whereas the semi-colon and new-line are used to separate statements. Processing the infix as a sequence only happens if pattern matching did not succeed with the infix form.

```
print "One"; print "Two"
print "Three"
```

- An `use` statement, which is the only statement that requires processing in all three executation phases.

```
use XL.MATH.COMPLEX
```

- A `syntax` definition, which only plays a role during parsing is ignored during the declaration and evaluation phases.

```
syntax { INFIX 290 <=> }
```

- An infix `is`, which is called a *definition*, an infix `:` or `as`, which are called *type annotations*, or an infix assignment operator `:=` with a `:` type annotation on the left, called a *variable initialization*. Definitions, type annotations and variable initializations are collectively called *declarations*, and are processed during the declaration phase.

```
pi is 3.1415                 // Definition of 'pi'
e as real is 2.71828         // Typed definition of 'e'
Count : integer              // Variable declaration of 'Count'
byte_size X as integer       // Function declaration of 'byte_size X'
Remaining : integer := 100   // Variable initialization of 'Remaining'
```

- Anything else, which is called a *statement* and is processed during the evaluation phase.

```
print "This is a statement"
```

For example, consider the following code:

```
pi is 3.14
circumference 5.3
circumference Radius:real is 2 * pi * Radius
```

The first and last line are representing a definition of `pi` and `circumference Radius:real` respectively. The second line is made of one statement that computes `circumference 5.3`. There are two definitions, one statement and no type annotation in this code.

Note that there is a type annotation for `Radius` in the definition on the last line, but that annotation is *local* to the definition, and consequently not part of the declarations in the top-level sequence.

In that specific case, that type annotation is a declaration of a *parameter* called `Radius`, which only accepts `real` values. Sometimes, such parameters are called *formal parameters*. A parameter will receive its value from an *argument* during the evaluation. For example the `Radius` parameter will be *bound* to argument `5.3` while evaluating the statement on the second line.

The *result* of a sequence is the value of its last statement. In our example, the result of executing the code will be the value computed by `circumference 5.3`.

3.1.4. Declaration phase

The declaration phase of the program begins as soon as the parsing phase finishes.

During the declaration phase, all declarations are stored in order in the context, so that they appear before any declaration that was already in the context. As a result, the new declarations may *shadow* existing declarations that match(declaration,shadowing.

In the example above, the declaration phase would result in a context that looks something like:

```
CONTEXT1 is
    pi is 3.14
    circumference Radius:real is 2 * pi * Radius
    CONTEXT0
    HIDDEN0
```

An actual implementation is likely to store declarations is a more efficient manner. For example, an interpreter might use some hashing or some form of balanced tree. Such optimizations must preserve the order of declarations, since correct behavior during the evaluation phase depends on it.

In the case of a compiled implementation, the compiler will most likely assign machine locations to each of the declarations. When the program runs, a constant like `pi` or the definition of `circumference` may end up being represented as a machine address, and a variable such as `Radius` may be represented as a "stack location", i.e. a preallocated offset from the current stack pointer, the corresponding memory location only containing the value, i.e. the right-hand side of `:=`. Most of the type analysis can be performed at compile time, meaning that most type information is unnecessary at program run time and can be eliminated from the compiled program.

Note that since the declaration phase occurs before the execution phase, all declarations in the program will be visible during the evaluation phase. In our example, it is possible to use `circumference` before it has been declared. Definitions may therefore refer to one another in a circular way. Some other languages such as C require "forward declarations" in such cases, XL does not.

The parse tree on the left of `is`, `as` or `:` is called the *pattern* of the declaration. The pattern will be checked against the *form* of parse trees to be evaluated. The right operand of `:` or `as` is the type of the type annotation. The parse tree on the right of `is` is called the *body* of the definition.

3.1.5. Evaluation phase

The evaluation phase processes each statement in the order they appear in the program. For each statement, the context is looked up for matching declarations in order. There is a match if the shape of the tree being evaluated matches the pattern of the declaration. Precise pattern matching rules will be detailed below. In our example, `circumference 5.3` will not match the declaration of `pi`, but it will match the declaration of `circumference Radius:real` since the value `5.3` is indeed a real number.

When a match happens, a new context is created with definitions that *bind* formal parameters to the corresponding argument. Such definitions are, unsurprisingly, called *bindings*. This new context is called a *local context* and will be used to evaluate the body of the definition. For example, the local context for `circumference Radius:real` would be:

```
CONTEXT2 is
    Radius:real := 5.3
    CONTEXT1
    HIDDEN1
HIDDEN1 is CONTEXT1
```

As a reminder, `Radius` is a *formal parameter*, or simply *parameter* that receives the *argument* 5.3 as a result of *binding*. The binding remains active for the duration of the evaluation of of the body of the definition. The binding, at least conceptually, contains the type annotation for the formal parameter, ensuring that all required type constraints are known and respected. For example, the context contains the `Redius:real` annotation, so that attempting `Radius := "Hello"` in the body of `circumference` would fail, because the type of `"Hello"` does not match the `real` type.

Bindings can be marked as mutable or constant. In this document, bindings made with `:=` are mutable, while binding made with `is` are constant. Since by default, an `X : T` annotation creates a mutable binding, the binding for `Radius` is made with `:=`.

Once the new context has been created, execution of the program continues with the body of the definition. In that case, that means evaluating expression `2 * pi * Radius` in the newly created local context.

After execution of the body completes, the result of that execution replaces the statement that matched the definition's pattern. In our example, `circumference 5.3` behaves like `2 * pi * Radius` in a context containing `Radius is 5.3`.

The process can then resume with the next statement if there is one. In our example, there isn't one, so the execution is complete.

3.2. Expression evaluation

Executing the body for the definition of `circumference Radius:real` involves the evaluation of expression `2 * pi * Radius`. This follows almost exactly the same process as for `circumference 5.3`, but in that case, that process needs to be repeated multiple times to complete the evaluation.

If we apply the evaluation process with `2 * pi * Radius`, assuming the declarations in the standard library, no declaration has a larger pattern like `X * Y * Z` that could match the whole expression. However, there is a definition for a multiplication between `real` numbers, with a pattern that looks like `X:real * Y:real as real`, as well as another for `integer` multiplication, with a pattern that looks like `X:integer * Y:integer`. There may be more, but we will ignore them for the rest of this discussion. The code below shows what the relevant declaration might look like (`…` indicates irrelevant code):

```
X:integer * Y:integer   as integer  is ...
X:real * Y:real         as real     is ...
```

The `*` operator is left-associative, so `2 * pi * Radius` parses as `(2 * pi) * Radius`. Therefore, we will be looking for a match with `X` corresponding to `2 * pi` and `Y` corresponding to `Radius`. However, that information alone is insufficient to determine if either sub-expression is `integer` or `real`. In order to be able to make that determination, immediate evaluation of the arguments is required. The evaluation process therefore repeats with sub-expression `2 * pi`, and like before, it is necessary to evaluate `pi`. This in turns gives the result `3.14` given the current context. That result replaces `pi`, so that we now must evaluate `2 * 3.14`.

The `2 * 3.14` tree does not match `X:real * Y:real` because `2` is an `integer` and not a `real`. It does not match `X:integer * Y:integer` either because `3.14` is a `real` and not an `integer`. However, the standard library provides a definition of an *implicit conversion* that looks something like this:

```
X:integer as real     is builtin IntegerToReal
```

This implicit conversion tells the compiler how to transform an `integer` value like `2` into a `real`. Implicit conversions are only considered if there is no exact match, and only one of them can be used to match a given parameter. In our case, there isn't an exact match, so the evaluation will consider the implicit conversion to get a `real` from `integer` value `2`.

The body of the implicit conversion above is therefore evaluated in a context where `X` is set to `2`:

```
CONTEXT3 is
    X:integer := 2
    CONTEXT2
    HIDDEN2
HIDDEN2 is CONTEXT2
```

The result of that implicit conversion is `2.0`. Evaluation can then resume with the `X:real * Y:real as real` definition, this time called with an argument of the correct `real` type for `X`:

```
CONTEXT4 is
    X:real := 2.0
    Y:real := 3.14
    CONTEXT2
    HIDDEN2
```

The result of the multiplication is a `real` with value `6.28`, and after evaluating `Radius`, evaluation of the second multiplication will then happen with the following context:

```
CONTEXT5 is
    X:real := 6.28 // from 2 * pi
    Y:real :=5.3  // from Radius
    CONTEXT2
    HIDDEN2
```

The result of the last multiplication is a `real` with value `33.284`. This is the result of evaluating `circumference 5.3`, and consequently the result of executing the entire program.

The standard XL library only provides implicit conversions that do not cause data loss. On most implementation, `real` has a 53-bit mantissa, which means that the implicit conversion from `integer` to `real` actually needs to check the converted value in a platform-dependent way:

```
X:integer as real when X >= -2^53 and X < 2^53 is ...
```

3.3. Pattern matching

As we have seen above, the key to execution in XL is *pattern matching*, which is the process of finding the declarations patterns that match a given parse tree. Pattern matching is recursive, the *top-level pattern* matching only if all *sub-patterns* also match.

For example, consider the following declaration:

```
log X:real when X > 0.0 is ...
```

This will match an expression like `log 1.25` because:

1. `log 1.25` is a prefix with the name `log` on the left, just like the prefix in the pattern.
2. `1.25` matches the formal parameter `X` and has the expected `real` type, meaning that `1.25` matches the sub-pattern `X:real`.
3. The condition `X > 0.0` is true with binding `X is 1.25`

There are several kinds of patterns that will match different kinds of expressions:

- Name definitions match whole names.
- Wildcards match arbitrary arguments.
- Type annotations match arguments based on their type.
- Function (prefix) definitions match prefix forms ("functions").
- Postfix definitions match postfix forms.
- Infix definitions match infix forms.
- Argument splitting match names bound to infix, prefix or postfix values to infix, prefix or postfix patterns.
- Conditional patterns match values based on arbitrary conditions
- Literal constants match constants with the same value.
- Metabox values match values computed by the comiler.
- Blocks change the priority of expressions.
- Scope pattern matching allows large lists of paraameters to be passed as argument ina more readable way.

3.3.1. Name definitions

Top-level name patterns only match the exact same name.

Declaration	Matched by	Not matched by
`pi is 3.14`	`pi`	`ip`, `3.14`

Definitions with a top-level name pattern are called *name definitions*.

This case only applies to names, not to operators. You cannot define a `+` operator that way.

3.3.2. Wildcards

Name patterns that are not at the top-level can match any expression, and this does not require immediate evaluation. In that case, the expression will be bound to the name in the argument context, unless it is already bound in the current context. In that latter case, the value `New` of the new expression is compared with the already bound value `Old` by evaluating the `New=Old` expression, and the pattern only matches if that check evaluates to `true`.

Declaration	Matched by	Not matched by
`X+Y`	`2+"A"`	`2-3`, `+3`, `3+`
`N+N`	`3+3`, `A+B` when `A=B`	`3-3`, `3+4`

Such name patterns are called *wildcard parameters* because they can match any expression, or *untyped parameters* because no type checking occurs on the matched argument.

In order to catch anything at the top-level, for example in maps, it is necessary to use the `lambda` notation:

Declaration	Matched by	Not matched by
`\N`	Any value	Nothing

> This case only applies to names, not to operators. You cannot define a `+` parameter that way.

3.3.3. Type annotations

When the pattern is an infix `:` or `as`, it matches an expression if the expression matches the pattern on the left of the infix, and if the type of the expression matches the type on the right of the infix.

A type annotation as a top-level pattern is a declaration:

Top-level pattern	Matched by	Not matched by
`X:integer`	`X`	`2`, `'X'`
`seconds as integer`	`seconds`	`2`, `"seconds"`

A type annotation as a sub-pattern declares a parameter:

Parameter pattern	Matched by	Not matched by
`X:integer`	`42`	`X` (unless bound to an `integer`)
`seconds as integer`	`42`	`X` (unless constant bound to an `integer`)

Such patterns are called *type annotations*, and are used to perform type checking. Normally, type annotations using `:` are used to declare the type of parameters, whereas `as` is used to declare the type of the expression being defined, as shown for the pattern on the left of `is` in the example below:

```
X:real + Y:real as real is ...
```

For readability, a type annotation for a name can also be matched by an assignment or a name definition with the same name as the formal parameter:

```
circle (Radius:real, CenterX:real, CenterY:real) as circle
C : circle := circle(Radius := 3.5, CenterX := 6.5, CenterY := 3.3)

picture is type picture
    Width  : size
    Height : size
    Buffer : buffer
P : picture is picture
    Width  is 640
    Height is 480
    Buffer is my_buffer
```

The scope pattern matching makes it possible to give arguments in a different order in that case.

3.3.4. Function (prefix) definitions

When the pattern is a prefix, like `sin X`, the expression will match only if it is a prefix with the same name, and when the pattern on the right of the prefix matches the right operand of the expression.

Pattern	Matched by	Not matched by
`sin X`	`sin (2.27 + A)`	`cos 3.27`
`+X:real`	`+2.27`	`+"A"`, `-3.1`, `1+1`

When the prefix is a name, definitions for such patterns are called *function definitions*, and the corresponding expressions are usually called *function calls*. Otherwise, they are called *prefix definitions*.

3.3.5. Postfix definitions

When the pattern is a postfix, like `X%`, the expression will match only if it is a postfix with the same name, and when the pattern on the left of the postfix matches the left operand of the expression.

Pattern	Matched by	Not matched by
`X%`	`2.27%`, `"A"%`	`%3`, `3%2`
`X km`	`2.27 km`	`km 3`, `1 km 3`

Definitions for such patterns are called *postfix definitions*, and the corresponding expressions are usually called *postfix expressions*. The name or operator is sometimes called the *suffix*.

3.3.6. Infix definitions

When the pattern is an infix, it matches an infix expression with the same infix operator when both the left and right operands of the pattern match the corresponding left and right operands of the expression.

Pattern	Matched by	Not matched by
`X:real+Y:real`	`3.5+2.9`	`3+2`, `3.5-2.9`
`X and Y`	`N and 3`	`N or 3`

Definitions for such patterns are called *infix definitions*, and the corresponding expressions are called *infix expressions*.

3.3.7. Argument splitting

When the pattern is an infix, a prefix or a postfix, it also matches a name if that name is bound to an infix, prefix or postfix expression that would match. In that case, the bound value is said to be *split* to match the parameters.

Pattern	Matched by	Not matched by
`write X,Y`	`write Items` when `Items is "A","B"`	`write Items` when `Items is "A"+"B"`, `wrote 0,1`
`write X%`	`write Items` when `Items is 2%`	`write Items` when `Items is 2!`
`write -X`	`write Items` when `Items is -2`	`write Items` when `Items is +2`

A very common idiom is to use comma `,` infix to separate multiple parameters, as in the following definition:

```
write Head, Tail is write Head; write Tail
```

This declaration will match `write 1, 2, 3` with bindings `Head is 1` and `Tail is 2,3`. In the evaluation of the body with these bindings, `write Tail` will then match the same declaration again with `Tail` being split, resulting in bindings `Head is 2` and `Tail is 3`.

3.3.8. Conditional patterns

When a top-level pattern is an infix like `Pattern when Condition`, then the pattern matches an expression if the pattern on the left of the infix matches the expression, and if the expression on the right evaluates to `true` after bindings

Pattern	Matched by	Not matched by
`log X when X > 0`	`log 3.5`	`log(-3.5)`

Such patterns are called *conditional patterns*. They do not match if the expression evaluates to anything but `true`, notably if it evaluates to any kind of error. For example:

```
log X when X > 0 is ...
log "Logging an error"        // Will not match the definition above
```

3.3.9. Literal constants

When the pattern is an `integer` like `0`, a `real` like `3.5`, a `text` like `"ABC"`, it only matches an expression with the same value, as verified by evaluating the `Pattern = Value` expression, where `Pattern` is the literal constant in the pattern, and `Value` is the evaluated value of the expression. Checking that the value matches will therefore require immediate evaluation.

Pattern	Matched by	Not matched by
`0!`	`N!` when `N=0`	`N!` when `N<>0`

This case applies to sub-patterns, as was the case for `0! is 1` in the definition of factorial. It also applies to top-level patterns, which is primarily useful in maps:

```
digits is
    0 is "Zero"
    1 is "One"
```

3.3.10. Metabox values

When the pattern is a an expression between two square brackets, like `[[true]]`, it is called a *metabox*, and it only matches a value that is equal to the value computed by the metabox. This equality is checked by evaluating `Pattern = Value`, where `Pattern` is the expression in the metabox, and `Value` is the expression being tested.

Pattern	Matched by	Not matched by
`[[true]]`	`true`, `not false`	`"true"`, `1`

A metabox is used in particular when a name would be interpreted as a parameter. The two declarations below declare a short-circuit boolean `and` operator:

```
[[true]]  and X   is X
[[false]] and X   is false
```

By contrast, the two definitions would not work as intended, since they would simply declare parameters called `true` and `false`, always causing the first one to be evaluated for any `A and B` expression:

```
true  and X       is X
false and X       is false
```

3.3.11. Blocks

When the pattern is a block, it matches what the block's child would match. In other words, blocks in patterns can be used to change the relative precedence of operators in a complex expression, but play otherwise no other role in pattern matching.

Definition	Matched by	Not matched by
`(X+Y)*(X-Y) is X^2-Y^2`	`[A+3]*[A-3]`	`(A+3)*(A-4)`

The delimiters of a block cannot be tested that way. In other words, a pattern with angle brackets can match parentheses or conversely. For example, `[A:integer]` will match `2` or `(2)` or `{2}`.

It is possible to test the delimiters of a block, but that requires a conditional pattern. For example the following code will check if its argument is delimited with parentheses:

```
has_parentheses B:block when B.opening = "(" and B.closing = ")"  is true
has_parentheses B:block                                          is false
```

In some cases, checking if an argument matches a pattern requires evaluation of the corresponding expression or sub-expression. This is called immediate evaluation. Otherwise, evaluation will be lazy.

STYLE The rules of pattern matching give a lot of freedom with respect to coding style. Several conventions are recommended and are generally followed in this document:

- When a function takes multiple parameters, they are generally represented using a comma-separated parameter list, altough in some cases, other infix operators would do just as well:

  ```
  circle CenterX:real, CenterY:real, Radius:real is ...
  ```

- When there is such a comma-separated parameter list and when there is more than one formal parameter, it is customary to surround it with parentheses when the function is intended to be used in expressions, because in such an expression context, the parentheses are necessary at the call site. For example, if `circle` is intended to create a `circle` object rather than to draw a circle, the above definition might be written as follows:

  ```
  circle (CenterX:real, CenterY:real, Radius:real) as circle is ...
  C : circle := circle(0.3, 2.6, 4.0)
  ```

3.3.12. Scope pattern matching

When a block in a pattern defines a scope, i.e. a sequence of declarations or definitions, that scope is called a *parameter scope*, and it can be matched by any *argument scope* that provides matching definitions. In that case, the definitions in the argument scope may be provided in a different order, and the scope does not need to use the same delimiters or separators:

```
circle(Radius:real, CenterX:real, CenterY:real) as circle ①
C1 : circle := circle(3.5, 2.6, 3.2) ②
C2 : circle := circle(CenterX is 0.0; CenterY is 1.5; Radius is 2.4) ③
C3 : circle := circle ④
    Radius  is 1.5
    CenterX is 3.5
    CenterY is 2.4
```

① The formal parameters are a comma-separated sequence of declarations, meaning that they form a valid scope. A semi-colon or new-line could interchangeably be used there.

② This is the normal *positional form* for argument passing.

③ An argument scope is passed, which contains the necessary definitions to match the parameter scope. A semi-colon `;` must be used to separate the definitions, because the comma `,` has a higher precedence than `is`, and therefore cannot be used to separate `is` definitions without parentheses.

④ The argument scope need not use the same separators as the parameter scope. Using indentation and line separators removes the need for parentheses, since all kinds of blocks are equivalent.

This form is often used for data types containing a large number of parameters:

```
person is type person
    first_name : text
    middle_name: text
    last_name  : text
    birthdate  : date
    address    : address
JohnDoe : person := person
    last_name   is "Doe"
    first_name  is "John"
    middle_name is "W"
    birthdate   is date { Month is December; Day is 5; Year is 1968 }
    address     is address
        city    is "New-York"
        street  is "42nd"
        no      is 42
        zip     is 00002
```

3.3.13. Pattern-matching scope

When matching a pattern, a local execution context is created that holds the bindings associated to the patterns being matched. This *pattern-matching scope* is used while evaluating the body of the definition.

Consider the following simple example:

```
foo T:text, A:real is
    print "T=", T, " A=", A
foo "Hello", 2.5
```

As indicated earlier, the body associated to the `foo` pattern will evaluate with a pattern-matching scope that looks like:

```
CONTEXT1 is
    T : text is "Hello"
    A : real is 2.5
```

This is particularly useful for structured data values and user-defined data types. In XL, types are defined by the shape of a parse tree, and that shape is typically defined using a pattern. The scoping operator can then be used on values of the type to access the pattern scope.

For example, a `complex` data type and the addition of `complex` numbers can be written as follows:

```
complex is type complex(Re:real, Im:real) ①
Z1:complex + Z2.complex as complex is complex(Z1.Re+Z2.Re, Z1.Im+Z2.Im) ②
Z:complex := complex(1.3, 4.5) + complex(6.3, 2.5) ③
```

① This is a type definition based on a pattern. It indicates that the `complex` data type corresponds to all the values that have the parse-tree shape following `type`.

② The `Z1.Re` notation is a scoping operator, and evaluates `Re` in the pattern-matching scope of `Z1`.

③ Two constructors create two `complex` values, that are bound to `Z1` and `Z2` respectively. In the expression `Z1.Re`, the name `Re` is looked up in pattern-matching scope for these constructors, so that `Z1.Re` is `1.3` and `Z2.Im` is `2.5`.

3.4. Overloading

There may be multiple declarations where the pattern matches a given parse tree. This is called *overloading*. For example, as we have seen above, for the multiplication expression `X*Y` we have at least `integer` and `real` candidates that look something like:

```
X:integer * Y:integer as integer        is ...
X:real    * Y:real    as real           is ...
```

The first declaration above would be used for an expression like `2+3` and the second one for an expression like `5.5*6.4`. It is important for the evaluation to be able to distinguish them, since they may result in very different machine-level operations.

In XL, the various declarations in the context are considered in order, and the first declaration that matches is selected. A candidate declaration matches if it matches the whole shape of the tree.

> Historically, the XL2 implementation does not select the first that matches, but the *largest and most specialized* match. This is a slightly more complicated implementation, but not by far, and it has some benefits, notably with respect to making the code more robust to reorganizations. For this reason, this remains an open option. However, it is likely to be more complicated with the more dynamic semantics of XL, notably for dynamic dispatch, where the runtime cost of finding the proper candidate might be a bit too high to be practical.

For example, `X+1` can match any of the declarations patterns below:

```
X:integer + Y:integer
X:integer + 1
X:integer + Y:integer when Y > 0
X + Y
Infix:infix
```

The same `X+1` expression will not match any of the following patterns:

```
foo X
+1
X * Y
```

Knowing which candidate matches may be possible statically, e.g. at compile-time, for example if the selection of the declaration can be done solely based on the type of the arguments and parameters. This would be the case if matching an `integer` argument against an `integer` parameter, since any value of that argument would match. In other cases, it may require run-time tests against the values in the declaration. This would be the case if matching an `integer` argument against `0`, or against `N:integer when N mod 2 = 0`.

For example, a definition of the Fibonacci sequence in XL is given below:

```
fib 0   is 0
fib 1   is 1
fib N   is (fib(N-1) + fib(N-2))
```

> Parentheses are required around the expressions statements in the last declaration in order to parse this as the addition of `fib(N-1)` and `fib(N-2)` and not as the `fib` of `(N-1)+fib(N-2)`.

When evaluating a sub-expression like `fib(N-1)`, three candidates for `fib` are available, and type information is not sufficient to eliminate any of them. The generated code will therefore have to evaluate `N-1`. Immediate evaluation is needed in order to compare the value against the candidates. If the value is `0`, the first definition will be selected. If the value is `1`, the second definition will be used. Otherwise, the third definition will be used.

A binding may contain a value that may itself need to be split in order to be tested against the formal parameters. This is used in the implementation of `print`:

```
print Items             is { write Items; print }
write Head, Rest        is { write Head; write Rest }
write Item:integer      is ...  // Implementation for integer
write Item:real         is ...  // implementation for real
```

In that case, finding the declaration matching `print "Hello", "World"` involves creating a binding like this:

```
CONTEXT1 is
    Items is "Hello", "World"
    CONTEXT0
```

When evaluating `write Items`, the various candidates for `write` include `write Head, Rest`, and this will be the one selected after splitting `Items`, causing the context to become:

```
CONTEXT2 is
    Head is "Hello"
    Rest is "World"
    CONTEXT0
    HIDDEN1 is CONTEXT1
```

3.5. Dynamic dispatch

As shown above, the declaration that is actually selected to evaluate a given parse tree may depend on the dynamic value of the arguments. In the Fibonacci example above, `fib(N-1)` may select any of the three declarations of `fib` depending on the actual value of `N`. This runtime selection of declarations based on the value of arguments is called *dynamic dispatch*.

In the case of `fib`, the selection of the correct definition is a function of an `integer` argument. This is not the only kind of test that can be made. In particular, dynamic dispatch based on the *type* of the argument is an important feature to support well-known techniques such as object-oriented programming.

Let's consider an archetypal example for object-oriented programming, the `shape` class, with derived classes such as `rectangle`, `circle`, `polygon`, and so on. Textbooks typically illustrate dynamic dispatch using a `Draw` method that features different implementations depending on the class. Dynamic dispatch selects the appropriate implementation based on the class of the `shape` object.

In XL, this can be written as follows:

```
draw R:rectangle    is ... // Implementation for rectangle
draw C:circle       is ... // Implementation for circle
draw P:polygon      is ... // Implementation for polygon
draw S:shape        is ... // Implementation for shape

draw Something      // Calls the right implementation based on type of Something
```

A single dynamic dispatch may require multiple tests on different arguments. For example, the `and` binary operator can be defined (somewhat inefficiently) as follows:

```
[[false]] and [[false]]     is false
[[false]] and [[true]]      is false
[[true]]  and [[false]]     is false
[[true]]  and [[true]]      is true
```

When applied to types, this capability is sometimes called *multi-methods* in the object-oriented world. This makes the XL version of dynamic dispatch somewhat harder to optimize, but has interesting use cases. Consider for example an operator that checks if two shapes intersect. In XL, this can be written as follows:

```
X:rectangle intersects Y:rectangle  as boolean  is ... // two rectangles
X:circle    intersects Y:circle     as boolean  is ... // two circles
X:circle    intersects Y:rectangle  as boolean  is ... // rectangle & circle
X:polygon   intersects Y:polygon    as boolean  is ... // two polygons
X:shape     intersects Y:shape      as boolean  is ... // general case

if shape1 intersects shape2 then    // selects the right combination
    print "The two shapes touch"
```

> Type-based dynamic dispatch is relatively similar to the notion of *virtual function* in C++, although the XL implementation is likely to be quite different. The C++ approach only allows dynamic dispatch along a single axis, based on the type of the object argument. C++ also features a special syntax, `shape.Draw()`, for calls with dynamic dispatch, which differs from the C-style syntax for function calls, `Draw(shape)`. The syntax alone makes the `intersects` example difficult to write in C++.

As another illustration of a complex dynamic dispatch not based on types, Tao3D uses theme functions that depend on the names of the slide theme, master and element, as in:

```
theme_font "Christmas", "main",        "title"   is font "Times"
theme_font "Christmas", SlideMaster,   "code"    is font "Menlo"
theme_font "Christmas", SlideMaster,   SlideItem is font "Palatino"
theme_font SlideTheme,  SlideMaster,   SlideItem is font "Arial"
```

As the example above illustrates, the XL approach to dynamic dispatch takes advantage of pattern matching to allow complex combinations of argument tests.

3.6. Immediate evaluation

In the `circumference` examples, matching `2 * pi * Radius` against the possible candidates for `X * Y` expressions required an evaluation of `2 * pi` in order to check whether it was a `real` or `integer` value. This is called *immediate evaluation* of arguments, and is required in XL for statements, but also in the following cases:

1. When the formal parameter being checked has a type annotation, like `Radius` in our example, and when the annotation type does not match the type associated to the argument parse tree. Immediate evaluation is required in such cases in order to check if the argument type is of the expected type after evaluation. Evaluation is *not* required if the argument and the declared type for the formal parameter match, as in the following example:

   ```
   write X:infix   is  write X.left, " ", X.name, " ", X.right
   write A+3
   ```

   In that case, since `A+3` is already an `infix`, it is possible to bind it to `X` directly without evaluating it. So we will evaluate the body with binding `X:infix is A+3`.

2. When the part of the pattern being checked is a constant or a metabox. For example, this is the case in the definition of the factorial below, where the expression `(N-1)` must be evaluated in order to check if it matches the value `0` in pattern `0!`:

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

   This is also the case for the condition in `if-then-else` statements, to check if that condition matches either `true` or `false`:

   ```
   if [[true]]  then TrueBody else FalseBody    is TrueBody
   if [[false]] then TrueBody else FalseBody    is FalseBody
   ```

3. When the same name is used more than once for a formal parameter, as in the following optimization:

   ```
   A - A    is 0
   ```

   Such a definition would require the evaluation of `X` and `2 * Y` in expression `X - 2 * Y` in order to check if they are equal.

4. When a conditional clause requires the evaluation of the corresponding binding, as in the following example:

```
syracuse N when N mod 2 = 0  is N/2
syracuse N when N mod 2 = 1  is N * 3 + 1
syracuse X+5 // Must evaluate "X+5" for the conditional clause
```

Evaluation of sub-expressions is performed in the order required to test pattern matching, and from left to right, depth first. Patterns are tested in the order of declarations. Computed values for sub-expressions are memoized, meaning that they are computed at most once in a given statement.

3.7. Lazy evaluation

In the cases where immediate evaluation is not required, an argument will be bound to a formal parameter in such a way that an evaluation of the formal argument in the body of the declaration will evaluate the original expression in the original context. This is called *lazy evaluation*. The original expression will be evaluated every time the parameter is evaluated.

To understand these rules, consider the canonical definition of `while` loops:

```
while Condition loop Body is
    if Condition then
        Body
        while Condition loop Body
```

Let's use that definition of `while` in a context where we test the Syracuse conjecture:

```
while N <> 1 loop
    if N mod 2 = 0 then
        N /= 2
    else
        N := N * 3 + 1
    print N
```

The definition of `while` given above only works because `Condition` and `Body` are evaluated multiple times. The context when evaluating the body of the definition is somewhat equivalent to the following:

```
CONTEXT1 is
    Condition is N <> 1
    Body is
        if N mod 2 = 0 then
            N /= 2
        else
            N := N * 3 + 1
        print N
    CONTEXT0
```

In the body of the while definition, Condition must be evaluated because it is tested against metabox [[true]] and [[false]] in the definition of if-then-else. In that same definition for while, Body must be evaluated because it is a statement.

The value of Body or Condition is not changed by them being evaluated. In our example, the Body and Condition passed in the recursive statement at the end of the while Condition loop Body are the same arguments that were passed to the original invokation. For the same reason, each test of N <> 1 in our example is with the latest value of N.

Lazy evaluation can also be used to implement "short circuit" boolean operators. The following code for the and operator will not evaluate Condition if its left operand is false, making this implementation of and more efficient than the one given earlier:

```
[[true]]  and Condition is Condition
[[false]] and Condition is false
```

3.8. Closures

The bindings given above for Condition and Body are somewhat simplistic. Consider what would happen if you wrote the following while loop:

```
Condition is N > 1
while Condition loop N -= 1
```

Evaluating this would lead to a "naive" binding that looks like this:

```
CONTEXT2 is
    Condition is Condition
    Body is N -= 1
    CONTEXT0
```

That would not work well, since evaluating Condition would require evaluating Condition, and indefinitely so. Something needs to be done to address this.

In reality, the bindings must look more like this:

```
CONTEXT2 is
    Condition is CONTEXT1 { Condition }
    Body is CONTEXT1 { N-= 1 }
    CONTEXT0
```

The notation CONTEXT1 { Condition } means that we evaluate Condition in context CONTEXT1. This one of the scoping operators, which is explained in more details below. A prefix with a context on the left and a block on the right is called a *closure*.

In the above example, we gave an arbitrary name to the closure, `CONTEXT1`, which is the same for both `Condition` and `Body`. This name is intended to underline that the *same* context is used to evaluate both. In particular, if `Body` contains a context-modifying operation like `N -= 1`, that will modify the same `N` in the same `CONTEXT1` that will later be used to evaluate `N > 1` while evaluating `Condition`.

A closure may be returned as a result of evaluation, in which case all or part of a context may need to be captured in the returned value, even after that context would otherwise normally be discarded.

For example, consider the following code defining an anonymous function:

```
adder N is { lambda X is X + N }
add3 is adder 3     // Creates a function that adds 3 to its input
add3 5              // Computes 8
```

When we evaluate `add3`, a binding `N is 3` is created in a new context that contains declaration `N is 3`. That context can simply be written as `{ N is 3 }`. A context with an additional binding for `M is "Hello"` could be written something like `{ N is 3; M is "Hello" }`.

The value returned by `adder N` is not simply `{ lambda X is X + N }`, but something like `{ N is 3 } { lambda X is X + N }`, i.e. a closure that captures the bindings necessary for evaluation of the body `X + N` at a later time.

This closure can correctly be evaluated even in a context where there is no longer any binding for `N`, like the global context after the finishing the evaluation of `add3`. This ensures that `add3 5` correctly evaluates as `8`, because the value `N is 3` is *captured* in the closure.

A closure looks like a prefix `CONTEXT EXPR`, where `CONTEXT` and `EXPR` are blocks, and where `CONTEXT` is a sequence of declarations. Evaluating such a closure is equivalent to evaluating `EXPR` in the current context with `CONTEXT` as a local context, i.e. with the declarations in `CONTEXT` possibly shadowing declarations in the current context.

In particular, if argument splitting is required to evaluate the expression, each of the split arguments shares the same context. Consider the `write` and `print` implementation, with the following declarations:

```
write Head, Tail        is { write Head; write Tail }
print Items             is { write Items; print }
```

When evaluating `{ X is 42 } { print "X=", X }`, `Items` will be bound with a closure that captures the `{ X is 42 }` context:

```
CONTEXT1 is
    Items is { X is 42 } { "X=", X }
```

In turn, this will lead to the evaluation of `write Items`, where `Items` is evaluated using the `{ X is 42`

} context. As a result, the bindings while evaluating write will be:

```
CONTEXT2 is
    Head is CONTEXT1 { "X=" }
    Tail is CONTEXT1 { X }
    CONTEXT1 is { X is 42 }
```

The whole processus ensures that, when write evaluates write Tail, it computes X in a context where the correct value of X is available, and write Tail will correctly write 42.

3.9. Memoization

A sub-expression will only be computed once irrespective of the number of overload candidates considered or of the number of tests performed on the value. Once a sub-expression has been computed, the computed value is always used for testing or binding that specific sub-expression, and only that sub-expression. This is called *memoization*.

For example, consider the following declarations:

```
X + 0               is Case1(X)
X + Y when Y > 25   is Case2(X, Y)
X + Y * Z           is Case3(X,Y,Z)
```

If you evaluate an expression like A + foo B, then foo B will be evaluated in order to test the first candidate, and the result will be compared against 0. The test Y > 25 will then be performed with the result of that evaluation, because the test concerns a sub-expression, foo B, which has already been evaluated.

On the other hand, if you evaluate A + B * foo C, then B * foo C will be evaluated to match against 0. Like previously, the evaluated result will also be used to test Y > 25. If that test fails, the third declaration remains a candidate, because having evaluated B * foo C does not preclude the consideration of different sub-expressions such as B and foo C. However, if the evaluation of B * foo C required the evaluation of foo C, then that evaluated version will be used as a binding for Z.

Another important effect of memoization is that it limits the number of evaluation of top-level constants. In other words, a single evaluation will not "chase constants". Consider the following example:

```
do_not_chase is
    0 is 1
    1 is 2
    2 is 3
do_not_chase 0          // Returns 1, not 3
```

The evaluation of sub-expression 0 happens only once, and therefore, 1 is not itself evaluated again for the same sub-expression. This is quite important to get sensible results for maps.

RATIONALE These rules are not just optimizations. They are necessary to preserve the semantics of the language during dynamic dispatch for expressions that are not constant. For example, consider a call like `fib(random(3..10))`, which evaluates the `fib` function with a random value between `3` and `10`. Every time `random` is evaluated, it returns a different, pseudo-random value. The rules above guarantee that the *same* value will be used when testing against `0`, `1` or as a binding with `N`. Witout these rules, it would be possible for the body of the general case to be called with a value that is `0` or `1`.

3.10. Self

In a definition body, `self` refers to the input parse tree. A special idiom is a definition where the body is `self`, called a *self definition*. Such definitions indicates that the item being defined needs no further evaluation. For example, `true` and `false` can be defined as:

```
true    is self
false   is self
```

This means that evaluating `true` will return `true`, and evaluating `false` will return `false`, without any further evaluation. Note that you cannot write for example `true is true`, as `true` in the body is a statement, which would require further evaluation, hence an infinite recursion.

It is possible to use `self` for data structures. For example, in order to ensure that comma-separated lists are not evaluated, you can write :

```
X, Y    is self
```

Note that the following values also evaluate as themselves:

1. `integer`, `real` or `text` constants, unless an explicit declaration in the current context matches.
2. Sequences of declarations, like `{ Zero is 0; One is 1 }`, in particular the contexts captured for closures.

3.11. Implicit result variable

Within the body of a definition, an implicit variable called `result` holds the value that will be given to the caller. For example, an iterative version of the factorial function can be written as follows:

```
factorial N:natural as natural is
    result := 1
    for I in 2..N loop
        result *= I
```

3.12. Returned value

The value returned by the body of a definition is, in order:

1. the value of a `return` statement if there is one. A `return` statement immediately stops evaluation.
2. the value of any statement that returns an `error`.
3. the last value assigned to the `result` variable
4. if `result` was never assigned to in the body, the value of the last statement evaluated in the body.

For example, in addition to the definition given in the previous section, a factorial can be written as follows using a `return` statement, although it is not quite idiomatic:

```
factorial_return N:natural as natural is
    if N = 0 then
        return 1
    return N * factorial_return(N-1)
```

An alternate form would use the last returned value:

```
factorial_last N:natural as natural is
    if N = 0 then
        1
    else
        N * factorial_last(N-1)
```

3.13. Nested declarations

A definition body may itself contain declarations, which are called *nested declarations*.

When the body is evaluated, a *local declaration phase* will run, followed by a *local evaluation phase*. The local declaration phase will add the local declarations at the beginning of a new context, which will be destroyed when the body evaluation terminates. The local declarations therefore shadow declarations from the enclosing context.

For example, a function that returns the number of vowels in some text can be written as follows:

```
count_vowels InputText is
    is_vowel C is
        Item in Head, Tail  is Item in Head or Item in Tail
        Item in RefItem     is Item = RefItem
        C in 'a', 'e', 'i', 'o', 'u', 'y', 'A', 'E', 'I', 'O', 'U', 'Y'

    Count : integer := 0
    for C in InputText loop
        if is_vowel C then
            Count += 1
    Count
count_vowels "Hello World" // Returns 3
```

This code example defines a local helper `is_vowel C` that checks if `C` is a vowel by comparing it against a list of vowels. That local helper is not visible to the outer scopes, in other words, to the rest of the program. You cannot use `is_vowel X` elsewhere in the program, since it is not present in the outer context. It is, however, visible while evaluating the body of `count_vowels T`.

Similarly, the local helper itself defines an even more local helper infix `in` in order ot evaluate the following expression:

```
C in 'a', 'e', 'i', 'o', 'u', 'y', 'A', 'E', 'I', 'O', 'U', 'Y'
```

While evaluating `count_vowels "Hello World"`, the context will look something like:

```
CONTEXT1 is
    is_vowel C is ...
    Count:integer := 0
    InputText is "Hello World"
    CONTEXT0
```

In turn, while evaluating `is_vowel Char`, the context will look something like:

```
CONTEXT2 is
    Item in Head, Tail is ...
    Item in RefItem is ...
    C is 'l'
    CONTEXT1
```

The context is sorted so that the innermost definitions are visible first, possibly shadowing outer declarations. Also, outer declarations are visible from the body of inner ones. In the example above, the body of `is_vowel Char` could validly refer to `Count` or to `InputText`.

> This example is designed for illustration purpose only. It is not idiomatic XL, since the standard library provides useful tools. A better way to write it would be:
>
> ```
> count_vowels InputText is count C in InputText where C in
> "aeiouyAEIOUY"
> ```

3.14. Scoping

A list of declarations, similar to the kind that is used in closures, is called a *map* and evaluates as itself. One of the primary uses for maps is *scoping*, in other words defining a common *scope* for the declarations that it contains. Since the declaration phase operates on entire blocks, all declarations within a scope are visible at the same time.

There are two primary operations that apply to a map:

1. *Applying* a map as a prefix to an operand, as we saw with closures, evaluates the operand in the context defined by overlaying the map definitions on top of the current context.
2. *Scoping* an expression within a map uses the infix `.` operator, where the expression on the right is evaluated in a context that consists *exclusively* of the declarations in the map on the left.

Evaluating a closure is a prime example of map application. The context is captured by the closure in a map, and the closure itself is a prefix that corresponds to the map application. Such an expression can also be created explicitly. For example, `{ X is 40; Y is 2 } { X + Y }` will evaluate as `42`, taking `X` and `Y` from the map, and taking the declaration used to evaluate `X + Y` from the current context.

Another common usage for maps is to store declarations where the patterns are constant values. For example, you can use a map called `digit_spelling` to convert a digit to its English spelling:

```
digit_spelling is
    0 is "zero"
    1 is "one"
    2 is "two"
    3 is "three"
    4 is "four"
    5 is "five"
    6 is "six"
    7 is "seven"
    8 is "eight"
    9 is "nine"
```

With this declaration, the expression `digit_spelling 3` evaluates to `"three"`. This kind of map application is called *indexing*. A suggested style choice is to make the intent more explicit using square brackets, i.e. `digit_spelling[4]`, as a nod to the syntax of programming languages such as C or C++.

When the index is an expression, for example `digit_spelling[A+3]` in a context where `A is 2`, we must evaluate `A+3` in the current context augmented with the declarations in `digit_spelling`. In other words, the relevant context for evaluation will look something like:

```
{ X:integer+Y:integer as integer is ... }
  { A is 2 }
    { 0 is "zero"; 1 is "one"; ... }
      [A+3]
```

The first candidate for evaluation has pattern `0`. This requires immediate evaluation of expression `A+3` to check if it matches the value. Naively, one might think that evaluating it requires matching once more against `0`, and that the evaluation would neve terminate. However, memoization of sub-expression `A+3` means that it can no longer be evaluated in the inner context.

It can still, however, be evaluted in the outer context. In that outer context, the pattern matches the `X:integer+Y:integer` pattern, from which it computes value `2+3`, and then returns `5` for comparison in the inner context, in order to compare it against `0`. Since `0=5` fails, it then considers the next candidate, but again because of memoization, there is no need to re-evaluate the value of sub-expression `A+3`. Instead, the computed value `5` will be compared successively against `1`, `2`, and so on, until it matches `5`. The returned value for the inner expression is therefore `"five"`.

A map is not restricted to constant patterns. For example, the following map performs a more complete spelling conversion for numbers below 1000 (the notation `\N` being a shortcut for `lambda N`):

```
number_spelling is
    \N when N<10    is digit_spelling[N]
    11              is "eleven"
    12              is "twelve"
    13              is "thirteen"
    14              is "fourteen"
    15              is "fifteen"
    16              is "sixteen"
    17              is "seventeen"
    18              is "eighteen"
    19              is "nineteen"
    20              is "twenty"
    30              is "thirty"
    40              is "forty"
    50              is "fifty"
    60              is "sixty"
    70              is "seventy"
    80              is "eighty"
    90              is "ninety"
    \N when N<100   is (number_spelling[N/10*10] & " " &
                        digit_spelling[N mod 10])
    \N when N<1000  is (digit_spelling[N/100] & " hundred and " &
                        digit_spelling[N mod 100])
```

Another common idiom is to use a named map to group related declarations. This is the basis for the XL module system. For example, consider the following declaration:

```
byte_magic_constants is
    num_bits    is 8
    min_value   is 0
    max_value   is 255
```

With that declaration, `byte_magic_constants.num_bits` evaluates to `8`. A declaration like this can of course be more than a simple name:

```
magic_constants(Bits) is
    num_bits    is Bits
    min_value   is 0
    max_value   is 2^Bits - 1
```

In that case, `magic_constants(4).max_values` will evaluate to `15`.

This is also exactly what happens when you `use` a module. For example, with `use IO = XL.CONSOLE.TEXT_IO`, a local name `IO` is created in the current context that contains the declarations in the module. As a result, `IO.write` will refer to the declaration in the module.

3.15. Super lookup

In a given context, `super` is a way to refer to the enclosing scope.

```
X is 42
foo X:integer is X + super.X     // super.X refers to X above
foo 3                            // Returns 45
```

3.16. Assignments and moves

The infix `:=` operator is used to perform *assignments* and returns the value being assigned. Variants such as `+=`, `-=`, `*=`, `/=` are equivalent to performing the corresponding operating and assigning the result.

```
X : integer := 0    // Initialize X to 0
X := 5              // Now X contains value 5
X += 7              // Now X contains value 12
```

The `:=` operator (and only that operator) is a *variable declaration* when its left operand is an infix `:`. This was discussed earlier, and corresponds to the first line in the example above. A variable declaration is *not* an assignment.

Seven combined operators are defined independently of the type as follows:

```
X += Y      is      X := X + Y
X -= Y      is      X := X - Y
X *= Y      is      X := X * Y
X /= Y      is      X := X / Y
X &= Y      is      X := X & Y
X |= Y      is      X := X | Y
X ^= Y      is      X := X ^ Y
```

XL offers two additional operators, the `:+` *copy* operator and the `:<` *move* operator (which is also sometimes *cut* operator because of its shape that evokes scissors). The `:+` operator guarantees that all data is being copied, and that the new object is an independent copy of the original (hence the `+` character in it). The `:<` operator may simply move ownership of the value if that is less expensive than copying it, and invalidates the right side of the operator, which may no longer be used.

Depending on the data type, `:=` may correspond to a copy or a move. The precise details of which operator is selected and the associated rationale are detailed in the next chapter. In all cases, the previous value that was held in the left operand is destroyed by the assignment.

The `:=` operator is used to transfer arguments to parameters. This means that passing an argument in XL, like in Rust, can make the argument invalid in the caller if it is moved rather than copied. There are, however, multiple ways to pass arguments. This is all discussed in more details in the next chapter.

RATIONALE For simple types such as arithmetic types, an assignment performs a copy, which is a relatively inexpensive memmory copy between fixed-size locations. For more complicated data types, such as `spreadsheet`, `graph` or `picture`, a copy involves copying possibly megabytes of data, or complex webs of interconnected objects, which can be very expensive, and often leaves an unused copy behind. For such data types, moving data is the frequently desirable operations, for example to pass objects around as arguments, and copying data is the less frequent case. In any case, the programmer remains in charge, always having the possibility to explicitly request a copy or a move.

3.17. Functions as values

Unlike in several functional languages, when you declare a "function", you do not automatically declare a named entity or value with the function's name.

For example, the first definition in the following code does not create any declaration for `my_function` in the context, which means that the last statement in that code will cause an error.

```
my_function X is X + 1
apply Function, Value is Function(Value)
apply my_function, 1        // Error: Nothing called 'my_function'
```

RATIONALE One reason for that choice is that overloading means a multiplicity of declarations often need to be considered for a single expression. Another reason is that declarations can have arbitrarily complex patterns. It is not obvious what name should be given to a declaration of a pattern like `A in B..C`: a "name" like `in..` does not even "work" syntactically.

It is not clear how such a name would be called as a function either, since some of the arguments may themselves contain arbitrary parse trees, as we have seen for the definition of `print`, where the single `Items` parameter may actually be a comma-separated list of arguments that will be split when calling `write Items` and matching it to `write Head, Tail`.

If you need to perform the operation above, it is however quite easy to create a map that performs the operation. That map may be given a name or be anonymous. The following code example shows two correct ways to write such an `apply` call for a factorial definition:

```
0!                      is 1
N!                      is N * (N-1)!
apply Function, Value   is Function(Value)

// Using an anonymous map to compute 3!
apply { \N is N! }, 3

// Using a named map to compute 5!
factorial   is { \N is N! }
apply factorial, 5
```

Passing definitions like this might be seen as related to what other languages call *anonymous functions*, or sometimes *lambda function* in reference to Church's lambda calculus. The way this works, however, is markedly different internally, and is detailed in the section on scoping above.

3.18. Error handling

Code that fails will generally report it by returning an `error` value. Error values have the `error` type. For example, consider the `sqrt` (square root) function. That function is only defined for positive values.

```
sqrt X:real as real     when X >= 0     is ...
print "Square root of 2 is ", sqrt 2        // OK
print "Square root of -1 is ", sqrt(-1)     // Error
```

This program will print something similar to the following

```
Square root of 2 is 1.41421356237
Square root of -1 is Error: No form matches sqrt(-1)
```

This message is not very informative. For that reason, it is customary to add specific error messages for well-identified conditions:

```
sqrt X:real as real     when X >= 0     is ...
sqrt X:real as error    when X <  0     is error "Square root of negative real ", X
```

In that case, the output will change to something like:

```
Square root of 2 is 1.41421356237
Square root of -1 is Error: Square root of negative real -1.0
```

There are multiple ways to handle errors:

- Taking error parameters lets you explicitly deal with errors, for example to show an error message.
- Fallible types deal with cases where you expect a value or an error.
- Try-Catch will let you special-case error conditions.
- Error statements automatically propagate errors without cluttering your code with error checking conditions.

3.18.1. Taking error parameters

The simplest way to handle errors is to have a variant of the function that takes an `error` as an argument. For example, you could extend your square root function as follows:

```
sqrt X:real as real     when X >= 0     is ...
sqrt X:real as error    when X <  0     is error "Square root of negative real ", X
sqrt E:error as error                   is error "Square root of error: ", E
```

Now if you attempt to take the square root of an error, you will get a different output:

```
print "Double error is ", sqrt(sqrt(-1))
Double error is Error: Square root of error: Square root of negative real -1.0
```

As the code above illustrates, `print` and `write` are examples of functions that take an `error` parameter. In that case, these functions will print the associated error message.

3.18.2. Fallible types

Another way to handle errors is to use `mayfail T` types, which hold either a `T` or an `error`. The `mayfail` type (without a type argument) is the same as `mayfail nil`, and is normally used for functions that are not expected to return a value, but can return an error.

`mayfail T` contains four accessible features:

- `value` is a `T` value, and can only be accessed when there was no error (otherwise, it returns… an `error`!)
- `error` is an `error` value that should only be accessed when there was an error. Otherwise, it returns `nil`.
- `good` is `true` if there was no error, and `bad` otherwise.
- `bad` is equivalent to `not good`.

The following code shows how to use a `mayfail real` type to return `0.0` for the `sqrt` of a negative value:

```
sanitized_sqrt X:real as real is
    R : mayfail real := sqrt X
    if R.bad then
        print "Got an error in sqrt: ", R.error
        R := 0.0
    return R.value
```

3.18.3. Try-Catch

A third way to handle errors is to use a `try Body catch Handler` form, which evaluates `Body`, and if `Body` returns an `error`, evaluates `Handler` instead. The error that was caught by `catch` is called `caught`.

With this construct, the `sanitized_sqrt` above can be written in a much shorter and more idiomatic way as follows:

```
sanitized_sqrt X:real as real is
    try
        sqrt X
    catch
        print "Got an error in sqrt: ", caught
        0.0
```

This may look like exception handling, and intentionally so. However, `error` values are not exceptions in that they don't automatically propagate across functions like C++ exceptions do. If an error happens at some level, you must deal with it at that level, if only to explicitly pass it along. This is done automatically in many cases, so that the end result may feel a little like exceptions, but conceptually, this is always an `error` value being returned, not an exception being thrown.

3.18.4. Error statements

If a statement, assignment or declaration returns an `error`, then as a special evaluation rule, that any `error` value is immediately returned by the enclosing function. It is a type error if the interface of the enclosing function does not allow an `error` return value.

For example, in C, it is frequent to have code that looks like:

```
Thing *read_thing_from_file(const char *filename)
{
    FILE *file = fopen(filename, "r");
    if (file == NULL)
        return NULL;
    Thing *thing = malloc(sizeof(Thing))
    if (thing == NULL)
    {
        fclose(file);
        return NULL;
    }
    thing->header = malloc(sizeof(ThingHeader));
    if (thing->header == NULL)
    {
        free(thing);
        fclose(file);
        return NULL;
    }
    size_t header_read = fread(&thing->header, 1, sizeof(ThingHeader), file);
    if (header_read != sizeof(ThingHeader))
    {
        free (thing->header);
        free (thing);
        fclose(file);
        return NULL;
    }
    if (thing->header.size < MIN_SIZE)
    {
        log_error("Header size is too small: %u", thing->header.size);
        free(thing->header);
        free(thing);
        fclose(file);
        return NULL;
    }
    // ... possibly more of the same
    fclose(file);
    return thing;
}
```

In XL, handling `error` values is implicit, so that code similar to the above can be written as follows:

```
read_thing_from_file FileName:text as mayfail own thing is
    F:file := file(FileName)            // May error out ①
    H:own thing_header := read(F)       // May error out (and close F) ②
    if H.size < MIN_SIZE then
        // Explicitly error out with custom message
        error "Header size %1 is too small", H.size ③
    T:own thing := thing(H)             // May error out, dispose H, close F ④
    // ... possibly more of the same
    T
```

① This may error out if you cannot open the file, for example because it does not exist. This would typically return a `file_error`.

② This may error out because of an I/O error, but also because of a storage error if there isn't enough heap space to allocate the `thing_header`.

③ This is a case where you explicit error out. Since `error` builds an `error` value, it also implicitly returns from the function.

④ This might error out if making a `thing` out of `H` fails, but also if a `storage_error` is raised trying to find some heap space for a `thing`.

The notation `own T` above is an owning type that dynamically allocates an object from the heap.

3.19. Interface and implementation

XL provides strong *encapsulation* by allowing a programmer to hide irrelevant details of an implementation. This is fundamental to provide a robust module system.

All values in XL expose an *interface*, which define *what* can be done with the value, and also have an *implementation* of their interface to tell the program *how* operations actually happen. The interface needs to be visible for the program to be correct, but various mechanisms may allow to hide the implementation.

For example, a variable `integer` value named `X` has the following interface:

```
X : integer
```

This is all that is really needed in order to recognize the validity and meaning of operations such as `X+X`, `2*X+1`, `X<0` or `X:=18`. The actual value of `X` does not matter. In other words, it is sufficient to have the interface above to use `X`, an implementation like the one shown below can be hidden to the users of `X`:

```
X : integer := 42
```

The same is true for functions. For example, a function checking if a value is even could expose the following interface:

```
is_odd N:integer as boolean
```

Based on this interface alone, I know that I can write code that checks if a value is even or odd:

```
for I in 1..100 loop
    if is_odd I then
        print I, " is odd"
    else
        print I, " is even"
```

It does not matter if `is_odd` is actually implemented as follows:

```
is_odd N:integer as boolean is N mod 2 <> 0
```

or maybe as folows using the bitwise `and` operator:

```
is_odd N:integer as boolean is N and 1 = 1
```

The declarations must specify the interface of the values being used, but they need not specify the implementation. A definitions of the value must be provided at some point that matches the declaration and specifies an implementation, but that definition may be in a different source file.

RATIONALE In languages such as C++, some members of a class can be made *private* or *protected*. This restricts their usage, but the compiler (and the programmer) still have knowledge of internal details of the implementation. This facilitates some low-level copmiler optimizations (most of which are obsolete or irrelevant today when link-time optimizations are widely available), but also results in a number of long-term maintenance issues. Exposing implementation details in the interface worsens the fragile base class problem, since some aspects of the implementation are public enough that they cannot be modified. In XL, the implementation can be truly hidden, and an implementation must be able to generate code that does not depend on the implementation when the situation requires it, for example if the implementation may be in a different shared library than the code using the interface.

Chapter 4. Types

XL types are a way to organize values by restricting which operations can be selected during evaluation. For example, knowing that `A` is a `real` allows expression `A+A` to match declaration pattern `X:real+Y:real`, but prevents it from matching pattern `X:integer+Y:integer`.

In XL, types are based on the *shape* of parse trees. A type identifies the tree patterns that belong to the type. The expression `type(Pattern)` returns the type for the given type declaration pattern. For example, the type for all additions where the first value is a `real` is `type(A:real+B)`.

This approach to typing means in particular that a same value can belong to *multiple* types. For example, the expression `2+3*5` belongs to `type(A+B*C)`, but also to `type(A:integer+B:integer)`, or to `infix`. Therefore, for XL, you shouldn't talk about *the* type of a value, but rather about *a* type. However, in the presence of a type annotation, it is customary to talk about *the type* to denote the single type indicated by the annotation. For example, for `X:integer`, we will ordinarily refer to the type of `X` as being `integer`, although the value of `X`, for example `2`, may also belong to other types such as `even_integer` or `positive_integer` or `type(2)`, a type that only contains the value `2`.

4.1. Type annotations

A type can be associated to a name or expression using a *type annotation*. For example, a type annotation such as `X:integer` indicates that the values that can be bound to the name `X` must belong to the `integer` type.

Two infix operators can be used for type annotations, `X:T` and `X as T`. Both are annotations indicating that `X` belongs to type `T`. Typical usage for these two kinds of annotations is illustrated below, indicating that the `<` operator between two `integer` values has the `boolean` type:

```
X:integer < Y:integer as boolean
```

The first difference between the two kinds of type annotations is parsing precedence. The infix `:` has precedence higher than most operators, whereas infix `as` has a very low precedence. In most declarations, an infix `:` is used to give a type to formal parameters, whereas an infix `as` is used to give a type to the whole expression. This is illustrated in the example above, where `X:integer` and `Y:integer` define the types of the two formal parameters `X` and `Y` in the pattern `X < Y`, and the `as boolean` part indicates that the result of an operation like `3 < 5` has the `boolean` type.

Another difference is mutability. If type `T` is not explicitly marked as `constant` or `variable`, `X:T` indicates that `X` is mutable, whereas `X as T` indicates that `X` is not mutable. For example, `seconds : integer` declares a *variable* named seconds, where you can store your own seconds values, whereas `seconds as integer` declares a *function* named seconds, possibly returning the number of seconds in the current time from some real-time clock.

4.2. Type definitions

In XL, types are values like any other value, which simply match the `type` type. In particular, types can be declared or defined like any other value.

The simplest case of *type definition* simply gives a new name to an existing type. The following code will create a type named `int` that is just another name for `integer`:

```
int is integer
```

In reality, the above is really equivalent to deriving `int` from `integer`. in other words, the definition above is equivalent to the following pattern-based definition:

```
int is type(base:integer)
```

One reason this is important is to maintain some guarantees during destruction, specifically make sure that destruction of the new type does not bypass the destruction of the original type.

More interesting types can be defined using the `type` function, which takes a pattern as an argument, and returns a type that matches that pattern. The expression `type(42)` returns a type that only matches the value `42`.

The `type` function can obviously be used to create much more interesting types. In order to create a `positive` type that only matches positive values, one only needs the following code:

```
positive is type(X when X > 0)
```

If you come from another language, it is important to realize that `positive` as defined above is a type that accepts both `integer` values such as `27` and `real` values such as `3.14`, since the expression `X > 0` is valid in both cases. As a matter of fact, it applies to any type where the expression `X > 0` is valid. Types like this are often used in type expressoins such as `integer and positive`. Using such types makes it possible to write *constrained generic code*, i.e. code that applies on a large class of cases, while being properly constrained.

A common usage is to use a named prefix to create types that are easier to identify. This kind of notation is called a constructor, and plays for XL the role that `struct` plays for C or `record` for Pascal. A `complex` data type can be created and used as shown in the following code:

```
complex is type complex(Re:real, Im:real)
Z:complex := complex(4.3, 2.1)
```

This definition of `complex` states that a value matches the type `complex` if and only if it matches the pattern `complex(Re:real, Im:real)`. In particular, it matches the value `complex(4.3, 2.1)`, which makes the assignment on the second line of code possible.

4.3. Type expressions

Type definitions are not restricted to names. XL offers extensive support for *type expressions*, i.e. expressions that return a type. For example, the `complex` type might take the `real` type as an argument, instead of assuming the standard `real` type:

```
complex[real:type] is type complex(Re:real, Im:real)
complex is complex[real]
Z:complex := complex(4.3, 2.1)
K:complex[real32] := complex(1.2, 3.4)
```

Type expressions can be used to create what is called a *generic type* in languages like Ada, or a *template* in languages like C++. However, as far as syntax is concerned, it is indistinguishable from a function taking a type argument and returning a type. This extends the range of capabilities for the feature, meaning that the implementation may sometime have fo fall-back to more dynamically typed ways to evaluate the code.

STYLE A stylistic convention is to use square brackets in type expressions and parentheses in regular expressions. Thus, this documentation will typically write `complex[real]` for the type expression, and `complex(1.2,3.4)` for the numerical expression, although it would be perfectly legal to write `complex real` and `complex[1.2,3.4]` respectively, since XL does not differentiate blocks except for precedence.

In practice, type expression are extremely frequent, notably being used to define a plethora of generic types using operator-like notations, like `pointer to T`.

Standard type expressions include:

- `T1 or T2` is a type that contains values belonging to `T1` or `T2`. It is similar to what other languages call union types, and is used in particular for error reporting through types like `real or error`.
- `T1 and T2` is a type that contains values belonging both to `T1` and `T2`. It is primarily used to constrain types in generic code, for example `integer and positive`.
- `not T` is a type that contains vlaues that do not belong to `T`. It can be used to exclude specific types from a definition.
- `mayfail T` is a shortcut for `T or error` and is used when a function may fail.
- `one_of Patterns` is a type that accept one of the following patterns. It can be used to implement enumerations, such as `one_of(RED,GREEN,BLUE)`, but also more complex variant types with more complex patterns, like for the definition of `color` below which describes various ways to describe a color:

```
component is real range 0.0..1.0 bits 16
angle     is real range 0.0..360.0 bits 16
color is one_of
    rgb   Red:component, Green:component, Blue:component
    rgba  Red:component, Green:component, Blue:component, Alpha:component
    hsv   Hue:angle, Saturation:component, Value:component
    hsva  Hue:angle, Saturation:component, Value:component, Alpha:component
    cymk  Cyan:component, Yellow:component, Magenta:component, Black:component
    named Name:text
    named Name:text, Alpha:component
```

- `any_of Patterns` is a type that accept any combination of the following patterns. It can be used to implement flags, like the representation of Unix-style permissions as `any_of(READ,WRITE,EXECUTE)`, but also more complex variant types that may combine multiple patterns, as in the `text_style` type defined below, where a text style can contain at most a family, a weight, a slant, a fill color and a line color:

```
text_style is any_of
    family F:font_family
    weigth W:font_weight
    slant S:font_slant
    fill_color C:color
    line_color C:color
```

Below is code for a `complex` type that uses some of these features, and is somewhat closer to the actual implementation in `XL.MATH.COMPLEX` than what we have shown so far:

```
complex[real:type]      is real or polar[real] or cartesian[real]
cartesian[real:type]    is type cartesian(Re:real, Im:real)
polar[real:type]        is type polar(Mod:real, Arg:real)

with
    real : type like number
    C1   : cartesian[real]
    C2   : cartesian[real]
    P1   : polar[real]
    P2   : polar[real]

C1 + C2         is cartesian(C1.Re + C2.Re, C1.Im + C2.Im)
C1 - C2         is cartesian(C1.Re - C2.Re, C1.Im - C2.Im)
P1 * P2         is polar(P1.Mod * P2.Mod, P1.Arg + P2.Arg)
```

Type expressions evaluate following the regular rules of evaluation for XL. This makes it possible to build types that would be impossible to build in many mainstream languages. For example, it is frequent, for example in networking, to have a `packet` that has a *header* followed by a *payload*, the payload having a size that depends on information in the header. In XL, you can describe a type like this as follows:

```
header  is type header (byte_count:size)
payload[byte_count:size] is array[byte_count] of byte
packet is type packet
    header  : header
    payload : payload[header.byte_count]
```

The XL type system provides very strong guarantees even for data types as complicated as this one. For example, the following code will fail type system checks:

```
resize (P:inout packet, S:size) is
    P.header.byte_count := S ①
```

① This is a type error on the `packet` type, because the existing `payload` field no longer has the correct size, therefore the result does not belong to the `packet` type, unless `S` is the existing size.

The correct way to write the code above will hightlight the need to possibly reallocate memory for the new packet, and deal with three distinct cases for resizing:

```
resize (P:inout packet, S:size) when S = P.header.byte_count is nil
resize (P:inout packet, S:size) when S < P.header.byte_count is
    new_header  : header := header(S)
    new_payload : payload(S) := P.payload[0..S-1]
    P := packet(new_header, new_payload)
resize (P:inout packet, S:size) when S > P.header.byte_count is
    new_header  : header := header(S)
    new_payload : payload(S) := array
        lambda I when I < P.byte_count   is P.payload[I]
        lambda I                         is byte(0)
    P := packet(new_header, new_payload)
```

In addition, a large number of standard types, including generic container types, can be used to quickly build useful data types.

4.4. Shared type annotations

In some cases, notably for modules, a number of very similar declarations will have to be written again and again. For example, consider that you are writing code for implementing complex arithmetic. This might look something like:

```
complex is type complex(Re:real, Im:real)
Z1:complex + Z2:complex as complex is ...
Z1:complex - Z2:complex as complex is ...
Z1:complex * Z2:complex as complex is ...
Z1:complex / Z2:complex as complex is ...
```

In this code, `Z1` and `Z2` are always `complex` values. It seems unnecessary to have to repeat the time over and over again. XL offers a feature, called *shared type annotations*, where a `with` prefix, followed by a block of declarations, can be used to give local type annotations that will be valid in the entire scope where they are being used. The above examples could then be written as:

```
complex is type complex(Re:real, Im:real)
with
    Z1 : complex
    Z2 : complex
Z1 + Z2 as complex is ...
Z1 - Z2 as complex is ...
Z1 * Z2 as complex is ...
Z1 / Z2 as complex is ...
```

A shared type annotation may contain more complicated type information. In particular, you can declare the type for expressions. For example, if you define a factorial expression, you might ensure that all variants of the definition have a consistent type as follows:

```
with
    N  :  natural
    N! as natural
0! is 1
N! is N * (N-1)!
```

The first declaration within the `with` block indicates that any variable named `N` will have type `natural`. This is in particular true for the declaration on the next line. In other words, the second line in the `with` block is equivalent to:

```
N:natural! as natural
```

The pattern `0!` matches `N:natural!`, and the same is true for the next declaration. Therefore, the two definitions for the factorial are equivalent to the code below:

```
0:natural! as natural is 1
N:natural! as natural is N * (N-1)!
```

A shared type annotation can take another form, `with Types in Body`, which makes it possible to restrict the type annotations to a specific subset of declarations. The declatations in `Body` really belong to the scope containing the `with Types in Body` form.

```
with
    Z1 : complex
    Z2 : complex
in
    Z1 + Z2 as complex is ...
    Z1 - Z2 as complex is ...
    Z1 * Z2 as complex is ...
    Z1 / Z2 as complex is ...
```

This is particularly useful to provide complex type parameters in generic declarations. The following example illustrates this syntax to declare a notation `find Item in List` where the `Item` must have the type of the elements of the `List`.

```
with
    T:type
    L:type list of T
in
    find Item:T in List:L as mayfail T
```

4.5. Standard types

The XL library provides a number of standard types representing fundamental data types common in most programming languages, as well as more advanced and more idiomatic data types, such as the types used as building blocks for a parse tree. This section will only give an quick overview of many of the available types, with the intent to list them more than to describe them. A more complete description of the available types will be given in a later section about the standard XL library.

4.5.1. Basic types

Some fundamental data types are available on all implementations, and do not require any `use` statement. These fundamental types are called *basic types*, and include the following:

- `type` is the type used for types...
- `nil` is a type that contains only the value `nil`. It is generally used to represent an absence of value.
- `integer` is an approximation of integer numbers with a limited range, typically between -2^{63} and 2^{63}-1, which are accessible as `integer.min` and `integer.max` respectively. That range cannot be less than -2^{31} and 2^{31}-1. Overflowing while performing operations on `integer` operations behaves like the underlying hardware of the target machine, typically wrapping values around on all modern hardware. `integer` is a type that matches literal values below `integer.min`, such as `12`, or the result of the prefix negation operator on literal values, such as `-3`.
- `natural` is an approximation of natural numbers, with a limited range, typically between 0 and

2^{64}-1. Like `integer`, it behaves like the underlying hardware in case of overflow. `unsigned` matches whole number literal values such as `0`, `16#FF` or `42`.

- `size` is a type similar to `natural`, but specifically intended to represent a size. In some cases, it may have a different range than `natural`.
- `count` is a type similar to `natural`, but specifically intended to represent a count. It should be at least as large as `size` and `natural`.
- `offset` is a type that plays for `integer` the role that `size` plays for `natural`, i.e. it is intended to indicate offsets for example while indexing an array.
- `byte` is the smallest unsigned type that is naturally represented on the machine. On most modern machine, it is an 8-bit value.
- `character` is a representation of the native character set on the target machine. On modern machines, it should generally follow the Unicode standard for encoding characters. The `character` type matches single-quote single-character literal text constants like `'A'`.
- `text` is a representation for sequences of characters. On modern machines, it should generally use a compact representation such as UTF-8, and have an interface that is compatible with the `string of character` type. The `text` type matches literal text constants that contain any number of characters, for example `"Hello World"`.
- `boolean` is a type containing two values, `true` or `false`, and intended to represent truth values, for example conditions in tests. Unlike languages like C, the `boolean` type is not a numerical type.

4.5.2. Sized data types

Basic data types are not very precisely sized, in order to leave the implementation free to pick up a size that is maximally efficient on the target machine. For example, `integer` should hold 32-bit values on a 32-bit machine, and 64-bit values on a 64-bit machine.

This may adversely affect portability, and for that reason, XL also offers *sized types*, with a precise number of bits specified in the name. The size is appended to the type name. For example, `i64` is an `integer` type that is guaranteed to be exactly 64-bit.

Such sized data types exist for the following base types:

- `i` for at least 8, 16, 32 and 64 bits, are signed integer values,
- `u` for at least 8, 16, 32 and 64 bits, are unsigned integer values,
- `real` for at least 32 and 64 bits,
- `character` for at least 8, 16 and 32 bits.

Additional sizes may be provided if they are native to the target machine. For example, some DSPs feature 24-bit operations, and compilers for such machines should provide types like `u24` to match.

The sized types are guaranteed to wrap around at the boundary for the given number of bits. For example, `u8` holds values between `0` and `255`, and will wrap around so that the next value after `255` is `0`, and the value preceding `0` is `255`.

When the standard sizes are not sufficient, it is easy to use integer subtypes to identify precise ranges of values, as in `integer range 1..5`, which only accepts values between `1` and `5`, or precise number of bits, such as `integer bits 24`, which wraps around like a 24-bit `integer` value.

4.5.3. Category types

Some types are intended primarily as an easy way to categorize values along generally useful boundaries, and are naturally called *category types*. Examples include:

- `anything` is the *most general type*, which accepts any value. It is typically used to create true generic types.
- `number`, a type that matches numerical data types such as `integer`, `real` or `complex`.
- `positive`, a type that accepts only positive values.
- `ordered`, a type that only matches values that can be compared using `<`. A variant, `totally_ordered`, ensures that the type is totally ordered. This is unfortunately not the case of common types such as `real`.
- `discrete`, a type that only matches discrete types, such as `integer` or `character`. Discrete types feature an `index` function that returns the index of the value in the type.
- `access`, a type that accepts only values used to access other types, such as pointers or references.

Category types are often used to implement *generic algorithms* and *generic types* without overly burdening the code. For example, the `vector` type represents a mathematical vector, and that requires a `number` type for the values in the vector. Similarly, the `sort` algorithm only works on `ordered` values.

4.5.4. Generic containers

In some cases, a general structure is shared by a number of data types. For example, all array types share an internal organization and provide similar features. XL features *generic types* to address this kind of need. Most often, generic types are declared with formal parameters, and are *instantiated* by supplying arguments for the required parameters.

This is particularly useful for *container types*, i.e. types that are primarily designed to store a possibly large number of values from some other type.

Container types include in particular the following:

- `array` store a fixed number of consecutive elements. They exist in multiple flavors:
  - Zero-based arrays such as `array[5] of integer`.
  - Range-indexed arrays, such as `array['A'..'Z'] of boolean`, which are indexed with a `discrete` range of values.
  - Multi-dimensional arrays such as `array['A'..'H', 1..8] of chess_piece` are simply a convenient shortcut for arrays of arrays.
- `string` store a variable number of consecutive elements. They also exist in multiple flavors:
  - Unbounded, zero-based strings, such as `string of integer`. The `text` type exposes a `string of character` interface.

    ◦ Bounded, zero-based strings, such as `string[1000] of integer`, which can hold up to `1000` values of the `integer` type.
    ◦ Bounded, range-indexed strings, such as `string[1..10] of real`, which can hold up to `10` values, indexed starting at `1`.
    ◦ Multi-dimesional strings, such as `string[25,80] of character`, which is a storage-efficient way to store possibly blank text screens.

- `list` to store a variable number of linked elements. Unlike arrays or strings, elements in lists are individually allocated in memory rather than as a large contiguous chunk. Lists exist in several flavors:
    ◦ Single-linked lists such as `list of integer`.
    ◦ Doubly-linked, double-ended queues, such as `queue of integer`.
    ◦ Xor-linked lists such as `xor_list of integer`.
- `stack` to expose `push` and `pop` operations, and the type is matched by several container type such as `string`, `list` or `queue`. In other words, you can treat a `list of T` as a `stack of T`.
- `map` to efficiently map source values to a stored value. For example, `map[text] of real` creates a map between `text` index values and `real` stored values:
- `set` to efficiently store a set of value, and make it easy to know if a value is in the set or not. A `set of character` holds an arbitrary number of `character` values.

4.5.5. True generic types

Often, an algorithm will apply to all variants of a generic type. For example, consider the operation that sums all the elements in an array. Its body can be written so as to not really depend on the type or number of elements.

In order to makea it easier to write such generic code, XL programmers can take advantage of a feature called *true generic type*, which is a way to define a type that will accept any variant of some underlying generic type. It is customary to begin the name of such types with `some_`, in order to indicate the intent. For example, for `array`, one could write the true generic type as follows:

```
some_array is type (array[index:array_index] of value:type)
```

This makes it possible to write true generic code that takes an `array` argument, as follows:

```
sum A:some_array as A.value is
    result := 0
    for I in A loop
        result += I
```

It is possible for the pattern of the true generic type to be somewhat more restrictive. For example, for a `complex` type, one might want to create a type that only accepts numbers for the `real` type, in other words something like:

```
some_complex is type complex[real:type like number]
```

A type constructed like this is called a *constrained generic type*.

RATIONALE In languages such as C++, the lack of this feature often leads to code that largely repeats the same `template` arguments for each individual declaration:

```
template <typename T>
T sum(const vector<T> &v)
{
    T s = 0;
    for (auto i : v)
        s += i;
    return s;
}
```

The recent standards for C++ have introduced the notion of concepts to address the need to constrain generic types.

4.5.6. Other generic types

Many generic types are not intended as containers. They include the following:

- `range of T` holds ranges of values of type `T`. For example, the type `range of integer` can hold a value such as `1..5`, which is the range between values `1` and `5` inclusive.
- `own T` holds a dynamically allocated value of type `T` that is owned by the `own` value, i.e. it is disposed of when the `own` value is destroyed.
- `ref T` holds a reference to a value of type `T`. Its lifetime must be less than the value being referenced.
- `in T` can be used to pass input arguments of type `T`, i.e. values that are created and owned by the caller.
- `out T` can be used for output arguments of type `T`, i.e. values that are created by the callee.
- `in_out T` can be used for input-output arguments of type `T`, i.e. values that can be modified by the callee but are created and owned by the caller. Due to the rules about name lookup in XL, it is also possible to spell this type as `inout T` or `io T`.
- `any T` can hold a value of type `T` or any derived type, preserving the original type information, so that dynamic dispatch will happen based on the actual type.
- `slice of T` holds a slice of contiguous containers such as `array` or `string`, and makes it possible to manipulate subsets of the container. This is also useful with `text`.
- `access T` matches any type that can be used to access values of type `T`, such as `own T`, `ref T` or `slice of T`.
- `attribute T` is an attribute with type `T`, i.e. a value that can be read or written to in a controlled fashion.

4.5.7. Mathematical types

Computers are often used to perform mathematical operations. XL features several mathematical types designed for that purpose:

- `number` represent all kinds of numbers and is intended to be used in generic code.
- `natural` represent natural numbers, i.e. non-negative whole numbers.
- `integer` represent integer numbers, i.e. signed whole numbers.
- `rational` represent rational numbers, i.e. a rato of two whole numbers. They can be used to perform accurate computations on ratios.
- `real` is the base floating-point type, but it also provides fixed-point subtypes.
- `range of T` is a type that represents a range of numbers, and features range arithmetic, which makes it possible to estimate the effect of rounding errors in complex calculations involving floating-point types.
- `complex[T]` is a generic representation for complex numbers using `T` as the representation for real numbers. Without an argument, `complex` denotes `complex[real]`. Values with the `complex` type have a `polar` and `cartesian` representation, and the compiler will select the representation based on usage.
- `quaternion[T]` is a generic representation for mathematical quaternions, and are particularly useful in the field of 3D graphics. Without an argument, `quaternion` is the same as `quaternion[real]`.
- `vector` is a generic representation of mathematical vectors, which takes a size and a number type, so that `vector[3] of real32` represents a 3-dimensional vector of `real32` values, and `vector[4]` is a 4-dimensional `real` vector. The `vector` type exposes an `array` interface, but also provides additional capabilities such as vector arithmetic. Operations on the `vector` type typically take advantage of SIMD or "multimedia" operations on the processor if available.
- `matrix` is a generic representation of mathematical matrices, with two underlying representations, `sparse` and `dense`. The `matrix` type exposes an interface for a two-dimensional `array` type. It features matrix algebra and matrix-specific operations, as well as operations combining `matrix` and `vector` for the same underlying numerical type. Operations on the `matrix` type are often highly parallelizable. The `matrix[4,3] of i32` will create a 4x3 matrix with `i32` as the underlying numerical type, whereas `matrix[2,2]` will create a 2x2 matrix of `real` values.

4.5.8. Parse tree types

The `XL.PARSER` module offers a number of types intended to represent or match elements in the parse tree:

- `tree` matches any parse tree.
- `integer` matches whole number literals such as `42`.
- `real` matches fractional number literals such as `3.14`.
- `text` matches text literals such as `"ABC"`.
- `character` matches character literals such as `'A'`.

- `name` matches names such as `A`.
- `operator` matches operators such as `+`.
- `symbol` matches either name or operators, as well as an empty symbol used in empty blocks, e.g. `()`.
- `infix` matches infix expressions such as `A+B`.
- `prefix` matches prefix expressions such as `+3`.
- `postfix` matches postfix expressions such as `4%`.
- `block` matches block expressions such as `(A)`.

4.5.9. Program-related types

A few other types are related to program evaluation, notably:

- `lifetime` represents the lifetime of values.
- `parser` represents an XL parser.
- `evaluator` represents an XL program evaluator.
- `task` represent a running task.

4.5.10. Other common types

A number of modules provide various generally useful, if more specialized types. Here are some examples:

- In module `XL.MEMORY`
  - `byte` is a type representing the smallest addressable unit of memory, typically 8-bit on most modern implementations.
  - `address` is a type representing addresses for the target machine, i.e. values that can be used to index individual `byte` elements in memory.
- In module `XL.FILE`
  - `file` is a representation for a file on the file system, allowing *operations such as reading or writing.
  - `name` is a representation for file names that can be converted to and from `text`.
  - `path` is a representation for hierarchical access paths made of individual `name` instances
  - `directory` is a representation for directories, allowing to get a of files.
  - `attributes` describe file-related attributes, such as creation and modification date or access rights.

There are many more, documented in the section about the standard library.

4.6. Type-related concepts

A number of essential concepts are related to the type system, and will be explained more in details below:

- the lifetime of a value is the amount of time during which the value exists in the program. Lifetime is, among other things, determined by scoping.
- creation and destruction defines how values of a given type are initialized and destroyed.
- errors are special types used to indicate failure.
- mutability is the ability for an entity to change value over its lifetime.
- compactness is the property of some types to have their values represented in the machine in a compact way, i.e. a fixed-size sequence of consecutive memory storage units (most generally bytes).
- ownership is a properties of some types to control the lifetime of the associated values or possibly some other resource such as a network connection. Non-owning types can be used to access values of an associated owning type.
- inheritance is the ability for a type to inherit all operations from another type, so that its values can safely be implicitly converted to values of that other type. Conversely, subtypes are types that add constraints to a type.
- the interface of a type is an optional scope that exposes *features* of the type, i.e. individually accessible values. The *implementation* of the type must provide all interfaces exposed in the type's interface.
- transfers are the ways values can be exchanged between different parts of the program, and include copy, move and binding.

4.6.1. Lifetime

The lifetime of a value is the amount of time during which the value exists in the program, in other words the time between its creation and its destruction.

An entity is said to be *live* if it was created but not yet destroyed. It is said to be *dead* otherwise.

> Some entities may be live but not accessible from within the current context because they are not visible. This is the case for variables declared in the caller's context.

The lifetime information known by the compiler about entity `X` is represented as compile-time constant `lifetime X`. The lifetime values are equipped with a partial order `<`, such that the expression `lifetime X < lifetime Y` being `true` is a compiler guarantee that `Y` will always be live while `X` is live. It is possible for neither `lifetime X < lifetime Y` nor `lifetime X > lifetime Y` to be true. This `lifetime` feature is used to implement Rust-like restrictions on access types, i.e. a way to achieve memory safety at zero runtime cost.

The lifetime of XL values fall in one of the following categories:

- *global values* become live during the declaration phase of the program, just before its evaluation phase, and they remain live until the end of that evaluation phase. Global values are typically preallocated statically in a reserved area of memory, before any program evaluation, by a program called a linker.

- *local values* become live during the local declaration phase of the bodies of the declarations corresponding to patterns being matched during the evaluation phase. Local values are typically allocated dynamically on a call stack allocated for each thread of execuion. That stack has a limited "depth", which may limit the depth of recursion allowed for a program.
- *dynamic values* are dynamically allocated using a "heap", and remain live as long as some other value owns them. That owning value may itself be a global, local or dynamic value. The heap is typically the largest available memory space for the program. XL offers a number of facilities to help you manage how this dynamic allocation happens, including facilities to build garbage collectors if and when this is an efficient management strategy.
- *temporary values* are created during evaluation of expressions, and can be discarded as soon as they have been consumed. For example, assuming a definition for `x+y`, the expression `a+b+c+d` will be processed as `((a+b)+c)+d`, and the result of evaluating `a+b` can be destroyed as soon as `(a+b)+c` has been evaluated. Temporary values are typically also allocated on the stack.

For example, consider the following piece of code:

```
use XL.CONSOLE.TEXT_IO ①
use XL.TEXT.FORMAT

print "Starting printing Fibonacci sequences" ②

fib 0 is 1 ③
fib 1 is 1
fib N is (fib(N-1) + fib(N-2)) ④

for I in 1..5 loop ⑤
    F is fib I ⑥
    print format("Fib(%1) is %2", I, F) ⑦
```

① The `use` statements import global values defined in other files. Here, the `XL` module, its sub-module `XL.CONSOLE`, and a third-level sub-module `XL.CONSOLE.TEXT_IO` are all imported by the first statement, and similarly, `XL.TEXT` and `XL.TEXT.FORMAT` are added by the second statement (`XL` being already imported)

② The declaration of `print` is a global value defined in `XL.CONSOLE.TEXT_IO`. This `print` statement is the first thing to be executed during program evaluated. Its evaluation will call code for an implementation of `print`, thereby adding a new context on the call stack. As we indicated earlier, this may involve further calls making the call stack deeper.

③ The three definitions with `fib` as a prefix are three distinct global values, even if, thanks to dynamic dispatch, they may be considered as implementing a single entity. Evaluating `fib N` or `fib I` may require considering all three global values as candidates.

④ The evaluation of the expression `(fib(N-1)+fib(N-2))` will need to create a number of temporaries, for example to compute `N-1` or `N-2`. Temporaries may be destroyed as soon as they are no longer needed. For example, the code to evaluate the expression could be someting similar to the following code, where `tmp1`, `tmp2`, `tmp3`, `tmp4` and `tmp5` are the required temporaries:

```
tmp1 is N-1
tmp2 is fib(tmp1)
delete tmp1
tmp3 is N-2
tmp4 is fib(tmp3)
delete tmp3
tmp5 is tmp2+tmp4
delete tmp2
delete tmp4
tmp5
```

⑤ The `for` loop creates a local variable named `I` that will successively take values `1`, `2`, `3`, `4`, `5`. The value for `I` will only be live within one iteration of the loop. In other words, the execution will be identical to the following (using a closure for the different values of `I`):

```
tmpBody is
    F is fib I
    print format("Fib(%1) is %2", I, F)
{ I is 1 } ( tmpBody )
{ I is 2 } ( tmpBody )
{ I is 3 } ( tmpBody )
{ I is 4 } ( tmpBody )
{ I is 5 } ( tmpBody )
```

⑥ The value defined by `F is fib I` is a local value that will be live for the duration of the evaluation of the enclosing block. It will be destroyed the end of each block. In other words, a more accurate description for `tmpBody` in the example above would have a `delete F` statement at the end, as follows:

```
tmpBody is
    F is fib I
    print format("Fib(%1) is %2", I, F)
    delete F
```

⑦ It may come as a suprise to people coming from C or C++ that XL does not *require* the call to `fib I` to be done before this point. The definition `F is fib I` can be read as either a constant initialized with `fib I`, or as a function returning `fib I`. If the compiler can determine that the result of calling evaluating `F` will always be identical, it is allowed to implement memoization, i.e. to store the value computed for `F` the first time, for example in an expression like `F+F`.

> Typically, a good compiler also makes use of machine *registers*, very fast storage in the processor itself, as a cache for values that are logically part of the call stack. In general, we will only talk about the stack, with the understanding that this includes registers where applicable.

4.6.2. Creation

Creation is the process of preparing a value for use. The XL language rules guarantee that values are never undefined while the value is live, by calling programmer-supplied code at the appropriate times.

The lifetime of a value `V` begins by implicitly evaluating a _creation_statement `create V`. This happens in particular if you create a local variable without initializing it.

For example, consider the following code:

```
Add Z:complex is
   T:complex
   Z+T
```

The code above is really equivalent to the following, where the implicitly-generated code has been put between parentheses:

```
Add Z:complex is
   T:complex
   (create T)
   Z+T
```

Values in containers receive well-defined values through creation. For example, if you create an `array[1..5] of complex`, the 5 `complex` values are created before you can access them.

A `create` operation must take a single `out` argument. All the values in this `out` argument are themselves created before the body of the definition begins. For example, consider the following:

```
create Z:out complex is
    print "Creator called"
```

This code does not lead to uninitialized values, because it is really equivalent to the following:

```
create Z:out complex is
   (create Z.Re)        // Implicit creation of complex fields
   (create Z.Im)
   print "Creator called"
```

The `create` operator for basic types is said to *zero initialize* them as follows:

- `type` and `nil` values receive `nil`.
- All integer types receive value `0`
- All character types receive `character 0`.
- `text` receive `""`.

- `boolean` receive `false`.

The `create` operation can be called by the programmer, and therefore must behave correctly if it is called multiple times. This is true by default because of the rule that `out` parameters are destroyed before a call.

For example, if you explicitly call `create` as in the following code:

```
Z:complex
create Z
```

this is really equivalent to the following, where implicit statements are between parentheses:

```
Z:complex
(create Z)      // because of the declaration above
(delete Z)      // because Z is passed as out argument to 'create'
create Z
```

The compiler may be able to elide some of these calls in such cases. Another important case where a compiler should elide creation calls is called *construction*, and is based on the shape defined for types. When you define a type, you need to specify the associate shape. For example, we defined a `complex` type as follows:

```
complex is type complex(Re:real, Im:real)
```

This means that a shape like `complex(2.3, 5.6)` is a `complex`. This also means that the *only* elementary way to create a `complex` is by creating such a shape. It is therefore not possible to have an uninitialized element in a `complex`, since for example `complex(1.3)` would not match the shape and not have the right type.

Using the shape explicitly given for the type is called the *constructor* for the type, and can be used in definitions or in variable declarations with an initial value. A constructor can never fail nor build a partial object. If an argument returns an error during evaluation, then that `error` value will not match the expected argument, except naturally if the constructor is written to accept `error` values.

Often, developers will offer alternate ways to create values of a given type. These alternate helpers are nothing else than regular definitions that return a value of the type.

For example, for the `complex` type, you may create an imaginary unit, `i`, but you need a constructor to define it. You can also recognize common expressions such as `2+3i` and turn them into constructors.

```
i   is complex(0.0, 1.0)

syntax { POSTFIX 190 i }
Re:real + Im:real i                  is complex(Re, Im)      // Case 1
Re:real + Im:real * [[i]]            is complex(Re, Im)      // Case 2
Re:real + [[i]] * Im:real            is complex(Re, Im)      // Case 3
Re:real as complex                   is complex(Re, 0.0)     // Case 4
X:complex + Y:complex as complex     is ...

2 + 3i              // Calls case 1 (with explicit concersions to real)
2 + 3 * i           // Calls case 2 (with explicit conversions to real)
2 + i * 3           // Calls case 3
2 + 3i + 5.2        // Calls case 4 to convert 5.2 to complex(5.2, 0.0)
2 + 3i + 5          // Error: Two implicit conversions (exercise: fix it)
```

The fact that the only elementary way to create a type is through the constructor is illustrated by the following code:

```
large is type (N when N > 42)
A:large := 44   // OK
B:large := 99.1 // OK
C:large         // Error
create V:out large is
    print "Creator called"
```

The `A:large` and `B:large` initializations are acceptable, because it is possible to validate that the initial values match the `large` pattern. The `C:large` definition, however, is not acceptable, despite the presence of a `create` operation. The reason is that there is no way to `create V.N`, first because the type to use cannot be deduced, second because if we picked a type like `integer`, the default initial value `0` would not match the pattern.

The code above can be fixed, however, by using a constructor for the `large` type inside the creator, which means supplying a value that matches the type's pattern. The following is an acceptable version of the `create` function:

```
large is type (N when N > 42)
C:large         // OK
create V:out large is
    print "Creator called"
    V := 44     // Creator for a large value
```

A type implementation may be *hidden* in a module interface, in which case the module interface should also provide some functions to create elements of the type. The following example illustrates this for a `file` interface based on Unix-style file descriptors:

```
MY_FILE as module with
    file as type
    open(Name:text) as file
    close F:io file

MY_FILE as module is
    file is type file(fd:integer)
    open(Name:text) as file is
        fd:integer := libc.open(Name, libc.O_RDONLY)
        file(fd)
    close F:inout file is
        if fd >= 0 then
            libc.close(F.fd)
            F.fd := -2
    delete F:inout file is close F    // Destruction, see below
```

If the interface provides a `create` operation, it must be ready to accept default-created values as input in all other functions of the module. In the module above, however, the only way to get a value of the `file` type is by using the `open` function. This also means that you cannot create a variable of type `file` without initializing it.

> **RATIONALE** This mechanism is similar to *elaboration* in Ada or to *constructors* in C++. It makes it possible for programmers to provide strong guarantees about the internal state of values before they can be used. This is a fundamental brick of programming techniques such as encapsulation, programming contracts or RAII.

4.6.3. Destruction

When the lifetime of a value `V` terminates, the statement `delete V` automatically evaluates. Declared entites are destroyed in the reverse order of their declaration. A `delete X:T` definition is called a *destructor* for type `T`. It often has an inout parameter for the value to destroy, in order to be able to modify its argument, i.e. a destructor often has a signature like `delete X:inout T`.

Symmetrical to creation, the body of a `delete V` automatically invokes `delete V.X` for any field `X` in `V` at exit of the body of the definition.

For example, consider the definition below:

```
delete Z:inout complex is
    print "Deleting complex ", Z
```

That definition is actually equivalent to the definition below:

```
delete Z:inout complex is
    print "Deleting complex ", Z
    (delete Z.Im)
    (delete Z.Re)
```

There is a built-in default definition of that statement that has no effect and matches any value, and which only deletes the fields:

```
delete Anything is nil
```

There may be multiple destructors that match a given expression. When this happens, normal lookup rules happen. This means that, unlike languages like C++, a programmer can deliberately override the destruction of an object, and remains in control of the destruction process. More importantly, this means that the destruction process respects the global type semantics.

Consider for example the deletion of the `file` type defined in the `MY_FILE` module above. Since there is a special case for negative values, that might be reflected in the implementation as follows:

```
delete F:inout file when F.fd < 0  is ... // Invalid flie
delete F:inout file                is ... // Valid file
```

However, this also means that the programmer could create a `valid_file` type corresponding to the case where `F.fd<0` is false. If you have a `valid_file` value to `delete`, normal type system and lookup rules ensure that the second case will be selected.

Consider another interesting example, where you have the following declarations:

```
positive is type (N when N > 0)
integers is string of integer
N:integers > 0 as boolean is
    for I in N loop
        if not (I > 0) then
            return false
    return true

delete N:inout positive is
    print "Deleting positive: ", N

delete N:integer is
    print "Deleting integer: ", N

delete N:integers is
    print "Deleting integers with size: ", size N

example is
   print "Beginning example"
   A:integers := string(1,8,4)
   B:integers := string(-1,0,5)
   print "End of example"
```

In this example, we create an `integers` type based on `string of integers`, for which we implement the `N>0` operator to mean that all elements in the `string` are positive. This in turn means that some values that have type `integers` also have type `positive`. In the body of `example`, `A` is `positive`, but `B` is not.

If one consider implicit inheritance and the implicitly inserted field destruction, the code for the `integers` type and `delete` operations above is really equivalent to the following:

```
integers is type(base:string of integer)

delete P:inout positive is
    print "Deleting positive: ", P
    (delete P.N)        // Delete the N bound in 'positive'

delete N:integer is
    print "Deleting integer: ", N
    (delete N.base)     // For 'integer', this is a no-op

delete S:integers is
    print "Deleting integers with size: ", size S
    (delete S.base)     // Delete the underlying 'string of integer'
```

As a result, the output of this program should be something like:

```
Beginning example
End of example
Deleting integers with size 3 ①
Deleting positive: 5 ②
Deleting integer: 5 ③
Deleting integer: 0 ④
Deleting integer: -1
Deleting positive: string(1,8,4) ⑤
Deleting integers with size 3 ⑥
Deleting positive: 4
Deleting integer: 4
Deleting positive: 8
Deleting integer: 8
Deleting positive: 1
Deleting integer: 1
```

① This is deleting local variable `B` using type `integers`, knowing that it failed to pass the test for `positive` because of value `-1`.

② This is deleting values in the `string of integer` container in local variable `B`, starting with the last one. Containers can destroy their values in any order, but for `string`, an efficient algorithm may start with the end of the container in order to be able to truncate before each element being removed simply by changing a "number of items" in the `string`. The local `delete` definitions are visible to the instantiation of `delete` for the type `string of integer` that is made for the call at the end of `example`. The first matching definition for value `5` is for the `positive` type.

③ This is implicitly deleting the `integer` value called `P.N` in the code above.

④ For value `0` in the `string of integer` value held in `B`, the `positive` test failed, so that the first destructor that works is for `integer`.

⑤ This is deleting local variable `A`. Since `A` is positive, the destructor for positive is called.

⑥ Unlike what happened for `B`, the destructor for `integers` is not called directly for `B` but implicitly for `P.N`.

It is possible to create local destructor definitions. When such a local definition exists, it is possible for it to override a more general definition. The general definition can be accessed using super lookup, and generally, it should in order to preserve the language semantics.

```
show_destructors is
    delete Something is
        print "Deleted", Something
        super.delete Something
    X is 42
    Y is 57.2
    X + Y
```

This should output something similar to the following:

```
Deleted 42.0
Deleted 57.2
Deleted 42
```

The first value being output is the temporary value created by the necessary implicit conversion of `X` from `integer` to `real`. Note that additional temporary values may appear depending on the optimizations performed by the compiler. The value returned by the function should not be destroyed, since it's passed to the caller.

Any destruction code must be able to be called multiple times with the same value, if only because you cannot prevent a programmer from writing:

```
delete Value
```

In that case, `Value` will be destroyed twice, once by the explicit `delete`, and a second time when `Value` goes out of scope. There is obviously no limit on the number of destructions that an object may go through.

```
for I in 1..LARGE_NUMBER loop
    delete Value
```

Also, remember that passing a value as an `out` argument implicitly destroys it. This is in particular the case for the target of an assignment.

4.6.4. Errors

Errors in XL are represented by values with the `error` type, or any type that inherits from error. The error type has a constructor that takes a simple error message, or a simple message and a payload.:

```
error as type is one_of
    error Message:text
    error Message:text, Payload
```

The message is typically a localizable format text taking elements in the payload as numbered argument in a way similar to the `format` function:

```
log X:real as error when X <= 0 is
    error "Logarithm of negative value %1", X
```

A function that may fail will often have a `T or error` return value. There is a specific shortcut for that, `mayfail T`:

```
mayfail T:type as type is T or error
```

For example, a logarithm returns an error for non-positive values, so that the signature of the `log` functions is:

```
log X:real as mayfail real     is ... // May return real or error
```

If possible, error detection should be pushed to the interface of the function. For the `log` function, it is known to fail only for negative or null values, so that a better interface would be:

```
log X:real as real  when X > 0.0    is ... // Always return a real
log X:real as error                 is ... // Always return an error
```

With the definitions above, the type of `log X` will be `real` if it is known that `X > 0.0`, `error` if it is known that the condition is false, and `real or error`, i.e. `mayfail real`, in the more general case. A benefit of writing code this way is that the compiler can more easily figure out that the following code is correct and does not require any kind of error handling:

```
if X > 0.0 then
    print format("Log(%1) is %2", X, log X)
```

RATIONALE By returning an `error` for failure conditions, XL forces the programmer to deal with errors simply to satisfy the type system. They cannot simply be ignored like C return values or C++ exceptions can be. Errors that may possibly return from a function are a fundamental part of its type, and error handling is not optional.

A number of types derive from the base `error` type to feature additional properties:

- A `range_error` indicates that a given value is out of range. The default message provided is supplemented with information comparing the value with the expected range.

```
T:text[I:offset] as character or range_error is
    if I >= length T then
        range_error "Text index %2 is out of bounds for text %2", I, T
    else
        P : memory_address[character] := memory_address(T.first)
        P += I
        *P
```

- A `logic_error` indicates an unexpected condition in the program, and can be returned by contract checks like `assert`, `require` and `ensure`.

```
if X > 0 then
    print "X is positive"
else if X < 0 then
    print "X is negative"
else
    logic_error "Some programmer forgot to consider this case"
```

- A `storage_error` is returned whenever a dynamic value is created, notably each time an `own T` object is created, but also when additional storage is needed for containers.

```
S : string of integer           // The string requires storage
loop
    V : own integer := 3        // This allocates an integer, freed each loop
    S &= V                      // Accumulate integers in an unbounded way
```

- A `file_error` reports when there is an error opening a file, for example because a file does not exist.
- A `permission_error` reports when a resource access is denied, whether it's a file or any other resource.
- A `compile_error` helps the compiler emit better diagnostic for situations which would lead to an invalid program. All errors can be emitted at compile-time if the compiler can detect that they will occur unconditionally, but `compile_error` makes it clearer that this is intended to detect an error at compile-time. A variant, `compile_warning`, emits a message but lets the compilation proceed.

```
// Emit a specific compile-time error if assigning text to an integer
X:integer := Y:text is
    compile_error "Cannot assign text %1 to integer %2", Y, X

// Emit a specific warning when writing a real into an integer
X:integer := Y:real is
    compile_warning "Assigning real to integer may lose data"
    T is integer Y
    if real T = Y then
        X := T
    else
        range_error "Assigned real value %1 is out of range for integer", Y
```

4.6.5. Mutability

A value is said to be *mutable* if it can change during its lifetime. A value that is not mutable is said to be *constant*. A mutable named entity is called a *variable*. An immutable named entity is called a *named constant*.

The `X:T` type annotations indicates that `X` is a mutable value of type `T`, unless type `T` is explicitly marked as constant. When `X` is a name, the annotation declares that `X` is a variable. The `X as T` type annotation indicates that `X` is a constant value of type `T`, unless type `T` is explicitly marked as variable. When `X` is a name, this may declare either a named constant or a function without parameters, depending on the shape of the body.

```
StartupMessage : text := "Hello World"  // Variable
Answer as integer is 42                 // Named constant
```

A mutable value can be initialized or modified using the `:=` operator, which is called an *assignment*. There are a number of derived operators, such as `+=`, that combine a frequent arithmetic operation and an assignment.

```
X : integer := 42       // Initialize with value 42
X := X or 1             // Binary or, X is now 43
X -= 1                  // Subtract 1 from X, now 42
```

Some entities may give access to individual inner values. For example, a `text` value is conceptually made of a number of individual `character` values that can be accessed individually. This is true irrespective of how `text` is represented. In addition, a `slice` of a `text` value is itself a `text` value. The mutability of a `text` value obviously has an effect on the mutability of accessed elements in the `text`.

The following example shows how `text` values can be mutated directly (1), using a computed assignment (2), by changing a slice (3) or by changing an individual element (4).

```
Greeting : text := "Hello"              // Variable text
Person as text is "John"                // Constant text
Greeting := Greeting & " " & Person     // (1) Greeting now "Hello John"
Greeting &= "!"                         // (2) Greeting now "Hello John!"
Greeting[0..4] := "Good m0rning"        // (3) Greeting now "Good m0rning John!"
Greeting[6] := 'o'                      // (4) Greeting now "Good morning John!"
```

None of these operations would be valid on a constant text such as `Person` in the code above. For example, `Person[3]:='a'` is invalid, since `Person` is a constant value.

> In the case (3) above, modifying a `text` value through an access type can change its length. This is possible because `Greeting[0..4]` is not an independent value, but an access type, specifically a `slice`, which keeps track of both the `text` (`Greeting` here) and the index range (`0..4` in that case), with a `:=` operator that modifies the accessed `text` value.

A constant value does not change over its lifetime, but it may change over the lifetime of the program. More precisely, the lifetime of a constant is at most as long as the lifetime of the values it is computed from. For example, in the following code, the constant `K` has a different value for every interation of the loop, but the constant `L` has the same value for all iterations of `I`

```
for J in 1..5 loop
    for I in 1..5 loop
        K is 2*I + 1
        L is 2*J + 1
        print "I=", I, " K=", K, " L=", L
```

RATIONALE There is no syntactic difference between a constant and a function without parameters. An implementation should be free to implement a constant as a function if this is more effective, or to use smarter strategies when appropriate.

4.6.6. Compactness

Some data types can be represented by a fixed number of contiguous memory locations. This is the case for example of `integer` or `real`: all `integer` values take the same number of bytes. Such data types are called *compact*.

On the other hand, a `text` value can be of any length, and may therefore require a variable number of bytes to represent values such as `"Hi"` and `"There once was a time where text was represented in languages such as Pascal by fixed-size character array with a byte representing the length. This meant that you could not process text that was longer than, say, 255 characters. More modern languages have lifted this restriction."`. These values are said to be *scattered*.

Scattered types are always built by *interpreting* compact types. For example, a representation for text could be made of two values, the memory address of the first character, and the size of the text. This is not the only possible representation, of course, but any representation require interpreting fixed-size memory locations and giving them a logical structure.

Although this is not always the case, the assignment for compact types generally does a copy, while the assignment for scattered types typically does a move.

4.6.7. Ownership

Computers offer a number of *resources*: memory, files, locks, network connexions, devices, sensors, actuators, and so on. A common problem with such resources is to control their *ownership*. In other words, who is responsible for a given resource at any given time.

In XL, like in languages like Rust or C++, ownership is largely determined by the type system, and relies heavily on the guarantees it provides, in particular with respect to creation and destruction. In C++, the mechanism is called RAII, which stands for *Resource Acquisition is Initialization*. The central idea is that ownership of a resource is an invariant during the lifetime of a value. In other words, the value gets ownership of the resource during construction, and releases this ownership during destruction. This was illustrated in the `file` type of the module `MY_FILE` given earlier.

Types designed to own the associated value are called *owner types*. There is normally at most one live owner at any given time for each controlled resource, that acquired the resource at construction time, and will release it at destruction time. It may be possible to release the owned resource early using `delete Value`.

The standard library provides a number of types intended to own common classes of resources, including:

- An `own` value owns a single item allocated in dynamic storage. Note that the value `nil` is not a valid `own` value (except for `own nil`). If you need `nil` as a value, you must use `own T or nil`.
- An `array`, a `buffer` and a `string` all own a contiguous sequence of items of the same type.
  - An `array` has a fixed size during its lifetime and allocates items directly, e.g. on the execution stack.
  - A `buffer` has a fixed size during its lifetime, and allocates items dynamically, typically from a heap.
  - A `string` has a variable size during its lifetime, and consequently may move items around in memory as a result of specific operations.
- A `text` owns a variable number of `character` items, and inherits from the `string of character` type.
- A `file` owns an open file.
- A `mutex` owns execution by a single thread while it's live.
- A `timer` owns a resource that can be used to measure time and schedule execution.
- A `thread` owns an execution thread and the associated call stack.
- A `task` owns an operation to perform that can be dispatched to one of the available threads of execution.
- A `process` owns an operating system process, including its threads and address space.
- A `context` captures an execution context.

4.6.8. Access types

Not all types are intended to be owner types. Many types delegate ownership to another type. Such types are called *access types*. When an access type is destroyed, the resources that it accesses are *not* disposed of, since the access type does not own the value. A value of the acces type merely provides *access* to a particular value of the associated owner type.

For example, if `T` is a `text` value and if `A` and `B` are `integer` values, then `T[A..B]` is a particular kind of access value called a *slice*, which denotes the fragment of text between `0`-based positions `A` and `B`. By construction, slice `T[A..B]` can only access `T`, not any other `text` value. Similarly, it is easy to implement bound checks on `A` and `B` to make sure that no operation ever accesses any `character` value outside of `T`. As a result, this access value is perfectly safe to use.

Access types generalize *pointers* or *references* found in other languages, because they can describe a much wider class of access patterns. A pointer can only access a single element, whereas access types have no such restriction, as the `T[A..B]` example demonstrates. Access types can also enforce much stricter ownership rules than mere pointers.

> The C language worked around the limitation that pointers access a single element by abusing so-called "pointer arithmetic", in particular to implement arrays. In C, `A[I]` is merely a shortcut for `*(A+I)`. This means that `3[buffer]` is a valid way in C to access the third element of `buffer`, and that there are scenarios where `ptr[-1]` also makes sense as a way to access the element that precedes `ptr`. Unfortunately, this hack, which may have been cute when machines had 32K of memory, is now the root cause of a whole class of programming errors known as *buffer overflows*, which contribute in no small part to the well-deserved reputation of C as being a language that offers no memory safety whatsoever.

The standard library provides a number of types intended to access common owner types, including:

- A `ref` is a reference to a live `own` value.
- A `slice` can be used to access range of items in contiguous sequences, including `array`, `buffer` or `string` (and therefore `text` considered as a `string of character`).
- A `reader` or a `writer` can be used to access a `file` either for reading or writing.
- A `lock` takes a `mutex` to prevent multiple threads from executing a given piece of code.
- Several types such as `timing`, `dispatch`, `timeout` or `rendezvous` will combine `timer`, `thread`, `task` and `context` values.
- The `in`, `out` and `inout` type expressions can sometimes be equivalent to an access types if that is the most efficient way to pass an argument around. However, this is mostly invisible to the programmer.
- An `XL.SYSTEM.MEMORY.address` references a specific address in memory, and is the closest there is in XL to a raw C pointer. It is purposely verbose and cumbersome to use, so as to discourage its use when not absolutely necessary.

4.6.9. Inheritance

A type is said to *inherit* another type, called its *base type*, if it can use all its operations. The type is then said to *derive* from the base type. In XL, this is achieved simply by providing an *implicit conversion* between the derived type and the base type:

```
Derived:derived as base is ...
```

As a consequence of this approach, a type can derive from any number of other types, a feature sometimes called *multiple inheritance*. There is also no need for the base and derived type to share any specific data representation, although this is often done in practice. For example, there is an implicit conversion from `i16` to `i32`, altough the machine representation is different, so in XL, one can say that `i16` derives from `i32`.

Sometimes, it is necessary to denote a type that inherits from a specific type. For example, if you want to create a constrained generic type for `complex`, you might want it to accept only `number` for its type argument. The following code is an incorrect way to do it, since it creates a type that only accept `number` *values* as an argument, i.e. it would accept `complex[3.5]` but not `complex[real]`:

```
some_complex is type complex[real:number]
```

To denote a type that derives from a base type, one can use the `type like base` notation. The correct way to implement the above restriction is as follows, which indicates that the argument is a type, not a value:

```
some_complex is type complex[real:type like number]
```

The notation `type like base` can also be used in the interface for a type. In both cases, the notation is a promise that the type in question can be implicitly converted to the base class. It does not specify how this is done. One way, indicated above, is to provide an implicit conversion function. Another way is by adopting a similar data structure, an approach called data inheritance.

4.7. Subtypes

A type can be given additional constraints, which define a *subtype*. A subtype can always be converted to the type it was derived from, and therefore derives from that type in the inheritance sense. A subtype machine representation may differ from the type it derives from.

For example, from the `integer` type, one can construct a `month` type that matches only `integer` values between `1` and `12` using a regular conditional pattern as follows:

```
month is type(M:integer when M >= 1 and M <= 12)
```

The language defines a number of standard subtypes. All these subtypes can be implemented using regular language features.

4.7.1. Range subtypes

Subtyping to select a range is common enough that there is a shortcut for it. For any type with an order, subtypes can be created with the `range` infix operator, creating a *range subtype*:

```
T:type range Low:T..High:T      is type(X:T when X in Low..High)
```

With this definition, the `month` type can be defines simply as follows:

```
month is integer range 1..12
```

4.7.2. Size subtypes

The infix `bits` operator creates a *size subtype* with the specified number of bits. It applies to `real`, `integer` and `character` types.

For example, the `i8` type can be defined as:

```
i8 is integer bits 8
```

This implicitly implies a `range` that depends on the type being subtyped. For example, for `integer` and `natural`, the range would be defined as follows:

```
[[integer]] bits N:natural      is integer range -2^(N-1)..2^(N-1)-1
[[natural]] bits N:natural      is natural range  0..2^N-1
```

The `bits` subtypes are intended to specify the bit size of the machine representation. The requested size may be rounded up to a more convenient or more efficient machine representation. For example, on a 32-bit machine, `integer bits 22` might be more efficiently represented as a 32-bit value in registers and as 3 bytes, i.e. 24 bits, in memory.

4.7.3. Real subtypes

The `real` type can be subtyped with a `range` and a `bits` size, as well as with additional constraints more specific to the `real` type:

- a `digits` count specifies the number of accurate decimal digits, For example, `real digits 3` is represents values with at least 3 significant digits.
- a `quantum` followed by a literal real value specifies a representation that should be representable exactly. For example, `real quantum 0.25` that value `0.25` must be represented exactly.
- an `exponent` specifies the maximum decimal exponent. For example, `real exponent 100` will ensure that values up to `1.0e100` can be represented.
- a `base` specifies the base for the internal representation. Only bases `2`, `10` and `16` are allowed. Base `2` requires a binary floating-point representation. Base `10` requires a decimal floating-point representation. Base `16` requires an hexadecimal floating-point representation on historical platforms that support it.

In some cases, the types created using one of these operators may not be subtypes of `real`. For example, on any machine using machine using the most common floating-point representation available in hardware, IEEE-754, the value `0.01` cannot be represented accurately. This means that an implicit conversion from `real quantum 0.01` to `real` would implicitly destroy accuracy. As a result, `real quantum 0.01` must be converted to `real` *explicitly*. In some other cases, implementation limitations may cause errors. For example, an implementation is not required to accept `real digits 100`.

A `real` subtype should be represented using *fixed-point arithmetic* if one of the following conditions is true:

- The `exponent` is specified as `0`.

- The `range` is small enough to be representable entirely with the same exponent and the available number of bits.
- A `quantum` is specified and no `exponent` is specified.

For example, the `hundredth` type defined below could be represented internally by an `natural` values between `0` and `100`, and converted to `real` by multiplying this value by the given `quantum` value.

```
hundredth is real range 0.0..1.0 quantum 0.01
```

4.7.4. Saturating subtypes

The `saturating` prefix operator can be used on `integer` and `real` types (primarily intended for use with fixed-point subtypes) to select *saturation arithmetic* in case of overflow.

For example, a `color_component` type that has values between `0.0` and `1.0` and saturates can be defined as follows:

```
color_component is saturating real range 0.0..1.0 bits 16
Red : color_component := 0.5
Red += 0.75              // Red is now 1.0
```

4.7.5. Character subtypes

The `character` types can be subtyped with the `range` and `bits` operators:

```
letter is character range 'A'..'Z'
ASCII  is character bits 7
```

In addition, character types can be subtyped with the following infix operators:

- The `encoding` operator specifies the encoding used for the text, for example `character encoding "ASCII"`.
- The `locale` operator specifies the locale for the text, for example `character locale "fr_FR"` will select a French locale.
- The `collation` operator specifies collating order. For example, to have `character` values that sort following German rules, you would use `character collation "de_DE"`

4.7.6. Text subtypes

The `text` type can be subtyped with the `encoding`, `locale` and `collation` operators, with the same meaning as for `character`.

4.7.7. Memory access subtypes

Many recent machines provide several ways to access memory, for example to deal with synchronization between multiple CPUs or between CPUs and memory-mapped devices. XL presents this kind of features as subtypes.

The following standard subtypes implement memory-related semantics:

- `atomic T` is a type derived from `T` that offers atomic operations. This may come at the expense of performance.
- `unaligned T` is a type derived from `T` that may not respcet normal alignment rules, and may require slower mis-aligned accesses.
- `T aligned N` guarantees that values of type `T` use memory aligned on `N` bytes boundaries.
- `volatile T` is a type derived from `T` where the compiler cannot assume that the value does not change externally due to causes external to the current code.
- `uncached T` is a type derived from `T` where the compiler should ensure that data accesses are consistent with external memory, even if this is significantly more expensive.
- `packed T` is a type derived from `T` where data is packed as tightly as possible, even to the detriment of performance.

Additional subtypes may be available to match the features of the machine the code runs on.

4.8. Type interface

The interface of a type specifies how the type can be used, and what operations can be performed with it. Specifically, the interface defines:

- *features* of the type, which are elements that are visible in the scope defined by a value of the type. Mutable features are sometimes called *fields*, whereas constant features are sometimes called *methods*.
- *inheritance* of the type, which indicates what type, if any, the type derives from.

A feature is said to be *advertised* if it is explicitly and intentionally part of the interface. A feature is said to be *exposed* if it is made visible as an unintentional or even undesirable side effect of the interface. For example, if an implementation reqauires dynamic values, this may force the interface to expose the `storage_error` values that might be generated as a result of out-of-memory conditions.

The code below defines a `picture` type that advertises `width`, `height` and `pixels` fields, as well as an `area` method that is used to compute the total number of pixels and is the size for the `pixels` buffer. In this interface, one might argue that the fact that `pixels` is a `buffer` falls more in the "exposed" category than "advertised".

```
picture as type with
    width  : size
    height : size
    pixels : buffer[area] of byte
    area as size
```

The code below indicates that the type `text` derives from `string of character` with additional features:

```
text as type like string of character with
    byte_count as size                  // Number of bytes used by characters
    as_number[T:type like number] as T  // Numerical conversion
```

An *abstract type* is a type for which features or inheritance are not provide. The only thing known about such a type is its name. In the earlier `MY_FILE` example, `file` was an abstract type defined as folows:

```
file as type
```

A type for which only the interface is known is called a *tag type*. Since nothing is known about the parse tree shape associated with the type, a tag type can only match values that were *tagged* with the same type using some explicit type annotation.

In partticular, knowing only the interface of a type does not allow values of the type to be created. It is sometimes useful or desirable to preclude the creation of values of the type, for example when creating true generic types. For most concrete types, however, the interface of a function creating values of the type should generally also be provided. In the rest of the discussion for the `picture` type, we will assume that there is a `picture` function returning a `picture` value with the following function interface:

```
picture(width:size, height:size) as picture
```

4.8.1. Information hiding

The interface of a type does not reveal any information on the actual implementation of the type, e.g. on the shape of the parse tree associated with it. This is called *information hiding* and is the primary way in XL to achieve *encapsulation*.

While the type interface for `picture` above does not give us any clue about how the type is actually implemented, it still provides very useful information. It remains sufficient to validate code that uses values of the type, like the following definition of `is_square`:

```
is_square P:picture is P.width = P.height
```

In that code, `P` is properly tagged as having the `picture` type, and even if we have no idea how that type is implemented, we can still use `P.width` and deduce that it's an `integer` value based on the type interface alone.

Information hiding is specially useful in the context of modules, where the interface and implementation typically reside in different source files.

4.8.2. Direct implementation

The simplest way to implement any feature of a type is to ensure that the implementation of the type has a matching feature. This is called a *direct implementation* of the feature.

A feature is directly implemented by providing a definition for it in the pattern matching scope for the type implementation. In particular, the formal parameters for a constructor naturally provide a direct implementation for the interface. For example, a constructor for type `picture` might look like:

```
picture is type picture
    pixels : buffer[area] of u8
    width  : size
    height : size
```

In a scope where the implementation is known, if `P` is a `picture`, we know that there is a pattern matching scope in `P` which contains the bindings of the formal parameters to the argument values. This pattern matching scope is where the binding, and therefore value of `width` in `P.width` is found.

In a context where only the interface is visible, an expression like `P.width` is known to be valid because we can look it up in the interface for the `picture` type. However, in order to identify the implementation for `P.width`, we must look this expression up in a context where the implementation of `picture` is known. Finding an implementation definition that matches the interface definition is the mechanism underlying direct implementation of the feature.

4.8.3. Data inheritance

To implement inheritance, a direct implementation is to simply reuse the existing data in the original type, possibly adding more to it. This is called *data inheritance*, and is implemented by using the `T with Fields` notation.

For example, using data inheritance to implement a `colored_text` that derives from the `text` type and adds a color, the interface might be:

```
colored_text as type like text with
    foreground : color
    background : color
```

An implementation using data inheritance would look like this:

```
colored_text is text with
   background : color
   foreground : color
```

As shown in the example, the fields need not be in the same order in the implementation as in the interface. In the resulting implementation type, a field named `base` refers to the base type on the left of `with`. For values of that derived type, the field `base` refers to the base value. Using data inheritance lets the translator automatically generate implicit conversion code that takes a derived value and returns its base. In our case, that implicit conversion would look like this:

```
derived:colored_text as text is derived.base
```

4.8.4. Indirect implementation

However, the implementation may be entirely different from the interface, as long as any expression that is valid knowing the interface is valid in a context where the implementation is known. An implementation that provides an advertised feature of the interface withtout using a similar definition is called an *indirect implementation* of that feature.

A simple case of indirect implementation for inheritance is to provide an implicit conversion function. For example, the implementation of `text` may simply be defined in a scope that also features the following function:

```
T:text as string of character is convert_to_string(T)
```

While it is reasonable to infer from the interface that `text` and `string of character` might share a common internal representation, this is only an efficient way to provide inheritance, but it is not a constraint on the implementation.

A type may also provide entirely different implementations of its features, while still allowing individual features to be accessed almost as if they were provided by a direct implementation. We will see a number of examples in the next sections.

4.8.5. Delegation

An interesting case is when the type *delegates* most of its implementation to some other type. For example, the `picture` type might actually be using a `bitmap` type as its internal representation:

```
bitmap as type with
    width  : u16
    height : u16
    buf    : array[width, height] of u8
picture as type picture
    bits:bitmap
```

Such an implementation of the `picture` type must perform some serious adjustments in order to delegate the work to the underlying `bitmap` value while providing the expected interface. Dealing with the `width` and `height` fields seems relatively straightforward:

```
picture is type picture
    bits:bitmap

    width  is bits.width
    height is bits.height
```

This, however, will not work as is, because we have only provided a way to *read* `width` and `height`, not to write it. This does not match the interface. A simple solution would be to modify the `picture` interface, for example so that it reads `width as size`. With this interface change, the code above will correctly allow us to find an implementation for expressions such as `P.width`.

4.8.6. Attributes implementation

If, however, we still want to be able to *write* into `width` and `height`, the implementation must provide a way to assign to the field. The first problem here is with the notation that properly identifies this operation. The following for example does not work:

```
picture is type picture
    bits:bitmap

    width  is bits.width
    height is bits.height

    width  := W is bits.width := W
    height := H is bits.width := H
```

The reason it does not work is that `width` in the `width := W` pattern is a formal parameter. The metabox solution does not work in that context, since writing `[[width]] := W` would match the *value* of `width`, not its shape, and unless `width` is defined as `self` like `true` or `false` can be, attempting to evaluate `width` will not achieve the desired effect.

It is possible to resort to a more convoluted solution that intercepts assignments to `width` and height by using `type(width)` as a way to only match `width`, which leads to somewhat inelegant and unreadable code like:

```
picture is type picture
    bits:bitmap

    width  is bits.width
    height is bits.height

    width:type(width)   := W    is      bits.width := W
    height:type(height) := H    is      bits.width := H
```

To avoid this issue, XL provides a helper generic type called `attribute T`, which helps implementing an *attribute* of the desired type. An `attribute T` behaves like a `T` that provides `get` and `set` features, as well as support for assignments. The implementation for the `picture` type type can be correctly written as follows:

```
picture is type picture
    bits:bitmap

    width as attribute[size] is attribute
        get     is bits.width
        set W   is bits.width := W
    height as attribute[size] is attribute
        get     is bits.height
        set H   is bits.height := H
```

If `P` is a `picture`, then the `width` value can be read using `P.width` or `P.width.get`, and written to using `P.width := W`, `P.width.set W` or `P.width W`. The best method will depend on the use case. In any case, an `attribute T` can be used to implement a field of type `T` declared in the interface.

4.8.7. Generic implementations

By providing as little type information as possible, the code as written above remains as generic as possible. This enables better optimizations. For instance, writing `width 320` might generate only code for `u16` without any need for any `size` value to be ever created.

Unfortunately, that code will only be accepted by the compiler until you try to use it with values that do not fit within an `u16`. It is accepted because it is not typed, therefore generic, so that some errors cannot be detected until you instantiate it. Furthermore, no error will be generated if the values being passed all fit within an `u16`.

If `P` is a `picture`, then `P.width 320` will work, but `P.width 1_000_000` will not, since that value is out of range for `u16`. This is problematic because it is quite likely that the attribute will receive some unknown `size` value. After all, the interface was designed for an `sie`, not an `u16`, so it makes sense to invoke it with values of this type. There will be an error in that case, because nothing in our code accepts a `size` value that does not fit in the `u16` subtype.

4.8.8. Attribute error checking

This can also be fixed, but if we want to do it correctly, we need to range-check the input. One cheap and lazy way to do it is to ignore bad input and just print some run-time error.

```
picture is type picture
    bits:bitmap

    width as attribute[size] is
        get             is bits.width
        set W:u16       is bits.width := W
        set W           is
             print error("Invalid picture width %1", W
             bits.width
    height as attribute[size] is
        get             is bits.height
        set H:u16       is bits.height := H
        set H           is
            print error("Invalid picture height %1", H)
            bits.height
```

If we instead want to return the error, then we need to expose the error in the interface, for example:

```
picture as type with
    width  : mayfail size
    height : mayfail size
    pixels : buffer[area] of byte
    area as size
```

That, however, suggests that `width` might accept `error` values as input. A better interface would be to expose the attribute nature of `width` and `height`:

```
picture as type with
    width  as attribute[size]
    height as attribute[size]
    pixels : buffer[area] of byte
    area as size
```

But if we are going that way, we may as well expose the simpler interface that does not lie about the underlying values:

```
picture as type with
    width  : u16
    height : u16
    pixels : buffer[area] of byte
    area as size is width * height
```

These various examples show that XL provides powerful tools that makes it possible to evolve software significantly without having to change the interface, preserving compatibility for client code. However, information hiding can never be perfect. The XL type system will catch a large number of errors and force you to deal with them, giving you several ways to safely evolve the code.

4.8.9. Exposed details

A similar, if slightly more complicated problem arises with the proposed interface for `pixels` in the `picture` type, because `buffer` is a type that may require dynamic allocation, and therefore some operations may have to return a `storage_error`. This is already present in the interface for the `buffer` type, so one might feel a bit safer than for the mismatch between `size` and `u16`.

However, since there is no actual `buffer` present in `bitmap`, a buffer may need to be created or at least given storage in the implementation of `P.pixels` for `picture` values. This may involve the *creation* of a buffer while simply reading a field. In other words, a `storage_error` may now result from apparently *reading* `P.pixels`, something that would not happen with a direct implementation of `P.pixels` as a field.

In other words, in an ideal world, `P.pixels` might be implemented as an actual buffer or as a function computing a buffer and returning it. In the real world, however, the XL type system will force you to distinguish between the two cases, because the place where dynamic allocations may fail is different depending on the chosen implementation.

In that case too, the interface may need to be changed to expose an implementation detail regarding when the buffer is actually created. It might seem like an inability of XL to offer sufficient information hiding capabilities, but in reality, it's a testament to the power of the XL type system that it should catch such subtle errors, even if it is at the cost of convenience. XL will accept to hide details as long as these details are not critical for safe evaluation. The details that will not be hidden in that case are what might cause your program to crash or your rocket to explode.

4.9. Transfers

One of the most important operation that can happen to values of any type is to *transfer* them around in the program. These operations are so crucial to the behavior of the program that XL provides a number of ways to define and optimize them. In particular, transfers interact with the ownership guarantees that the language may provide, in combination with the rules about lifetime, creation and destruction. Additionally, they are so frequent that it is necessary to consider performance of transfers, which is one reason why there are two flavors, copy and move.

4.9.1. Assignments

As was already discussed, an *assignment* is the primary way for a programmer to explicitly transfer values from one context to another. The `:=` operator is used in XL to represent assignments.

For example, the following code implements the core computation of a Julia set:

```
julia_depth(Z:complex, Mu:complex, Bound:real, Max:count) as count is
    while result < Max and Z.Re^2 + Z.Im^2 < Bound ^2 loop
        result := result + 1
        Z := Z^2 - Mu
```

This code contains two assignments, one to `result`, the implicit variable holding the return value, and one to `Z`. The assignment `result := result + 1` updates the variable `result` with the next `count` value. This kind of assignment combined with a simple operation is so frequent that there are shortcut notations for it, and you could write `result += 1` to achieve the same effect.

An assignment is a shortcut for either a copy of the value, which is the case in the code above, or a move when a copy would be unreasonably expensive. In general, using the assignment operator is the safe choice, since it will automatically select the most efficient operation for you based on the type.

When you create a new type, you get to choose if the assignment operator for that type gets to perform a copy or a move. For example, for `complex` we may want a copy, and for `picture` we may want a move because there is a lot of data. This would be implemented as follows:

```
Target:out complex := Source:complex    is Target :+ Source     // Copy
Target:out picture := Source:picture    is Target :< Source     // Move
```

4.9.2. Copy

A *copy* is a kind of transfer which creates a new, "identical", yet independent value. The new value can then be modified independently from the original, and has a different lifetime. XL uses either the prefix `copy` function or the infix `Target :+ Source` as a notation for a copy, the `+` in that operator being a reminder that a new copy is created.

The copy operation has the following mandatory interface for all types:

```
Target:out T :+ Source:T as mayfail T
copy Source:T as mayfail T is result :+ Source
```

The copy returns the `Target` after it has been copied into, or an `error` value if there was some failure. When it returns a failure, any value that may have been created as part of the copy must have been deleted by the copy operation.

The new copied value is created by the operator, and should be identical in its behavior to the source. It needs not be identical in its internal representation. For example, if you copy a `picture` type, the content of the `pixels` buffer, i.e. the values that it contains, should be identical to the original, but the binary representation for the `pixels` field itself will be different, since it will refer to a new buffer in memory.

Some types may have a complex hierarchical structure, with several layers of values referencing one another. For example, a `tree` structure may have some arbitrarily nested branches. In such a case, the copy operator should perform what is known as a *deep copy*, in other words make sure that the new value has ownership of all the elements it refers to, independently from the source value.

In some cases, it is important to also feature a *shallow copy* operation, which only copies up to a certain point. Since the condition for stopping the copy are highly dependent on the type, it is not possible to provide a single unified interface that would work for all types. However, as far as possible, such operation should have an interface that looks like the following:

```
copy (Source:T, StopConditions) as mayfail T
```

For example, the shallow copy for a tree could have the following interface, where either a depth or a `node_filter` :

```
copy(Source:tree, Depth:count) as mayfail T
copy(Source:tree, Keep:node_filter) as mayfail T
node_filter is type(N:node as boolean)
```

4.9.3. Move

A *move* is a kind of destructive transfer which gives a value to a new location, and destroys the original value. The intent is that a move is generally cheaper to perform than a copy. XL uses the prefix `move` function or the infix `Target :< Source` as a notation for the move, where the `:<` is designed to look like a scissor and is pronounced as *cuts*.

The performance benefit of using a move rather than a copy can be significant. For example, even for a complex, deep data structure, a move may involve simply copying a pointer to some new location and making sure that the original pointer is no longer used, whereas a copy may require a number of memory allocations and memory copies. The downside, obviously, is that the source of the move may no longer be used after the move.

The move operation has the following mandatory interface for all types:

```
Target:out T :< Source:in_out T as mayfail T
move Source:T as mayfail T is result :< Source
```

The move returns the `Target` after it has been moved into, or an `error` value if there was some failure. When there is a failure, the move should essentially have had no effect, i.e. it should not cause partially moved, partially created or partially destroyed values.

The new moved value is created by the operator, and should be identical in its behavior to the source. Like for copy, it needs not be identical in its internal representation. For example, if you move a value with internal pointers, it may need to adjust the pointers. Such cases should be infrequent, and are undesirable since the main reason for `move` to exist is performance.

The source of a move may no longer be used after a move operation, and should be in the same state as right after a `delete` operation. The compiler should normally issue a diagnostic when an attempt is made to use a value that was moved or deleted.

4.9.4. Binding

The transfer of an argument to a formal parameter during a call is called *binding*, and like an assignment, it may involve either copying or moving the value. Argument binding uses the assignment operator to assign the argument to the formal paramerer, ensuring that the same copy or move semantics applies to assignment and to binding.

However, the precise transfer operations associated to binding may be modified depending on the direction of the transfer between the caller and the callee. The following type modifiers are provided to indicate this direction and optimize argument passing accordingly:

- `in T` indicates that the corresponding value is passed from the caller to the callee. An `in T` value is read-only in the callee, since it is possible that it may be passed by reference. In other words, `in T` derives from `constant T`. A parameter with the `in` type modifier is called an *input parameter* and the corresponding argument is called an *input argument*. The lifetime of an input argument must be larger than that of the call, and ownership remains in the caller.
- `out T` indicates that the corresponding value is passed from the callee to the caller. In that case, the value is created in the callee, and moved to the caller on exit, destroying the previous value that may have existed for that argument in the caller. An `out T` inherits from `T`. A parameter with the `out` type modifier is called an *output parameter* and the corresponding argument is called an *output argument*.
- `inout T`, which can also be spelled `in_out T`, `in out T` or `io T`, indicates that the corresponding value is tranferred to the callee for the duration of the call, and then transferred back to the caller. Its lifetime is interrupted in the caller for the duration of the call, and ownership is temporarily transferred to the callee. The `inout T` type also inherits from `T`. A parameter with the `inout` modifier is called a *bidirectional parameter* and the corresponding argument is called a *bidirectional argument*.

All these types are intended to provide access to a value of type `T` within the caller in a way that is possibly less expensive than directly using the assignment for `T`. Common optimization strategies include:

- Passing a pointer to large values rather than copying them, a technique often called passing the value *by reference*.
- Copying or moving the value only in the required direction.
- Passing a smart reference that precisely tracks ownership.

An interesting example that illustrates the use of these type modifiers is the `text` type. This is an example of owning type, since it owns the `character` values in it. Since there is a possibly large number of characters, it may be quite expensive to copy.

It would be inconveient to have pure move semantics for `text`, since it would mean that an assignment would destroy the source value. Consider the following code to see how this would be uncomfortable:

```
window_name  : text := "Untitled"
if file_name <> "" then
    window_name := file_name
path_is_absolute : boolean := file_name.begins_with("/")
```

Under a pure move semantics, in the above code, the second use of `file_name` to compute `path_is_absolute` would be incorrect, since the value in `file_name` might have been moved to `window_name` in just the previous line. That would make using `text` quite complicated. Indeed, Rust developers have to pay attention to this kind of considerations with many types such as `vector`.

Instead, assignment for `text`, and therefore passing `text` values as an argument, performs a copy of the `text` value. The binding modifiers for `text`, however, use not `text` but `slice of text`, an access type that is cheap to copy around, and that references the original text. This organization provides the convenience of copy semantics for local variables and the speed of move semantics for calls.

4.9.5. Name parameters

In some cases, it is interesting for a form to create new variables based on a name given as an argument. The corresponding parameter is called a *name parameter*. An archetypcal example for this is the `for` loop.

Consider the following simple example:

```
for I in 1..5 loop
    print "I=", I
```

The variable `I` in this example does not exist outside of the `for` loop. This is accomplished by taking a `name` as a parameter. The metabox notation can then be used to refer to the actual name.

For example, a `for` loop on a range of discrete values can be written as follows:

```
for N:name in R:[range of discrete] loop Body is ①
    loop_context is ②
        [[N]] : R.type := R.first ③
    LoopVar is loop_context.[[N]] ④
    while LoopVar <= R.last loop
        (loop_context) (Body) ⑤
        ++LoopVar ⑥
```

① `N` is declared as a `name`. This type is defined in `XL.PARSER`. The programmer can control what kind of input is acceptable to a new programming construct like the `for` loop defined here. The fact that the XL type system is based on the shape of parse trees gives a powerful way to achieve that objective.

② We need to create a `loop_context` scope that holds the variable created from the input argument to make sure that it does not pollute the current context of the function. For example, if the argument for `N` was `R`, we would not want to hide the `R` formal parameter with a local definition that has the same name.

③ The metabox evaluates `N`, which gives us access to the name referenced by `N`. In the declaration, simply writing `N` would create a local variable named `N`.

④ The `LoopVar` declaration is a shortcut to facilitate access to within `loop_context`. The reference to `N` also needs to go through a metabox in order to lookup what `N` contains, and not the variable `N` within loop_context, does not exist except in cases where parameter N receives name N as an argument.

⑤ The parentheses are not strictly necessary, but a good reminder that what we are doing here is evaluate Body after injecting the loop_context context. This is what makes the variable name referenced by visible in Body. This approach lets the programmer precisely control what is being injected while evaluating Body. In particular, all the local variables in the implementation of the for loop, like N, R, Body, LoopVar or loop_context are not visible while evaluating Body.

⑥ The ++LoopVar notation is a generic way to increment any discrete value. In that case, LoopVar += 1 would not work if R was for example a range of character like 'A'..'Z'.

4.9.6. Attributes

Attributes are a types that behave like a value of a given type, but with controlled access when reading or writing values. Attributes also offer the convention that they can be written to by using them as a prefix.

For example, a background feature with the color type can be implemented as an attribute, which can then be used in any of the following ways:

```
background : attribute[color]
if background = blue then
    background := red           // Assigment style
if background = green then
    background blue             // Attribute style
```

It is considered bad taste in XL to use explicitly getters or setters as is common usage in languages such as Java. For example, the following code is a very poor alternative to attributes:

```
get_background as color
set_background C:color as color
if get_background = blue then
   set_background red
```

Chapter 5. Programming paradigms

The XL language features make it quite easy to follow and even enforce the rules necessary to apply various common and less common *programming paradigms*.

In the following sections, we will consider the following major paradigms:

- Object-oriented programming
- Functional programming
- Generic programming
- Design by contract
- Distributed programming

We will also take a look at a few less common, if highly esteemed, programming paradigms, which all carved a niche through somewhat special-purpose programming languages:

- Aspect-oriented programming
- Logic programming
- Declarative programming
- Reactive programming
- Synchronous programming

The fact that XL seamlessly supports so many different programming paradigms *together* is a testimony to the extensible nature of the language.

5.1. Object-oriented programming

Object-oriented programming is a programming paradigm based on the concept of *objects*, which are self-contained units acting as black boxes, providing a controlled interface to their internal *data* through *methods*. This provides a form of *information hiding* that helps with the long-term maintenance of the software by enforcing rules about how to access the object. Object-oriented programming relies on a number of features such as *encapsulation dynamic dispatch*, (sometimes called *message passing*), *inheritance*, and *polymorphism*.

A common way to introduce object-oriented programming is with a program that deals with various shapes. This is a good way to illustrate the use of the fundamental object-oriented techniques. In the following example, we will write such a program, which will both illustrate the similarities of XL with a language like C++, as well as the differences and how they can matter even on such a simple example.

5.1.1. Type interface

The first part of the program is to define the interface for the type `shape`, which will act as the *base class* of our class hierarchy, and for the derived types that we will use in our code. We will assume that we have an existing `coordinate` type for shape coordinates, `dimension` for shape dimensions, and `color` for shape colors.

A first attempt at building a type interface for `shape` and a couple of derived types using the syntax for types we already saw would be:

```
// Base class
shape as type with
    draw         as nil                          // Draw the shape
    fill_color   as attribute color              // Color of the fill
    stroke_color as attribute color              // Color of the stroke
    stroke_width as attribute dimension          // Width of the stroke
    top          as attribute coordinate         // Top coordinate
    left         as attribute coordinate         // Left coordinate
    bottom       as attribute coordinate         // Bottom coordinate
    right        as attribute coordinate         // Right coordinate
    x            as attribute coordinate         // Center horizontal coordinate
    y            as attribute coordinate         // Center vertical coordinate
    width        as attribute dimension          // Width of the shape
    height       as attribute dimension          // Height of the shape

// A few derived classes
rectangle as type like shape

square as type like rectangle with
    side        as attribute dimension

ellipse as type like shape

circle as type like shape with
    radius      as attribute dimension
    diameter    as attribute dimension
```

This style of code is called a *type interface*. It already shows interesting properties that XL brings to the table. From the interface, we see that we have multiple ways to access the same data, namely the dimensions and position of the shape, which are obviously related. The width of the shape is related to the left and right coordinates, for example. However, the interface does not tell us how the implementation choses to store the data. This provides stronger information hiding than languages like C++, where some "private" data fields for the class would probably be exposed.

This means that client code can access all these features in a very consistent and flexible manner, without bothering about how the data is stored internally:

```
S : shape
S.top := 42                     // Assignment form
S.bottom 640                    // Attribute setting form
if S.height <> 598 then         // Reading the value
    logic_error "Somethign is wrong with your implementation, height is %1", S.height
```

As part of information hiding, the interface also purposefully does not expose how `draw` is actually implemented. The `draw` method, obviously, needs a different implementation for a `rectangle` and a `circle`, but the interface does not expose that information, only the fact that there is a `draw` method in `shape` and all derived types.

5.1.2. Class interface

However, our first interface remains somewhat verbose and cumbersome to write. In order to support object-oriented programming better, there is a `class` declaration helper that will generate the above code based on the following input, transforming all fields into attributes and keeping methods as is. This coding style is called a *class interface*:

```
class shape with
    draw        as nil
    fill_color   : color
    stroke_color : color
    stroke_width : dimension
    top          : coordinate
    left         : coordinate
    bottom       : coordinate
    right        : coordinate
    x            : coordinate
    y            : coordinate
    width        : dimension
    height       : dimension

class rectangle like shape

class square like rectangle with
    side         : dimension

class ellipse like shape

class circle like shape with
    radius       : dimension
    diameter     : dimension
```

5.1.3. Class implementation

In XL, the implementation of a type can be directly related to the interface, but every field may also be implemented indirectly using attributes.

For the shape class, let's make the choice that we will store the center of the shape, its width and its height, and compute the rest. We need a convention on what happens when we set `top`, for example. A convention will be that it moves the whole shape without changing its size, but we could also decide to not move the bottom for example. We will also assume that `y` increases towards the top. This particular choice leads to a class implementation that looks like:

```
class shape is
    draw is logic_error "Drawing a base shape"
    fill_color   : color
    stroke_color : color
    stroke_width : dimension
    x            : coordinate
    y            : coordinate
    width        : dimension
    height       : dimension

    left as attribute coordinate is
        get     is x - width / 2
        set L   is x += L - left
    right as attribute coordinate is
        get     is x + width / 2
        set R   is x += R - right
    top as attribute coordinate is
        get     is y + height / 2
        set T   is y += T - top
    bottom as attribute coordinate is
        get     is y - height / 2
        set B   is y += B - bottom
```

This kind implementations demonstrates how much freedom there is in implementing a class interface.

5.1.4. Direct derivation

A derived class like `rectangle` can leverage the base class using data inheritance, and then add a `draw` method. This kind of derivation is called *direct derivation*.

```
class rectangle is shape with
    draw is
        draw_polygon fill_color, stroke_color, stroke_width,
            move_to top, left
            line_to top, right
            line_to bottom, right
            line_to bottom, left
            line_to top, left
```

Since the purpose of this document is not to focus on shape drawing algorithms, but on object-oriented programming, we simply assumed that there is some general `draw_polygon` utility that we can use.

Unlike languages like C++, XL does not require the interface of a derived class like `rectangle` to indicate that it will override the `draw` feature from the base class.

5.1.5. Indirect derivation

Implementing the square type allows us to demonstrate an interesting possibility of XL that does not exist in C++, which is to make a derived class that does not inherit its data members from its base class. A square does not need a separate width and height, so we could make the data storage requirements for a square smaller by only having it store a single side data value. The implementation of square can however still inherit a lot of the implementation from rectangle, but needs to rewrite width and height as attributes related to side:

```
class square is
    draw is rectangle.draw
    fill_color   : color
    stroke_color : color
    stroke_width : dimension
    x            : coordinate
    y            : coordinate
    side         : dimension

    width as attribute dimension is
        get     is side
        set W   is side := W
    height as attribute dimension is
        get     is side
        set H   is side := H

    left        is rectangle.left
    right       is rectangle.right
    top         is rectangle.top
    bottom      is rectangle.left
```

Obviously, the savings in the case of something as simple as a square are quite limited. There are cases where this approach may lead to much more significant improvements.

More importantly, this approach makes it possible to enforce stricter rules for the derived types. In particular, this implementation of square makes it absolutely impossible to end up with a square that has a different value for width and height, even by mistake.

This is not a purely theoretical concern. In C++, if the base Rectangle class has different fields for width and height, the derived Square class might initially enforce the rules, and be later broken for example when the base Shape class adds a scale method that scales width and height differently. Unless the Square class itself is modified to implement its own version of scale, it will be broken by a change in the base class. This category of issue is known as the fragile base class problem.

We can also take advantage of this feature to have a different hierarchy for data inheritance than for the interface. For example, we can implement the ellipse and circle classes as follows:

```
class ellipse is rectangle with
    draw is
        draw_ellipse fill_color, stroke_color, stroke_width, x, y, width, height

class circle is square with
    draw is ellipse.draw
```

This flexibility gives the maximum potential for code reuse.

5.1.6. Dynamic dispatch

If we want to add a `group` shape, we need to be able to draw each shape individually, according to its class.

```
class group like shape with
    children : string of shape
```

There is a problem with this interface, however. Each time you add a `shape` to `children`, you really add a value that has the `shape` type, not the `rectangle` or `circle` type. In order to preserve the original type information, we need to use the `any shape` notation. The correct class interface for `group` is:

```
class group like shape with
    children : string of any shape
```

The implementation can then invoke `draw` for each shape in `children`, and since `any shape` retains the type of the associated shape, this will call the appropriate `draw` feature for each of the individual shapes. This technique is called *dynamic dispatch*.

The implementation of class `group` does not need the `shape` fields, since it can compute features from the shapes in `children`. The example below only shows the case of `draw`, `fill_color`, `top` and `width`, but other features can be implemented in a similar way. The `fill_color` attribute uses a local variable to hold the color value for the whole group. These three examples show the respective benefits of the various ways to set an attribute.

```
class group is
    // Draw a group by drawing all shapes in it
    draw is
        for S in children loop
            S.draw

    // The fill color is the last one set for the group
    fill_color as attribute color is
        value : color := black
        get   is value
        set C is
            value := C
            for S in children loop
                S.fill_color C

    // The top is the maximum of the top of all shapes
    top as attribute coordinate is
        get is
            result := coordinate.min
            for S in children loop
                if result < S.top then
                   result := S.top
        set T is
            delta is T - top
            for S in children loop
                S.top += delta

    // The width is still right - left, and adjusted by scaling
    width as attribute coordinate is
        get   is right - left
        set W is
            old is width
            for S in children loop
                S.width := S.width * W / old
```

In this code, all references to `S` have the type `any shape`, and therefore, `S.width` or `S.draw` will be dynamically dispatched. This plays the role to *virtual functions* in C++.

5.1.7. Multiple dispatch

An operation like `draw` really depends on a single `shape` value. Many operations, however, require more than one value to operate. A simple example would be an operation to intersect two shapes, which we could write as `S1 and S2`, or merge two shapes, which we could write as `A or B`. There are various special cases, followed by a more general case:

```
A:rectangle and B:rectangle as rectangle
A:ellipse   and B:ellipse   as ellipse or path
A:rectangle and B:ellipse   as rectangle or ellipse or path
A:ellipse   and B:rectangle as rectangle or ellipse or path
A:group     and B:group     as group
A:any shape and B:any shape as any shape
```

The implementation for groups, for example, could look someting like the following code, which intersects all pairs of shapes and adds the result when it is not empty:

```
A:group and B:group as group is
    for SA in A.children loop
        for SB in B.children loop
            child is SA and SB
            if child.width > 0 and child.height > 0 then
                result.children &= child
```

The operation that computes `child` in the code above is `SA and SB`. Since both `SA` and `SB` are dynamically typed, that operation must perform a dynamic dispatch on both `SA` and `SB`. A feature like tihs is sometimes called *multiple dispatch* or *multi-methods* in other languages. For example, if both shapes have the `rectangle` type, then the first declaration would be called; if both shapes have the `ellipse` type, then the second one would be invoked; and so on.

With code like this, it is possible to write a function that performs clipping within a rectangle, where dynamic dispatch will occur only for the second argument, and where the returned value will itself be dynamically typed:

```
clip(Bounds:rectangle, Shape:any shape) as any shape is
    result := Bounds and Shape
```

5.2. Functional programming

This section will soon cover techniques that are familiar to programmers using languages derived from Lisp and similar languages:

- Functions as first-order values
- Anonymous functions, already quickly covered earlier
- Currying
- Purely-functional code
- Immutable values

5.3. Generic programming

This section will soon cover techniques that are familar primarily to C++ programmers:

- Generic containers
- Generic algorithms
- Traits
- Concepts

5.4. Design by contract

This section will soon cover techniques that are familiar primarily to Eiffel developers:

- Invariants
- Preconditions
- Postconditions

5.5. Distributed programming

This section will cover techniques used for distributed programming, that come from a variety of sources, and includes some innovations that I believe are specific to the XL family of languages:

- Processes
- Threads
- Tasks
- Thread pools
- Erlang-style message passing
- Ada-style rendez-vous
- ELFE-style distributed programs

5.6. Aspect-oriented programming

This section will cover techniques to address cross-cutting concerns that were studied in languages such as AspectJ, as well as other aspects that were explored in XL2:

- AspectJ-style code injection
- Automated trace / logging injection
- Translation statemnts in XL2

5.7. Logic programming

This section will cover techniques developed in particular by the Prolog programming language.

5.8. Declarative programming

This section will cover techniques that make programs look or behave more declarative in a declarative than imperative way. It will in particular cover more extensively some declarative aspects of Tao3D, and how to use XL as a document description language.

- Non-imperative evaluation
- Documentation generation

5.9. Reactive programming

This section will cover techniques that were in particular developed for Tao3D:

- Partial re-evaluation
- Value dependency tracking
- Data flow programming

5.10. Synchronous programming

This section will cover concepts and techniques that were particular notably by the Esterel family of languages:

- Signals
- Model of time
- Determinism

Chapter 6. Compiling XL

The XL language is quite different from a language like C. Some of the language constructs described in this document may seem a little bit mysterious to programmers coming from such languages. This section will explain how some of the features described previously can be implemented, by showing examples using C code.

In the C code, the notation `` `expr` `` will denote a magical macro creating a unique name attributed to the XL expression `expr`, and `⋯` will indicate that what is presented is only a partial representation, with the expectation that whas surrounds `⋯` allows the reader to infer what can be there.

In addition, the example code will borrow one feature from C++, inheritance, so that we may write something like the following in the examples:

```
struct Derived : Base { ... };
```

6.1. Normal representation

All XL source code and data can be represented using a homoiconic representation that looks something like the following code, where `P` is a shortcut for `XL.PARSER`.

First, there is a tag that can be used at run-time to discriminate between the possible values, and a struct that holds the common fields, notably a `position` field that can be used to identify a precise source code position while printing error messages:

```
enum kind { NATURAL, REAL, CHARACTER, ... };
struct `P.tree`                        { enum kind kind; position_t position; ... };
```

The leaf nodes simpy contain one of the basic values tpyes that can be foudn in the source code:

```
struct `P.natural`    : `P.tree`    { `natural` value; };
struct `P.real`       : `P.tree`    { `real` value; }
struct `P.character`  : `P.tree`    { `character` value; };
struct `P.text`       : `P.tree`    { `text` value; }
struct `P.bits`       : `P.tree`    { size_t size; void *data; }
struct `P.symbol`     : `P.tree`    { `text` value; }
struct `P.name`       : `P.symbol`  { };
struct `P.operator`   : `P.symbol`  { };
```

The inner nodes contain pointers to other nodes. These pointers are never `NULL`, and the structure is a tree that is easily allocated or deallocated. The previous XL implementations have pools for each type, allowing faster fixed-size allocation without memory fragmentation.

```
struct `P.prefix`     : `P.tree`    { `P.tree` *left, *right; };
struct `P.postfix`    : `P.tree`    { `P.tree` *left, *right; };
struct `P.infix`      : `P.tree`    { `P.tree` *left, *right; `text` name; }
struct `P.block`      : `P.tree`    { `P.tree` *child; `text` opening, closing; }
```

This representation is called the *normal representation* or *normal form* of the parse tree, and it corresponds almost directly to the XL declarations given in module `XL.PARSER`. This is in particular the input to the `XL.TRANSLATOR` module, and it can be evaluated directly.

6.2. Machine representation

The normal form is not very efficient, and generated machine code will normally only use a *machine representation* or *machine form* for both data and code.

When this is useful for proper evaluation, a normal form can be *boxed* into its equivalent machine representation, and conversely, that machine representation can be *unboxed*. This process is called *autoboxing*, and should be entirely transparent to the developer.

The machine representation for data is normally a `struct` that contains the definitions in the pattern matching scope for the type being considered. For example:

```
// complex is type complex(Re:real, Im:real)
typedef struct `complex` { `real` Re; `real` Im; } complex_t;

// Z : complex := complex(1.2, 3.4)
complex_t Z = { 1.2, 3.4 };

// Z1:complex + Z2.complex is complex(Z1.Re+Z2.Re, Z1.Im+Z2.Im)
#define complex_add  `Z1:complex + Z2.complex`
complex_t complex_add(complex_t Z1, complex_t Z2)
{
    complex_t result;
    result.Re = Z1.Re + Z2.Re;
    result.Im = Z1.Im + Z2.Im;
    return result;
}
```

In the special case of types that contain only a single value, that machine type itsef is used, since the XL compiler takes care of the type checking aspects. In other words, there is no additional cost in using this kind of type.

```
// distance is type distance(meters:real)
typedef `real` distance_t;    // Probably a double

// D1:distance + D2:distance is distance(D1.meters + D2.meters)
#define distance_add `D1:distance + D2:distance`
inline distance_t distance_add(distance_t D1, distance_t D2)
{
    return D1 + D2;
}
```

At lower levels of optimization, as was the case in Tao3D, the machine representation for types with open-ended or unconstrained values will typically contain the normal form, knowing that the passed normal form will generally be the normal form for closures:

```
// if_statement is type { if Condition then TrueClause else FalseClause }
struct `if_statement`
{
    `P.tree` *Condition;
    `P.tree` *TrueClause;
    `P.tree` *FalseClause;
};
```

6.3. Closures representation

Executable code and closures also have machine forms. The machine form for a closure may look something like:

```
typedef `type of expression` expression_type;
struct `expression`
{
    expression_type (*code) (struct `expression` *closure, args...);
    `P.tree` *self; // If needed
    ... // additional data fields for captured data used by 'code'
};
```

For example, consider the `adder` example given as an example for closures earlier, assuming we know that `N` and `X` are `integer` values.

A first structure represents a function that takes a single `X` argument, again assuming that argument is an `int`:

```
// adder N is { lambda X is X + N }
typedef struct `lambda X`
{
    int (*code) (struct `lambda X` *closure, int X);
} lambda;
```

However, if the code uses a variable like `N`, that variable is said to be *captured*. In order to be able to preserve the value of that variable outside of the scope that created it, another structure inheriting from `lambda` is needed:

```
typedef struct `lambda X capturing N` : lambda
{
    int N;
} lambda_N;
```

Using this data structure, it is now possible to generate the code for the body of the anonymous function:

```
#define anonymous_function `X + N`
int anonymous_function(lambda_N *closure, int X)
{
    return X + closure->N;
}
```

That body can then be used to generate the code for the `adder` function itself:

```
#define adder_name      `adder N`
lambda_N adder_name(int N)
{
    lambda_N result;
    result.code = anonymous_function;
    result.N = N;
    return result;
}
```

That code can be used to create values that represent the adder, along with the data that it needs:

```
// add3 is adder 3
#define add3    `add3`
lambda_N add3 = adder_name(3)
```

Invoking that code is straightforward:

```
// add3 5
add3.code(&add3, 5);
```

6.4. Type representation

The `type` type is a standard data type, which exposes a number of features used by the XL runtime to manipulate values of the type:

```
struct `type`
{
    size_t (*size)    (void *value);    // Compute size of a value
    void   (*create)  (void *value);    // Create value of the type
    void   (*delete)  (void *value);    // Delete value of the type
    bool   (*contains)(void *value);    // Check if a value belong
    ...
};
```

When a type interface is provided, the actual type implementation may be implemented as a structure that derives from `struct `type``.

```
// shape_t as type with
//    draw         as nil
//    width        : coordinate

struct `shape`_interface : `type`
{
    void                (*`draw`) (`shape` *shape);
    `coordinate`        (*`width`)(`shape` *shape);
    `coordinate`        (*`width`)(`shape` *shape, `coordinate` width);
}
```

The type interface can then be used by generated code in a way that is largely independant of the actual layout for the object. The type interface therefore gives a standard way to refer to the features of the type. Since all the features are represented by pointer, derived types may populate them by alternate values. This mechanism is overall very similar to so-called *vtables* in C++.

6.5. Compiled representations

Code and any data can also have one or several *compiled forms*. The compiled forms are generally very implementation-dependent, varying with the machine you run the program on as well as with the compiler technology being used.

Types also determine properties such as the size and binary representation of values. For example, on most machines, `integer` will be represented as a 64-bit 2-complement binary value, and `real` using the IEEE-754 64-bit representation.

6.6. Data

6.7. Lifetime

6.8. Closures

6.9. Compact vs. Packed

Chapter 7. Basic operations

Chapter 8. Modules

Chapter 9. Standard Library

The standard library provides the vast majority of the features available to the XL developer.

9.1. Garbage collection

Chapter 10. History of XL

The status of the current XL compiler is a bit messy. There is a rationale to this madness. I attempt to give it here.

There is also a blog version if you prefer reading on the web (but it's not exactly identical). In both cases, the article is a bit long, but it's worth understanding how XL evolved, and why the XL compiler is still work in progress.

10.1. It started as an experimental language

Initially, XL was called LX, "Langage experimental" in French, or as you guessed it, an experimental language. Well, the very first codename for it was "WASHB" (What Ada Should Have Been). But that did not sound very nice. I started working on it in the early 1990s, after a training period working on the Alsys Ada compiler.

What did I dislike about Ada? I never liked magic in a language. To me, keywords demonstrate a weakness in the language, since they indicated something that you could not build in the library using the language itself. Ada had plenty of keywords and magic constructs. Modern XL has no keyword whatsoever, and it's a Good Thing ™.

Let me elaborate a bit on some specific frustrations with Ada:

- Tasks in Ada were built-in language constructs. This was inflexible. Developers were already hitting limits of the Ada-83 tasking model. My desire was to put any tasking facility in a library, while retaining an Ada-style syntax and semantics.
- Similarly, arrays were defined by the language. I wanted to build them (or, at least, describe their interface) using standard language features such as generics. Remember that this was years before the STL made it to C++, but I was thinking along similar lines. Use cases I had in mind included:
  - interfacing with languages that had different array layouts such as Fortran and C,
  - using an array-style interface to access on-disk records. Back then, `mmap` was unavailable on most platforms,
  - smart pointers that would also work with on-disk data structures,
  - text handling data structures (often called "strings") that did not expose the underlying implementation (e.g. "pointer to char" or "character array"), …
- Ada text I/O facilities were uncomfortable. But at that time, there was no good choice. You had to pick your poison:
  - In Pascal, `WriteLn` could take as many arguments as you needed and was type safe, but it was a magic procedure, that you could not write yourself using the standard language features, nor extend or modify to suit your needs.
  - Ada I/O functions only took one argument at a time, which made writing the simplest I/O statement quite tedious relative to C or Pascal.
  - C's `printf` statement had multiple arguments, but was neither type safe nor extensible, and

the formatting string was horrid.

- I also did not like Ada pragmas, which I found too ad-hoc, with a verbose syntax. I saw pragmas as indicative that some kind of generic "language extension" facility was needed, although it took me a while to turn that idea into a reality.

I don't have much left of that era, but that first compiler was relatively classical, generating 68K assembly language. I reached the point where the compiler could correctly compile a "Hello World" style program using an I/O library written in the language. I was doing that work at home on Atari-ST class machines, but also gave demos to my HP colleagues running XL code on VME 68030 boards.

From memory, some of the objectives of the language at the time included:

- Giving up on superfluous syntactic markers such as terminating semi-colon.
- Using generics to write standard library component such as arrays or I/O facilities.
- Making the compiler an integral part of the language, which led to…
- Having a normalised abstract syntax tree, and…
- Considering "pragmas" as a way to invoke compiler extensions. Pragmas in XL were written using the `{pragma}` notation, which would indirectly invoke some arbitrary code through a table.

Thus, via pragmas, the language became extensible. That led me to…

10.2. LX, an extensible language

I wanted to have a relatively simple way to extend the language. Hence, circa 1992, the project was renamed from "experimental" to "extensible", and it has kept that name since then.

One example of thing I wanted to be able to do was to put tasking in a library in a way that would "feel" similar to Ada tasking, with the declaration of task objects, rendez-vous points that looked like procedures with parameter passing, and so on.

I figured that my `{annotations}` would be a neat way to do this, if only I made the parse tree public, in the sense that it would become a public API. The idea was that putting `{annotation}` before a piece of code would cause the compiler to pass the parse tree to whatever function was associated with `annotation` in a table of annotation processors. That table, when pointing to procedures written in XL, would make writing new language extensions really easy. Or so I thought.

Ultimately, I would make it work. If you are curious, you can see the grand-child of that idea in the `translation` statements under `xl2/`. But that was way beyond what I had initially envisionned, and the approach in the first XL compiler did not quite work. I will explain why soon below.

The first experiment I ran with this, which became a staple of XL since then, was the `{derivation}` annotation. It should have been `{differentiation}`, but at that time, my English was quite crappy, and in French, the word for "differentiation" is "derivation". The idea is that if you prefixed some code, like a function, with a `{derivation}` annotation, the parse tree for that function would be passed to the `derivation` pragma handler, and that would replace expressions that looked like differential expressions with their expanded value. For example, `{derivation} d(X+sin(X))/dX` would generate code that looked like `1 + cos(X)`.

If you are curious what this may look like, there are still tests in the XL2 test suite using a very similar feature and syntax.

10.3. LX, meet Xroma

That initial development period for LX lasted between 1990, the year of my training period at Alsys, and 1998, when I jointed the HP California Language Lab in Cupertino (CLL). I moved to the United States to work on the HP C++ compiler and, I expected, my own programming language. That nice plan did not happen exactly as planned, though…

One of the very first things I did after arriving in the US was to translate the language name to English. So LX turned into XL. This was a massive rename in my source code, but everything else remained the same.

As soon as I joined the CLL, I started talking about my language and the ideas within. One CLL engineer who immediately "got it" is Daveed Vandevoorde. Daveed immediately understood what I was doing, in large part because he was thinkering along the same lines. He pointed out that my approach had a name: meta-programming, i.e. programs that deal with programs. I was doing meta-programming without knowing about the word, and I felt really stupid at the time, convinced that everybody in the compilers community knew about that technique but me.

Daveed was very excited about my work, because he was himself working on his own pet language named Xroma (pronounced like Chroma). At the time, Xroma was, I believe, not as far along as XL, since Daveed had not really worked on a compiler. However, it had annotations similar to my pragmas, and some kind of public representation for the abstract syntax tree as well.

Also, the Xroma name was quite Xool, along with all the puns we could build using a capital-X pronounced as "K" (Xolor, Xameleon, Xode, …) or not (Xform, Xelerate, …) As a side note, I later called "Xmogrification" the VM context switch in HPVM, probably in part as a residual effect of the Xroma naming conventions.

In any case, Daveed and I joined forces. The combined effort was named Xroma. I came up with the early version of the lightbulb logo still currently used for XL, using FrameMaker drawing tools, of all things. Daveed later did a nice 3D rendering of the same using the Persistence of Vision ray tracer. I don’t recall when the current logo was redesigned.

10.4. XL moves to the off-side rule

Another major visual change that happened around that time was switching to the off-side rule, i.e. using indentation to mark the syntax. Python, which made this approach popular, was at the time a really young language (release 1.0 was in early 1994).

Alain Miniussi, who made a brief stint at the CLL, convinced me to give up the Ada-style `begin` and `end` keywords, using an solid argumentation that was more or less along the lines of "I like your language, but there’s no way I will use a language with `begin` and `end` ever again". Those were the times where many had lived the transition of Pascal to C, some still wondering how C won.

I was initially quite skeptical, and reluctantly tried an indentation-based syntax on a fragment of the XL standard library. As soon as I tried it, however, the benefits immediately became apparent. It was totally consistent with a core tenet of concept programming that I was in the process of developing (see below), namely that the code should look like your concepts. Enforcing indentation made sure that the code did look like what it meant.

It took some effort to convert existing code, but I've never looked back since then. Based on the time when Alain Miniussi was at the CLL, I believe this happened around 1999.

10.5. Concept programming

The discussions around our respective languages, including the meta-programming egg-face moment, led me to solidify the theoretical underpinning of what I was doing with XL. My ideas actually did go quite a bit beyond mere meta-programming, which was really only a technique being used, but not the end goal. I called my approach *Concept Programming*. I tried to explain what it is about in this presentation. Concept programming is the theoretical foundation for XL.

Concept programming deals with the way we transform concepts that reside in our brain into code that resides in the computer. That conversion is lossy, and concept programming explores various techniques to limit the losses. It introduces pseudo-metrics inspired by signal processing such as syntactic noise, semantic noise, bandwidth and signal/noise ratio. These tools, as simple as they were, powerfully demonstrated limitations of existing languages and techniques.

Since then, Concept Programming has consistently guided what I am doing with XL. Note that Concept Programming in the XL sense has little do do with C++ concepts (although there may be a connection, see blog referenced above for details).

10.6. Mozart and Moka: Adding Java support to XL

At the time, Java was all the rage, and dealing with multiple languages within a compiler was seen as a good idea. GCC being renamed from "GNU C Compiler" to the "GNU Compiler Collection" is an example of this trend.

So with Daveed, we had started working on what we called a "universal program database", which was basically a way to store and access program data independently of the language being used. In other words, we were trying to create an API that would make it possible to manipulate programs in a portable way, whether the program was written in C, C++ or Java. That proved somewhat complicated in practice.

Worse, Daveed Vandevoord left the HP CLL to join the Edison Design Group, where he's still working to this date. Xroma instantly lost quite a bit of traction within the CLL. Also, Daveed wanted to keep the Xroma name for his own experiments. So we agreed to rename "my" side of the project as "Mozart". For various reasons, including a debate regarding ownership of the XL code under California law, the project was open-sourced. The web site still exists to this day, but is not quite functional since CVS support was de-commissioned from SourceForge.

Part of the work was to define a complete description of the source code that could be used for different language. Like for Xroma, we stayed on bad puns and convoluted ideas for naming. In Mozart that representation was called `Coda`. It included individual source elements called `Notes` and the serialized representation was called a `Tune`. Transformation on Notes, i.e. the operations of compiler plug-ins, were done by `Performer` instances. A couple of years later, I would realize that this made the code totally obfuscated for the non-initiated, and I vowed to never make that mistake again.

Mozart included Moka, a Java to Java compiler using Mozart as its intermediate representation. I published an article in Dr Dobb's journal, a popular developers journal at the time.

But my heart was never with Java anymore than with C++, as evidenced by the much more extensive documentation about XL on the Mozart web site. As a language, Java had very little interest for me. My management at HP had no interest in supporting my pet language, and that was one of the many reasons for me to leave the CLL to start working on virtualization and initiate what would become HPVM.

10.7. Innovations in 2000-vintage XL

By that time, XL was already quite far away from the original Ada, even if it was still a statically typed, ahead-of-time language. Here are some of the key features that went quite a bit beyond Ada:

- The syntax was quite clean, with very few unnecessary characters. There were no semi-colons at the end of statement, and parentheses were not necessary in function or procedure calls, for example. The off-side rule I talked about earlier allowed me to get rid of any `begin` or `end` keyword, without resorting to C-style curly braces to delimit blocks.
- Pragmas extended the language by invoking arbitrary compiler plug-ins. I suspect that attributes in C++11 are distant (and less powerful) descendants of this kind of annotation, if only because their syntax matches my recollection of the annotation syntax in Xroma, and because Daveed has been a regular and innovative contributor to the C++ standard for two decades…
- Expression reduction was a generalisation of operator overloading that works with expressions of any complexity, and could be used to name types. To this day, expression reduction still has no real equivalent in any other language that I know of, although expression templates can be used in C++ to achieve similar effect in a very convoluted and less powerful way. Expression templates will not allow you to add operators, for example. In other words, you can redefine what `X+Y*Z` means, but you cannot create `X in Y..Z` in C++.
- True generic types were a way to make generic programming much easier by declaring generic types that behaved like regular types. Validated generic types extended the idea by adding a validation to the type, and they also have no real equivalent in other languages that I am aware of, although C++ concepts bring a similar kind of validation to C++ templates.
- Type-safe variable argument lists made it possible to write type-safe variadic functions. They solved the `WriteLn` problem I referred to earlier, i.e. they made it possible to write a function in a library that behaved exactly like the Pascal `WriteLn`. I see them as a distant ancestor of variadic templates in C++11, although like for concepts, it is hard to tell if variadic templates are a later reinvention of the idea, or if something of my e-mails influenced members of the C++ committee.
- A powerful standard library was in the making. Not quite there yet, but the key foundations were there, and I felt it was mostly a matter of spending the time writing it. My implementation of complex numbers, for example, was 70% faster than C++ on simple examples, because it

allowed everything to be in registers instead of memory. There were a few things that I believe also date from that era, like getting rid of any trace of a main function, top-level statements being executed as in most scripting languages.

10.8. XL0 and XL2: Reinventing the parse tree

One thing did not work well with Mozart, however, and it was the parse tree representation. That representation, called `Notes`, was quite complicated. It was some kind of object-oriented representation with many classes. For example, there was a class for `IfThenElse` statements, a `Declaration` class, and so on.

This was all very complicated and fragile, and made it extremely difficult to write thin tools (i.e. compiler plug-ins acting on small sections of code), in particular thin tools that respected subtle semantic differences between languages. By 2003, I was really hitting a wall with XL development, and that was mostly because I was also trying to support the Java language which I did not like much.

One of the final nails in the Mozart coffin was a meeting with Alan Kay, of Smalltalk fame, during an HP technical conference. Kay was an HP Fellow at the time. I tried to show him how my language was solving some of the issues he had talked about during his presentation. He did not even bother looking. He simply asked: *“Does your language self-compile?“*. When I answered that the compiler was written in C++, Alan Kay replied that he was not interested.

That gave me a desire to consider a true bootstrap of XL. That meant rewriting the compiler from scratch. But at that time, I had already decided that the internal parse tree representation needed to be changed. So that became my topic of interest.

The new implementation was called XL2, not just as a version number, but because I was seeing things as a three-layer construction:

- `XL0` was just a very simple parse tree format with only eight node types. I sometimes refer to that level of XL as "*XML without the M*", i.e. an extensble language without markup.
- `XL1` was the core language evaluation rules, not taking any library into account.
- `XL2` was the full language, including its standard library. At the time, the goal was to reconstruct a language that would be as close as possible at the version of XL written using the Mozart framework.

This language is still available today, and while it’s not been maintained in quite a while, it seems to still pass most of its test suite. More importantly, the `XL0` format has remained essentially unchanged since then.

The XL0 parse tree format is something that I believe makes XL absolutely unique among high-level programming languages. It is designed so that code that can look and feel like an Ada derivative can be represented and manipulated in a very simple way, much like Lisp lists are used to represent all Lisp programs. XL0, however, is not some minor addition on top of S-expressions, but rather the definition of an alternative of S-expressions designed to match the way humans parse code.

The parse tree format consists of only eight node types, four leaf node types (integer, real, text and symbol), four inner node types (infix, prefix, postfix and block).

- `Integer` represents integer numbers, like `123` or `16#FFFF_FFFF`. As the latter example shows, the XL syntax includes support for based numbers and digit grouping.
- `Real` represents floating-point numbers, like `123.456` or `2#1.001_001#e-3`. Like for `Integer`, XL supports based floating-point numbers and digit grouping.
- `Text` represents textual constants like `"Hello"` or `'A'`.
- `Name` represents names like `ABC` or symbols like `<=`.
- `Infix` represents operations where a name is between two operands, like `A+B` or `A and B`.
- `Prefix` represents operations where an operator precedes its operand, like `sin X` or `-4`.
- `Postfix` represents operations where an operator follows its operand, like `3 km` or `5%`.
- `Block` represents operations where an operand is surrounded by two names, like `[A]`, `(3)` or `{write}`.

Individual program lines are seen as the leaves of an infix "newline" operator. There are no keywords at all, the precedence of all operators being given dynamically by a syntax file.

10.9. Bootstrapping XL

The initial translator converts a simplified form of XL into C++ using a very basic transcoding that involves practically no semantic analysis. The limited XL2 acceptable as input for this translation phase is only used in the bootstrap compiler. It already looks a bit like the final XL2, but error checking and syntax analysis are practically nonexistent.

The bootstrap compiler can then be used to translate the native XL compiler. The native compiler performs much more extended semantic checks, for example to deal with generics or to implement a true module system. It emits code using a configurable "byte-code" that is converted to a variety of runtime languages. For example, the C bytecode file will generate a C program, turning the native compiler into a transcoder from XL to C.

That native compiler can translate itself, which leads to a true bootstrap where the actual compiler is written in XL, even if a C compiler is still used for the final machine code generation. Using a Java or Ada runtime, it would theoretically be possible to use a Java or Ada compiler for final code generation.

The XL2 compiler advanced to the point where it could pass a fairly large number of complex tests, including practically all the things that I wanted to address in Ada:

- Pragmas implemented as compiler plug-ins.
- Expression reduction generalising operator overloading.
- An I/O library that was as usable as in Pascal, but written in the language and user-extensible.
- A language powerful enough to define its own arrays or pointers, while keeping them exactly as usable as built-in types.

10.10. XL2 compiler plugins

XL2 has full support for compiler plug-ins, in a way similar to what had been done with Mozart. However, plug-ins were much simpler to develop and maintain, since they had to deal with a very simple parse tree structure.

For example, the differentiation plugin implements symbolic differentiation for common functions. It is tested here. The generated code after applying the plugin would look like this. The plugin itself is quite simple. It simply applies basic mathematical rules on parse trees. For example, to perform symbolic differentiation on multiplications, the code looks like this:

```
function Differentiate (expr : PT.tree; dv : text) return PT.tree is
    translate expr
        when ('X' * 'Y') then
            dX : PT.tree := Differentiate(X, dv)
            dY : PT.tree := Differentiate(Y, dv)
            return parse_tree('dX' * 'Y' + 'X' * 'dY')
```

Meta-programming became almost entirely transparent here. The `translate` statement, itself provided by a compiler plug-in (see below), matches the input tree against a number of shapes. When the tree looks like `X*Y`, the code behind the matching `then` is evaluated. That code reconstructs a new parse tree using the `parse_tree` function.

Also notice the symmetric use of quotes in the `when` clause and in the `parse_tree` function, in both cases to represent variables as opposed to names in the parse tree. Writing `parse_tree(X)` generates a parse tree with the name `X` in it, whereas `parse_tree('X')` generates a parse tree from the `X` variable in the source code (which must be a parse tree itself).

10.11. XL2 internal use of plugins: the `translation` extension

The compiler uses this plug-in mechanism quite extensively internally. A particularly important compiler extension provides the `translation` and `translate` instructions. Both were used extensively to rewrite XL0 parse trees easily.

We saw above an example of `translate`, which translated a specific tree given as input. It simply acted as a way to compare a parse tree against a number of forms, evaluating the code corresponding to the first match.

The `translation` declaration is even more interesting, in that it is a non-local function declaration. All the `translation X` from all modules are accumulated in a single `X` function. Functions corresponding to `translation X` and `translation Y` will be used to represent distinct phases in the compiler, and can be used a regular functions taking a tree as input and returning the modified tree.

This approach made it possible to distribute `translation XLDeclaration` statements throughout the compiler, dealing with declaration of various entities, with matching `translation XLSemantics` took care of the later semantics analysis phase.

Writing code this way made it quite easy to maintain the compiler over time. It also showed how concept programming addressed what is sometimes called aspect-oriented programming. This was yet another proof of the "extensible" nature of the language.

10.12. Switching to dynamic code generation

One issue I had with the original XL2 approach is that it was strictly a static compiler. The bytecode files made it possible to generate practically any language as output. I considered generating LLVM bitcode, but thought that it would be more interesting to use an XL0 input instead. One reason to do that was to be able to pass XL0 trees around in memory without having to re-parse them. Hence XLR, the XL runtime, was born. This happened around 2009.

For various reasons, I wanted XLR to be dynamic, and I wanted it to be purely functional. My motivations were:

- a long-time interest in functional languages.
- a desire to check that the XL0 representation could also comfortably represent a functional languages, as a proof of how general XL0 was.
- an intuition that sophisticated type inference, Haskell-style, could make programs both shorter and more solid than the declarative type systems of Ada.

While exploring functional languages, I came across Pure, and that was the second big inspiration for XL. Pure prompted me to use LLVM as a final code generator, and to keep XLR extremely simple.

10.13. Translating using only tree rewrites

I sometimes describe XLR as a language with a single operator, `is`, which reads as *transforms into*. Thus, `X is 0` declares that `X` has value `0`.

Until very recently, that operator was spelled using an arrow, as `→`, which I thought expressed the *transforms into* quite well. Around 2018, I decided that this was unreadable for the novice, and switched to using `is` as this *definition operator*. This `→` operator is still what you will find for example on the Tao3D web site.

This notation can be used to declare basic operators:

```
x:integer - y:integer as integer    is opcode Sub
```

It makes a declaration of `writeln` even shorter than it was in XL2:

```
write x:text as boolean            is C xl_write_text
write x:integer as boolean         is C xl_write_integer
write x:real as boolean            is C xl_write_real
write x:character as boolean       is C xl_write_character
write A, B                         is write A; write B
writeln as boolean                 is C xl_write_cr
writeln X as boolean               is write X; writeln
```

The precedence of `;` was changed over time. It is now lower than `is`, but was higher at the time to make it possible to write the code above without curly braces. See the rationale in expression vs. statement for an explanation of why this was changed.

More interestingly, even if-then-else can be described that way:

```
if true  then TrueBody else FalseBody   is TrueBody
if false then TrueBody else FalseBody   is FalseBody
if true  then TrueBody                  is TrueBody
if false then TrueBody                  is false
```

The above code now requires a metabox for `true` in the version of XL described in this document, i.e. `true` must be replaced with `[[true]]` in order to avoid being interpreted as a formal parameter.

Similarly for basic loops, provided your translation mechanism implements tail recursion properly:

```
while Condition loop Body is
    if Condition then
        Body
    while Condition loop Body

until Condition loop Body is while not Condition loop Body

loop Body is { Body; loop Body }

for Var in Low..High loop Body is
    Var := Low
    while Var < High loop
        Body
        Var := Var + 1
```

The fact that such structures can be implemented in the library does not mean that they have to. It is simply a proof that basic amenities can be constructed that way, and to provide a reference definition of the expected behaviour.

10.14. Tao3D, interactive 3D graphics with XL

When I decided to leave HP, I thought that XLR was flexible enough to be used as a dynamic document language. I quickly whipped together a prototype using XLR to drive an OpenGL 3D rendering engine. That proved quite interesting.

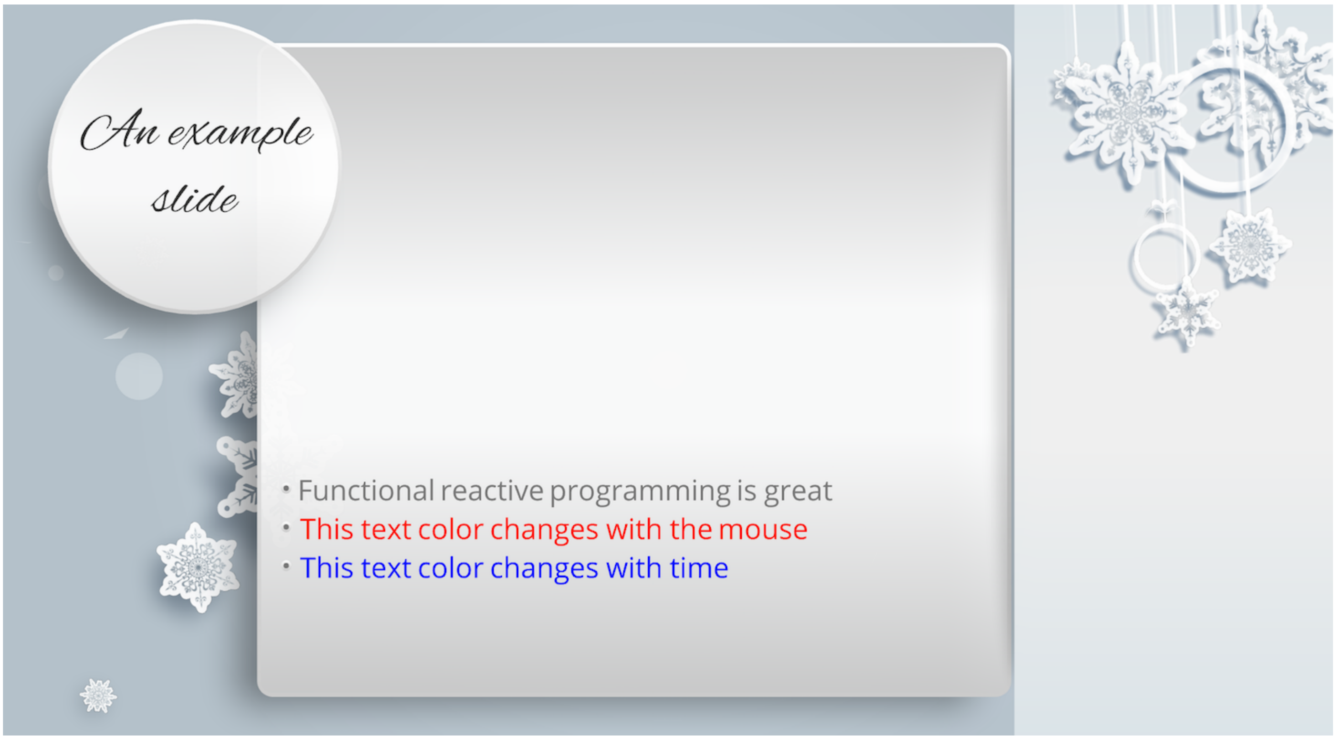
Over time, that prototype morphed into Tao3D. As far as the XLR language itself is concerned, there wasn't as much evolution as previously. A few significant changes related to usability popped up after actively using the language:

- Implicit conversions of integer to real were not in the original XLR, but it was quite annoying in practice when providing object coordinates.
- The XL version in Tao3D also became sensitive to spacing around operators, so as to distinguish `Write -A` from `X - Y`. Earlier versions forced you to use parentheses in the first case, as in `Write (-A)`, which was quite against the ideas of concept programming that your code must match your ideas.
- The more important change was the integration in the language of reactivity to transparently deal with events such as mouse, keyboard or time. Thus, the Tao3D language a fully functional-reactive language, without changing the core translation technology at all.

Precisely because the changes were so minor, Tao3D largely proved the point that XL was really extensible. For example, a `slide` function (that takes code as its second argument) makes it easy to describe a great-looking bullet points slide:

```
import WhiteChristmasTheme
theme "WhiteChristmas"

slide "An example slide",
    * "Functional reactive programming is great"
    color_hsv mouse_x, 100%, 100%
    * "This text color changes with the mouse"
    color_hsv time * 20, 100%, 100%
    * "This text color changes with time"
```

and get an animated slide that looks like this:



The same technique goes well beyond mere bullet points:

image:images/tao3d-slide.jpg

Tao3D developed a relatively large set of specialised modules, dealing with things such as stereoscopy or lens flares. As a product, however, it was never very successful, and Taodyne shut down in 2015, even if the open-source version lives on.

Unfortunately, Tao3D was built on a relatively weak implementation of XL, where the type system in particular was not well thought out (it was really a hack that only supported parse tree types). This made a few things really awkward. Notably, all values are passed by reference, which was mostly an implementation hack to enable the user-interface to "retrofit" values into the code when you move shapes on the screen. Unfortunately, this made the language brittle, and forced many modules to rely on poor hacks when updating values. To make a long story short, `X := Y` in Tao3D is a joke, and I'm rightfully ashamed of it.

10.15. ELFE, distributed programming with XL

ELFE was another experiment with XL, that took advantage of XL's extensibility to explore yet another application domain, namely distributed software, with an eye on the Internet of Things. The idea was to take advantage of the existence of the XL0 standard parse tree to communicate programs and data across machines.

An ELFE program looks as as if it was running on a single machine, but actively exchanges program segments and their associated data between distant nodes (in modern XL, `→` below would read `is`):

```
invoke "pi2.local",
   every 1.1s,
        rasp1_temp ->
            ask "pi.local",
                temperature
        send_temps rasp1_temp, temperature

   send_temps T1:real, T2:real ->
       if abs(T1-T2) > 2.0 then
           reply
               show_temps T1, T2

show_temps T1:real, T2:real ->
    write "Temperature on pi is ", T1, " and on pi2 ", T2, ". "
    if T1>T2 then
        writeln "Pi is hotter by ", T1-T2, " degrees"
    else
        writeln "Pi2 is hotter by ", T2-T1, " degrees"
```

ELFE only adds a very small number of features to the standard XL, which are simply regular XL functions implemented in C++:

- The `ask` statement sends a program, and returns the result of evaluating that program as if it has been evaluated locally. It works like a remote function call.
- An `invoke` statement sends a program to a remote node. It's a "fire and forget" operation, but leaves a reply channel open while it's executing.
- Finally, the `reply` statement allows a remote node to respond to whoever `invoke`'d it, by evaluating one of the available functions in the caller's context.

A few very simple ELFE demos illustrate these remote-control capabilities. For example, it's easy to monitor temperature on two remote sensor nodes, and to ask them to report if their temperatures differ by more than some specified amount. The code is very short and looks like this:

```
invoke "pi2.local",
   every 1.1s,
        rasp1_temp ->
            ask "pi.local",
                temperature
        send_temps rasp1_temp, temperature

   send_temps T1:real, T2:real ->
       if abs(T1-T2) > 2.0 then
           reply
               show_temps T1, T2

show_temps T1:real, T2:real ->
    write "Temperature on pi is ", T1, " and on pi2 ", T2, ". "
    if T1>T2 then
        writeln "Pi is hotter by ", T1-T2, " degrees"
    else
        writeln "Pi2 is hotter by ", T2-T1, " degrees"
```

ELFE was designed to run with a small memory footprint, so it provides a complete interpreter that does not require any LLVM. As the names in the example above suggest, it was tested on Raspberry Pi. On the other hand, the LLVM support in that "branch" of the XL family tree fell into a bad state of disrepair.

10.16. XL gets a type system

Until that point, XL lacked a real type system. What was there was mostly quick-and-dirty solutions for the most basic type checks. Over a Christmas vacation, I spent quite a bit of time thinking about what a good type system would be for XL. I was notably stumped by what the type of `if-then-else` statements should be.

The illumination came when I realized that I was blocked in my thinking by the false constraint that each value had to have a single type. Instead, the type system that seems natural in XL is that a type indicates the shape of a parse tree. For example, `integer` is the type of integer constants in the code, `real` the type of real constants, and `type(X+Y)` would be the type of all additions.

Obviously, that means that in XL, a value can belong to multiple types. `2+3*5` belongs to `type(X+Y)`, to `type(X:integer+Y:integer)` or to `infix`. This makes the XL type system extremely powerful. For example. a type for even numbers is `type(X:integer when X mod 2 = 0)`.

ELFE also gave me a chance to implement a relatively crude version of this idea and validate that it's basically sane. Bringing that idea to the optimizing compiler was an entirely different affair, though, and is still ongoing.

10.17. The LLVM catastrophy

For a while, there were multiple subtly distinct variants of XL which all shared the same XL0, but had very different run-time constraints.

- Tao3D had the most advanced library, and a lot of code written for it. But that code often depends on undesirable behaviours in the language, such as implicit by reference argument passing.
- ELFE had the most advanced type system of all XLR variants, being able to perform overloading based on the shape of parse trees, and having a rather complete set of control structures implemented in the library. It also has an interesting modular structure, and a rather full-featured interpreter.
- XLR has the most advanced type inference system, allowing it to produce machine-level instructions for simple cases to get performance that was on a par with C. Unfortunately, due to lack of time, it fell behind with respect to LLVM support, LLVM not being particularly careful about release-to-release source compatibility. And the type inference was never solid enough to retrofit it in Tao3D, which still uses a much simpler code generation.
- XL2 was the only self-compiling variant of XL ever written, and still had by far the most sophisticated handling of generics and most advanced library. But it has been left aside for a few years. As an imperative language, it is more verbose and feels heavier to program. Yet it is not obsolete, as the discussion above demonstrates. Its type system, with its support for generics or validation, is much more robust than whatever was ever implemented in all XLR variants. It would need quite a bit of love to make it really usable, for example improving the standard library and actually connect XLR as a bytecode back-end as initially envisioned.

In addition to this divergence, another problem arose externally to the XL project. The LLVM library, while immensely useful, proved a nightmare to use, because they purposely don't care about source cdoe compatibility between releases. XLR was initially coded against LLVM 2.5, and the majority of Tao3D development occured in the LLVM 2.7 time frame.

Around release 3.5, LLVM started switching to a completely different code generation model. Being able to support that new model proved extremely challenging, in particular for something as complex as Tao3D. The problem is not unique to Tao3D: the LLVM pipe in the Mesa project has similar issues. But in Tao3D, it was made much worse precisely because Tao3D uses both OpenGL and XL, and the Mesa implementation of OpenGL commonly used on Linux also uses LLVM. If the variants of LLVM used by the XL runtime and by OpenGL don't match, mysterious crashes are almost guaranteed.

From 2015 to 2018, all development of XL and Tao3D was practically stuck on this problem. It did not help that my job during that time was especially challenging time-wise. In practice, the development of Tao3D and XLR was put on hold for a while.

10.18. Repairing and reconverging

A project that lasted several months, called `bigmerge` allowed me to repair a lot of the issues:

- The XL2 compiler was brought back into the main tree
- The ELFE interpreter was merged with the main tree, and its modular approach (designed to allow the use of XL as an extension language) was incorporated in XL. As a result, ELFE is dead, but it lives on in the main XL tree. XL was also turned into a library, with a very small front-end calling that library to evaluate the code.

- The switch from `→` to `is` as the definition operator was implemented.
- The LLVM "Compatibility Restoration Adaptive Protocol" (LLVM-CRAP) component of XL was completely redesigned, giving up pre-3.5 versions of LLVM, but supporting all the recent ones (from 3.7 to 9.0).
- The Tao3D branch of the compiler was forward-ported to this updated compiler, under the name `FastCompiler`. That work is not complete, howver, because some of the changes required by the two previous steps are incompatible with the way Tao3D was interfacing with XL.

This is the current state of the XL tree you are looking at. Not pretty, but still much better than two years ago.

10.19. Language redefinition

During all that time, the language definition had been a very vaguely defined TeXMacs document. This document had fallen somewhat behind with respect to the actual language implementation or design. Notably, the type system was quickly retrofitted in the document. Also, the TexMacs document was monolithic, and not easy to convert to a web format.

So after a while, I decided to rewrite the documentation in markdown. This led me to crystalize decisions about a few things that have annoyed me in the previous definition, in particular:

- The ambiguity about formal parameters in patterns, exhibited by the definition of `if-then-else`. The XL language had long defined `if-then-else` as follows:

  ```
  if true  then TrueClause      is TrueClause
  if false then TrueClause      is false
  ```

  There is an obvious problem in that definition. Why should `true` be treated like a constant while `TrueClause` a formal parameter?

  The solution proposed so far so far was that if a name already existed in the context, then we were talking about this name. In other words, `true` was supposed to be defined elsewhere and not `TrueClause`.

  This also dealt with patterns such as `A - A is 0`. However, the cost was very high. In particular, a formal parameter name could not be any name used in the enclosing context, which was a true nuisance in practice.

  More recently, I came across another problem, which was how to properly insert a computed value like the square root of two in a pattern? I came up with an idea inspired parameters in `translate` statements in XL2, which I called a "metabox". The notation `[[X]]` in a pattern will evaluate `X`. To match the square root of 2, you would insert the metabox `[[sqrt 2]]`. To match `true` instead of defining a name `true`, you would insert `[[true]]` instead of `true`.

  Downside: fix all the places in the documentation that had it backwards.

- The addition of opaque binary data in parse trees, for example to put an image from a PNG file in an XL program. I had long been thinking about a syntax like `binary "image.png"` It should also be possible to declare arbitrary binary data inline, as in `binary 16#FFFF_0000_FFFF_0000_FF00_00FF_FF00_00FF`.

- Adding a `lambda` syntax for anonymous functions. Earlier versions of XL would use a catch-all pattern like `(X is X + 1)` to define a lambda function, so that `(X is X + 1) 3` would be `4`. That pattern was only recognized in some specific contexts, and in other contexts, this would be a definition of a variable named `X`. It is now mandatory to add `lambda` for a catch-all pattern, as in `lambda X is X + 1`, but then this works in any context.

10.20. Future work

The work that remains to make XL usable again (in the sense of being as stable as it was for Tao3D in the 2010-2015 period) includes:

- Complete the work on an Haskell-style type inference system, in order to make the "O3" compiler work well.
- Repair the Tao3D interface in order to be able to run Tao3D again with modern LLVM and OpenGL variants.
- Re-connect the XL2 front-end for those who prefer an imperative programming style, ideally connecting it to XLR as a runtime.
- Sufficient library-level support to make the language usable for real projects.
- Bootstrapping XLR as a native compiler, to validate that the XLR-level language is good enough for a compiler. Some of the preparatory work for this is happening in the `native` directory.
- Implement a Rust-style borrow checker, ideally entirely from the library, and see if it makes it possible to get rid of the garbage collector. That would be especially nice for Tao3D, where GC pause, while generally quite small, are annoying.
- Some reworking of XL0, notably to make it easier to add terminal node types. An example of use case is version numnbers like `1.0.1`, which are hard to deal with currently. The distinction between `Integer` and `Real` is indispensable for evaluation, but it may not be indispensable at parse time.
- Replace blocks with arrays. Currently, blocks without a content, such as `( )` or `{ }`, have a blank name inside, which I find ugly. It would make more sense to consider them as arrays with zero length. Furthermore, blocks are often used to hold sequences, for example sequences of instructions. It would be easier to deal with a block containing a sequence of instructions than with the current block containing an instruction or a chain of infix nodes.
- Adding a "binary object" node type, which could be used to either describe data or load it from files. I have been considering a syntax like:

  ```
  binary 16#0001_0002_0003_0004_0005_0006_0007_0008_0009
  binary "image.jpg"
  ```

It is unclear if I will be able to do that while at the same time working on my job at Red Hat and on various other little projects such as `make-it-quick` or `recorder` (which are themselves off-shoots of XL development).

Index