The ELIOT Programming Language

by Christophe de Dinechin

Taodyne SAS, 1300 Route des Cretes, Sophia Antipolis, France

1 Introduction

ELIOT is a dynamic language derived from XL and based on meta-programming, i.e. programs that manipulate programs. It is designed to enable a natural syntax similar to imperative languages descending from Algol (e.g. C, Pascal, Ada, Modula), while preserving the power and expressiveness of homoiconic languages descending from Lisp (i.e. languages where programs are data). But more importantlly, XL is designed from the ground up to be *extensible*. XL lets you extend the core language to suit your own needs. It should be as easy to add new language constructs to XL as it is to add functions or classes in existing programming languages.

Warning 1. This document describes the language XL as it should ultimately be implemented. The current ELIOT implementations are currently incomplete. An attempt will be made to indicate in this document features that are not yet available with a warning. See also Section 1.5.

1.1 Design objectives

ELIOT is intended to make it easy to write simple programs, yet impose no upper limit, since you can extend the language to suit your own needs. The goal is to make it as easy and robust to extend the language with new features or concepts as it is to add new functions or classes in more traditional programming languages.

This design is in response to the following observation: programmers have to deal with exponentially-growing program complexity. The reason is that the complexity of programs indirectly follows Moore's law, since users want to fully benefit from the capabilities of new hardware generations. But our brains do not follow a similar exponential law, so we need increasingly sophisticated tools to bridge the gap with higher and higher levels of abstraction.

Over time, this lead to a never ending succession of *programming paradigms*, each one intended to make the next generation of hardware accessible to programmers. For example, object-oriented programming was primarily fueled by the demands of graphical user interfaces.

The unfortunate side effect of this continuous change in programming paradigms is that code designed with an old approach quickly becomes obsolete as a new programming model emerges. For example, even if C++ is nominally compatible with C, the core development model is so incompatible that C++ replicates core functionality of C in a completely different way (memory allocation, I/Os, containers, sorts, etc).

The purpose of ELIOT is to allow the language to grow naturally over time, under programmers' control. ELIOT actually stands for "eXtensible[1] Language and Runtime". The long term vision is a language made both more powerful and easier to use thanks to a large number of community-developed and field-tested language extensions.

1.2 Keeping it simple

In order to keep programs easy to write and read, the ELIOT syntax and semantics are very simple. As Saint-Exupery once said, perfection is achieved, not when there's nothing to add, but when there's nothing left to take away.

1. In prehistoric versions of XL, the X stood for "eXperimental"

And indeed, the ELIOT syntax will seem very natural to most programmers, except for what it's missing: ELIOT makes little use of parentheses and other punctuation characters. Instead, the syntax is based on indentation. There was a conscious design decision to keep only symbols that had an active role in the meaning of the program, as opposed to a purely syntactic role. ELIOT programs look a little like pseudo-code, except that they can be compiled and run.

This simplicity translates into the internal representations of programs, which makes meta-programming not just possible, but easy. Any ELIOT program or data can be represented with just 8 data types: integer, real, text, name, infix, prefix, posfix and block. For example, the internal representation for `3 * sin X` is an infix `*` node with two children, the left child of `*` being integer `3`, and the right child of `*` being a prefix with the name `sin` on the left and the name `X` on the right. Therefore, these basic types define an *abstract syntax tree* (AST).

The data structure representing programs is simple enough to make meta-programming practical and easy. Meta-programming is the ability for a program to deal with programs. In the case of ELIOT, meta-programming is the key to language extensibility. Instead of wishing you had this or that feature in the language, you can simply add it. Better yet, the process of extending the language is so simple that you can now consider language notations or compilation techniques that are useful only in a particular context. In short, with ELIOT, creating your own *domain-specific languages* (DSLs) is just part of normal, everyday programming.

ASTs are also central to understanding how ELIOT programs are executed. Conceptually, an ELIOT program is a transformation of ASTs following a number of tree rewrite rules. In other words, there is no real difference in ELIOT between meta-programming and normal program execution, as both are represented by tranformations on ASTs. The entire ELIOT semantics boils down to this: incrementally rewriting abstract syntax trees.

1.3 Examples

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

Figure 1 illustrates the definition of the factorial function:

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

Figure 1. Declaration of the factorial function

Figure 2 illustrates operations usually known as *map*, *reduce* and *filter*. These operations are characteristic of a programming paradigm called *functional programming*, because they take functions as arguments. In ELIOT, *map*, *reduce* and *filter* operations can all use an infix `with` notation with slightly different forms for the parameters. Section 4.1.10 describes these operations in more details.

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

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

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

Figure 2. Map, reduce and filter

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

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

Figure 3. Declaration of if-then-else

The next sections will clarify how these operations work.

1.4 Concept programming

Concept programming is the underlying design philosophy behind ELIOT. The core idea is very simple:

Programming is the art of transforming ideas (i.e. concepts *that belong to* concept space*) into artifacts such as programs or data structure (i.e.* code *that belongs to* code space*).*

From concept to code: a lossy conversion Concepts and code do not exist in the same context, do not obey the same rules, and are generally hard to compare. However, experience shows that it is generally a good idea to make the code look and feel as close to the concept it represents as possible. Unfortunately, doing so is incredibly difficult in practice, in large part because computers and code are limiting in their ability to represent arbitrary concepts.

We are quite good at building abstractions that bridge the gap, for example integer data types and arithmetic that mimic mathematical integers and arithmetic. But then we tend to forget these are only abstractions, and get caught when they do not behave like the real thing, for example when an `int` overflows or wraps around, something that real integers never do.

The key takeaway is that the conversion of concept to code is necessarily *lossy*. Minimizing the loss remains a worthy goal, but doing so is difficult. By drawing our attention to the conversion process itself, concept programming gives us new and useful tools to solve old problems.

Pseudo-metrics Among the tools brought by concept programming is a set of *pseudo-metrics* allowing us to better evaluate the code we create. These are called pseudo-metrics because they apply to things that in all fairness cannot really be measured, like the distance between concepts in our brains and code in the computer. At the same time, they are easy to understand and use, and allow us to identify and solve problems that are otherwise hard to pinpoint.

Key pseudo-metrics in concept programming include:

1. *Syntactic noise* is a discrepancy between the appearance of the code and the usual or desired notation for the associated concept. For example, the usual mathematical operation $1 + 2$ is ideally represented in the code by `1+2`. Notations such as `(+ 1 2)` or `add(1,2)`, by contrast, introduce a little bit of syntactic noise.

2. *Semantic noise* is a discrepancy between the meaning of the code and the usual or desired notation for the associated concept. For example, when one needs to consider if computing `X+1` possibly overflows, runs out of memory, throws an exception or takes an unpredictable amount of time to compute, then a little bit of semantic noise appears.

3. *Bandwith* is the fraction of the concept space that is covered by a given code. The larger the bandwidth, the more general the code is. For example, the mathematical *minimum* concept includes the ability to compare almost anything provided there is an order relation (which may be total or partial); it applies to functions, to sets, to series, and so on. So it's fair to say that the following C function is very narrow band:

   ```
   int min(int x, int y) { return x<y?x:y; }
   ```

4. The *signal-noise ratio* is the fraction of the code that is actually useful to solve the problem from concept space, as opposed to code that is there only because of code-space considerations. In the same `min` example given above, semi-colons or curly braces have little to do with the problem at hand: they are noise rather than signal.

An amusing observation about this choice of terminology is that just like in engineering, noise cannot ever be completely eliminated, though many techniques exist to reduce it; and just like in art, what is noise to one person may be music to another.

Influence on ELIOT ELIOT is the first programming language designed specifically with concept programming in mind. As a result, it is also the first programming language that explicitly attempts to optimize the pseudo-metrics listed above.

1.5 State of the implementation

The current implementation of the language is available as an open-source program, in particular at http://xlr.sourceforge.net. A few details of the implementation are given in Section 6.

There are currently three wildly different implementations in one program:

- An interpreted mode, corresponding to optimization level 0, which proved so slow that it has not been maintained in a while and is a state of serous disrepair.
- The standard mode, which generates machine code with LLVM and runs acceptably fast on simple programs. This implementation is used by Tao Presentations, and has consequently been more field tested for relatively large real-time interactive applications. It has, however, serious limitations with respect to the language.
- The optimized mode, which generates much better machine code with LLVM (practically on a par with optimized C for simple examples) by using techniques such as type inference. This optimized mode was supposed to take over, and as a result, many language constructs where implemented only in that optimized mode (for example, guards). Unfortunately, bringing it up to par with the standard mode proved more difficult than planned, which means that optimized mode cannot yet support Tao Presentations.

The net result of this development history is that the same compiler supports slightly different features depending on the optimization level. Best effort attempts will be made to indicate which features are impacted with Warnings in this document.

2 Syntax

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

The precedence of operators is given by the `xl.syntax` configuration file. It can also be changed dynamically in the source code using the `syntax` statements. This is detailed in Section 2.6. Both methods to define syntax are called *syntax configuration*.

The rest of this document will occasionally refer to *normal ELIOT* for defaults settings such as the default syntax configuration, as shipped with the standard ELIOT distribution.

2.1 Spaces and indentation

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

Spaces and tabs are significant at the beginning of lines. ELIOT will use them to determine the level of indentation from which it derives program structures (off-side rule), as illustrated in Figure 4. Both space or tabs can be used for indentation, but cannot be mixed for indentation in a single source file. In other words, if the first indented line uses spaces, all other indentation must be done using spaces, and similarly for tabs.

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

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

2.2 Comments and spaces

Comments are section of the source text which are typically used for documentation purpose and play no role in the execution of the program. Comments begin with a comment separator, and finish with a comment terminator.

Comments in normal ELIOT are similar to C++ comments: they begin with `/*` and finish with `*/`, or they begin with `//` and finish at the end of line. This is illustrated in Figure 5.

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

Figure 5. Single-line and multi-line comments

While comments play no actual role in the execution of a normal ELIOT program, they are actually recorded as attachments in XL0. It is possible for some special code to access or otherwise use these comments. For example, a documentation generator can read comments and use them to construct documentation automatically.

2.3 Literals

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

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

2.3.1 Integer constants

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

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

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

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

Figure 6. Valid integer constants

2.3.2 Real constants

Real constants such as `3.14` consist of one or more digits (`0123456789`), followed by a dot `.` followed by one or more digits (`0123456789`). Note that there must be at least one digit after the dot, i.e. `1.` is not a valid real constant, but `1.0` is.

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

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

A real constant can have an exponent, which consists of an optional hash sign `#`, followed by the character `e` or `E`, followed by optional plus `+` or minus `-` sign, followed by one or more decimal digits `0123456789`. For example, `1.0e-3` is the same as `0.001` and `1.0E3` is the same as `1000.0`. The exponent value is always given in radix-10, and indicates a power of the given radix. For example, `2#1.0e3` represents 2^3, in other words it is the same as `8.0`.

The hash sign in the exponent is required for any radix greater than 14, since in that case the character `e` or `E` is also a valid digit. For instance, `16#1.0E1` is approximately the same as `1.05493`, whereas `16#1.0#E1` is the same as `16.0`.

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

Figure 7. Valid real constants

2.3.3 Text literals

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

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

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

Other text delimiters can be specified, which can be used to delimit text that may include line breaks. Such text is called *long text*. With the default configuration, long text can be delimited with `<<` and `>>`.

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

Figure 8. Valid text constants

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

Source code	Resulting text

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

Figure 9. Long text and indentation

The text delimiters are not part of the value of text literals. Therefore, text delimiters are ignored when comparing texts.

Warning 2. The current implementation may not treat indentation of long text as indicated above.

2.3.4 Name and operator symbols

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

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

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

After parsing, operator and name symbols are treated identically. During parsing, they are treated identically except in the *expression versus statement rule* explained in Section 2.5.4.

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

Figure 10. Examples of valid operator and name symbols

2.4 Structured nodes

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

1. Infix nodes, representing operations such as `A+B` or `A and B`, where the operator is between its two operands.
2. Prefix nodes, representing operations such as `+3` or `sin x`, where the operator is before its operand.
3. Postfix nodes, representing operations such as `3%` or `3 cm`, where the operator is after its operand.
4. Blocks, representing grouping such as `(A+B)` or `{lathe;rinse;repeat}`, where the operators surround their operand.

Infix, prefix and postfix nodes have two children nodes. Blocks have a single child node.

2.4.1 Infix nodes

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

Infix nodes are used to separate statements with semi-colons `;` or line breaks (referred to as `NEWLINE` in the syntax configuration).

2.4.2 Prefix and postfix nodes

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

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

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

2.4.3 Block nodes

Block nodes have one child bracketed by two delimiters.

Normal ELIOT recognizes the following pairs as block delimiters:

- Parentheses, as in `(A)`
- Brackets, as in `[A]`
- Curly braces, as in `{A}`
- Indentation, as shown surrounding the `write` statements in Figure 4. The delimiters for indentation are referred to as `INDENT` and `UNINDENT` in the syntax configuration.

2.5 Parsing rules

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

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

These rules are detailed below.

2.5.1 Precedence

Infix, prefix, postfix and block symbols are ranked according to their *precedence*, represented as a non-negative integer. The precedence is specified by the syntax configuration, either in the syntax configuration file, `xl.syntax`, or through `syntax` statements in the source code. This is detailed in Section 2.6.

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

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

The precedence associated to blocks is used to define the precedence of the resulting expression. This precedence given to entire expressions is used primarily in the *expression versus statement* rule described in Section 2.5.4.

2.5.2 Associativity

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

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

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

Operators with left and right associativity cannot have the same precedence, as this would lead to ambiguity. To enforce that rule, ELIOT arbitrarily gives an even precedence to left-associative operators, and an odd precedence to right-associative operators. For example, the precedence of `+` in the default configuration is `290` (left-associative), whereas the precedence of `^` is `395` (right-associative).

2.5.3 Infix versus Prefix versus Postfix

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

ELIOT resolves this ambiguity as follows[4]:

- The first symbol in a statement or in a block is a prefix: `and` in `(and x)` is a prefix.
- A symbol on the right of an infix symbol is a prefix: `and` in `A+and B` is a prefix.
- Otherwise, if the symbol has an infix precedence but no prefix precedence, then it is interpreted as an infix: `and` in `A and B` is an infix.
- If the symbol has both an infix precedence and a prefix precedence, and either a space following it, or no space preceding it, then it is an infix: the minus sign `-` in `A - B` is an infix, but the same character is a prefix in `A -B`.
- Otherwise, if the symbol has a postfix precedence, then it is a postfix: `%` in `3%` is a postfix.
- Otherwise, the symbol is a prefix: `sin` in `write sin x` is a prefix.

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

2.5.4 Expression versus statement

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

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

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

More precisely:

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

In practice, there is no need to worry too much about these rules, since normal ELIOT ensures that most text parses as one would expect from daily use of English or mathematical notations.

2.6 Syntax configuration

The default ELIOT syntax configuration file, named `xl.syntax`, looks like Figure 11 and specifies the standard operators and their precedence.

4. All the examples given are in normal XL, i.e. based on the default `xl.syntax` configuration file.

```
INFIX
        11      NEWLINE
        21      -> =>
        25      as
        31      else into
        40      loop while until
        50      then require ensure
        61      ;
        75      with
        85      := += -= *= /= ^= |= &=
        100     STATEMENT
        110     is
        120     written
        130     where
        200     DEFAULT
        211     when
        231     ,
        240     return
        250     and or xor
        260     in at contains
        271     of to
        280     .. by
        290     = < > <= >= <>
        300     & |
        310     + -
        320     * / mod rem
        381     ^
        500     .
        600     :

PREFIX
        30      data
        40      loop while until
        50      property constraint
        121     case if return yield transform
        350     not in out constant variable const var
        360     ! ~
        370     - + * /
        401     FUNCTION
        410     function procedure to type iterator
        420     ++ --
        430     &

POSTFIX
        400     ! ? % cm inch mm pt px
        420     ++ --

BLOCK
        5       INDENT UNINDENT
        25      '{' '}'
        500     '(' ')' '[' ']'

TEXT
        "<<" ">>"

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

SYNTAX "C"
        extern ;
```

Figure 11. Default syntax configuration file

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

```
// Declare infix 'weight' operator
syntax (INFIX 350 weight)
Obj weight W -> Obj = W

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

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

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

As a general stylistic rule, it is recommended to use restraint when introducing new operators using `syntax` statements, as this can easily confuse a reader who is not familiar with the new notation. On the other hand, there are cases where good use of new and well-chosen operators will render the code much more readable and easy to maintain.

Format of syntax configuration Spaces and indentation are not significant in a syntax configuration file. Lexical elements are identical to those of ELIOT, as detailed in Section 2.3. The significant elements are integer constants, names, symbols and text. Integer constants are interpreted as the precedence of names and symbols that follow them. Name and symbols can be given either with lexical names and symbols, or with text.

A few names are reserved for use as keywords in the syntax configuration file:

- `INFIX` begins a section declaring infix symbols and precedence. In this section:
  - `NEWLINE` identifies line break characters in the source code
  - `STATEMENT` identifies the precedence of statements
  - `DEFAULT` identifies the precedence for symbols not otherwise given a precedence. This precedence should be unique in the syntax confguration, i.e. no other symbol should be given the `DEFAULT` precedence.
- `PREFIX` begins a section declaring prefix symbols and precedence. In this section:
  - `FUNCTION` identifies the precedence for default prefix symbols, i.e. symbols identified as prefix that are not otherwise given a precedence. This precedence should be unique, i.e. no other symbol shoud be given the `FUNCTION` precedence.
- `POSTFIX` begins a section declaring postfix symbols and precedence.
- `BLOCK` begins a section declaring block delimiters and precedence. In this section:
  - `INDENT` and `UNINDENT` are used to mark indentation and unindentation.
- `TEXT` begins a section declaring delimiters for long text.
- `COMMENT` begins a section declaring delimiters for comments. In this section:
  - `NEWLINE` identifies line breaks
- `SYNTAX` begins a section declaring external syntax files. In normal ELIOT, a file `C.syntax` is used to define the precedences for any text between `extern` and `;` symbols. This is used to import C symbols using an approximation of the syntax of the C language, as described in Section 4.4. The `C.syntax` configuration file is shown in Figure 13.

```
INFIX
        41      ,

PREFIX
        30      extern ...
        400     FUNCTION
        450     short long unsigned signed

POSTFIX
        100     *

BLOCK
        500     '(' ')' '[' ']'

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

Figure 13. C syntax configuration file

3 Language semantics

The semantics of ELIOT is based entirely on the rewrite of XL0 abstract syntax trees. Tree rewrite operations define the execution of ELIOT programs, also called *evaluation*.

3.1 Tree rewrite operators

There is a very small set of tree rewrite operators that are given special meaning in ELIOT and treated specially by the ELIOT compiler:

- *Rewrite declarations* are used to declare operations. They roughly play the role of functions, operator or macro declarations in other programming languages. A rewrite declaration takes the general form `Pattern->Implementation` and indicates that any tree matching `Pattern` should be rewritten as `Implementation`.

```
0! -> 1
N! -> N*(N-1)!
3! // Computes 6
```

Figure 14. Example of rewrite declaration

- *Data declarations* identify data structures in the program. Data structures are nothing more than trees that need no further rewrite. A data declaration takes the general form of `data Pattern`. Any tree matching `Pattern` will not be rewritten further.

```
data complex(x, y)
complex(3,5) // Will stay as is
```

Figure 15. Example of data declaration

- *Type declarations* define the type of variables. Type declarations take the general form of an infix colon operator `Name:Type`, with the name of the variable on the left, and the type of the variable on the right.

```
data person
    first:text
    last:text
    age:integer
person
    "John"
    "Smith"
    33
```

Figure 16. Example of data declarations containing type declarations

- *Guards* limit the validity of rewrite or data declarations. They use an infix `when` with a boolean expression on the right of `when`, i.e. a form like `Declaration when Condition`.

```
syracuse X:integer when X mod 2 = 0 -> X/2
syracuse X:integer                  -> 3*X+1
```

Figure 17. Example of guard to build the Syracuse suite

- *Assignment* change the value associated to a binding. Assignments take the form `Reference := Value`, where `Reference` identifies the binding to change.

```
Zero := 0
```

Figure 18. Example of assignment

- *Sequence operators* indicate the order in which computations must be performed. ELIOT has two infix sequence operators, the semi-colon `;` and the new-line `NEWLINE`.

```
write "Hello"; writeln " World"
emit_loud_beep
```

Figure 19. Example of sequence

- *Index operators* perform particular kinds of tree rewrites similar in usage to "structures" or "arrays" in other programming languages. The notations `Reference.Field` and `Reference[Index]` are used to refer to individual elements in a data structure. These are only convenience notations for specific kinds of tree rewrites, see Section 3.1.7.

```
A[3] := 5
A.ref_count := A.ref_count + 1
```

Figure 20. Examples of index operators

Warning 3. The current implentation has a number of deficiencies with respect to guards, assignment and index operators. These are described in the corresponding sections below.

3.1.1 Rewrite declarations

The infix `->` operator declares a tree rewrite. Figure 21 repeats the code in Figure 3 illustrating how rewrite declarations can be used to define the traditional `if-then-else` statement.

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

Figure 21. Examples of tree rewrites

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

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

- Infix symbols and names are constant, like `+` in `A+B`.
- Block-delimiting symbols and names are constant, like `[` and `]` in `[A]`.

- A name on the left of a prefix is a constant, like `sin` in `sin X`.
- A name on the right of a postfix is a constant, like `cm` in `X cm`.
- A name alone on the left of a rewrite is a constant, like `X` in `X->0`.
- Operators are constant, like `+` in `X and +Y`.
- All other names are variable.

Figure 22 highlight in blue italic all variable symbols in the declarations of Figure 21.

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

Figure 22. Constants vs. Variable symbols

Constant symbol and names form the structure of the pattern, whereas variable names form the parts of the pattern which can match other trees. The names are called *parameters* and the tree they match are called *arguments*.

For example, to match the pattern in Figure 21, the `if`, `then` and `else` words must match exactly, but `TrueClause` may match any tree, like for example `write "Hello"`. `TrueClause` is a parameter, and `write "Hello"` would be the matching argument.

Note that there is a special case for a name as the pattern of a rewrite. A rewrite like `X->0` binds `X` to value `0`, i.e. `X` is a constant that must match in the tree being evaluated.

It is however possible to create a rewrite with a variable on the left by using a type declaration. For example, the rewrite `X:real->X+1` does not declare the variable `X`, but an *anonymous function*[5] that increments its input. Such rewrites are somewhat special, in particular because they are not visible to their implementation so as to avoid infinite recursion if their return type is identical to their input type.

An expression may use declarations that follow it in the same sequence. Declarations are visible to prior elements in the sequence and need not be evaluated, as shown in Figure 23, which computes `4`. More generally, rewrites in a sequence belong to the context for the entire sequence (contexts are defined in Section 3.2).

```
foo 3
foo N -> N + 1
```

Figure 23. Declarations are visible to the entire sequence containing them

3.1.2 Data declaration

The `data` prefix declares tree structures that need not be rewritten further. For instance, Figure 24 declares that `1,3,4` should not be evaluated further, because it is made of infix `,` trees which are declared as `data`.

```
data a,b
```

Figure 24. Declaring a comma-separated list

The tree following a data declaration is a pattern, with constant and variable symbols like for rewrite declarations. Data declarations only limit the rewrite of the tree specified by the pattern, but not the evaluation of pattern variables. In other words, pattern variables are evaluated normally, as specified in Section 3.3.

For instance, in Figure 25, the names `x` and `y` are variable, but the name `complex` is constant because it is a prefix. Using integer addition as defined in normal ELIOT, `complex(3+4, 5+6)` will evaluate as `complex(7,11)` but no further[6].

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

Figure 25. Declaring a `complex` data type

The declaration in Figure 25 can be interpreted as declaring a `complex` data type. There is, however, a better way to describe data types in ELIOT, which is detailed in Section 3.4.2.

The word `self` can be used to build data forms: `data X` is equivalent to `X->self`.

Warning 4. The `self` keyword is not implemented yet.

5. In functional programming, these are often called *lambda functions*.
6. Evaluation is caused by the need to check the parameter types, i.e. verify that `3+4` is actually an `integer`.

3.1.3 Type declaration

An *type declaration* is an infix colon `:` operator in a rewrite or data pattern with a name on the left and a type on the right. It indicates that the named parameter on the left has the type indicated on the right. A *return type declaration* is an infix `as` in a rewrite pattern with a pattern on the left and a type on the right. It specifies the value that will be returned by the implementation of the rewrite. Section 3.4 explains how types are defined.

Figure 26 shows examples of type declarations. To match the pattern for `polynom`, the arguments corresponding to parameters `X` and `Z` must be `real`, whereas the argument corresponding to parameter `N` must be `integer`. The value returned by `polynom` will belong to `real`.

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

Figure 26. Simple type declarations

The type declarations filter which rewrites can be selected to evaluate a particular tree. This enables *overloading*, i.e. the ability to have multiple functions or operators with a similar structure, but different types for the parameters. Return type declarations, on the other hand, plays no role in the selection of candidates[7]. If there is a return type declaration and the implementation does not actually return the declared return type, a type error expression of the form `type_error ExpectedType, ActualValue` will attempt to correct the problem.

A type declaration can also be placed on the left of an assignment, see Section 3.1.4.

3.1.4 Assignment

The assignment operator `:=` binds the reference on its left to the value of the tree on its right. The tree on the right is evaluated prior to the assignment.

An assignment is valid even if the reference on the left of `:=` had not previously been bound. In that case, it creates a new binding in the current context. This is shown in Figure 6.15.

```
// Assigns to locally created X
assigns_to_local -> X := 1
```

Figure 27. Creating a new binding

On the other hand, if there is an existing binding, the assignment replaces the corresponding bound value. This is shown in Figure 6.15:

```
// Assigns to global X defined below
assigns_to_global -> X := 1
X -> 0
```

Figure 28. Assignment to existing binding

Warning 5. In the current state of the standard implementation, assigning to an existing rewrite must respect the type and overwrites the value in place. For example, if there is a declaration like `X->0`, you may assign `X:=1` and then `X` will be replaced with `1`. But you will not be able to assign `X:="Hello"`. Furthermore, it is currently only possible to assign scalar types, i.e. integer, real and text values. You cannot assign an arbitrary tree to a rewrite.

Local variables If the left side of an assignment is a type declaration, that assignment creates a new binding in the local scope, as illustrated in Figure 6.15. That binding has a return type declaration associated with it, so that later assignments to that same name will only succeed if the type of the assigned value matches the previously declared type. This is shown in Figure 6.15:

```
// Global X
X := 0

// Assign to local X
assigns_new -> X:integer := 1
```

Figure 29. Assigning to new local variable

7. Ada is one of the few programming languages that have overloading based on return types.

Warning 6. Assigning to a new local may not work in the current implementation.

Assigning to references If the left side of an assignment is a reference, then the assignment will apply to the referred value, as shown in Figure 30. This may either modify the referred value if a binding already exists, or create a new binding in the context being referred to if no binding exists.

```
Data ->
    0 -> 3
    1 -> 2
Data.0 := 4 // replaces 3
Data.2 := 5 // Creates new binding 2->5 in Data
```

Figure 30. Assignment to references

An assignment can also assign to the following special references (see Section 4.1.9):

- `left X`, `right X` when `X` is an infix, prefix or postfix
- `child X` when `X` is a block
- `symbol X` when `X` is a name or infix and the assigned value is a text
- `opening X` and `closing X` when `X` is a block or text and the assigned value is a text

Warning 7. Assignment to references, and in particular to portions of a tree, is mostly broken and does not work in the current implementation, whether standard or optimized.

Assigning to parameters Assigning to a reference is particularly useful for parameters. In some cases, parameters may be bound without being evaluated (see Section 3.3.3). This means that the parameter is bound to a reference. In that case, assigning to the parameter will assign to the reference, making it possible to implement assignment-like operations, as illustrated in Figure 31.

```
A : integer := 5
A+=3

// Effectively assign to A
X+=Y -> X:=X+Y
```

Figure 31. Assigning to parameter

In that example, the context for evaluating the implementation `X:=X+Y` will contain a binding for `X` in the form `X->(A->5).A`, where `(A->5)` is the original execution context. The expression `(A->5).A` means that we evaluate `A` in the context that existed at the point where expression `A+=3` was evaluated. Therefore, assigning to `X` will affect the existing binding,, resulting in the updated binding `A->8` in the original context.

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

3.1.5 Guards

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

Figure 32 shows an improved definition of the factorial function which only applies for non-negative values. This set of rewrites is ignored for a negative `integer` value.

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

Figure 32. Guard limit the validity of operations

A form where the guard cannot be evaluated or evaluates to anything but the value `true` is not selected. For example, if we try to evaluate `'ABC'!` the condition `N>0` is equivalent to `'ABC'>0`, which cannot be evaluated unless you added specific declarations. Therefore, the rewrite for `N!` does not apply.

Warning 8. Guards are only implemented in optimized mode, which is not fully functional yet.

3.1.6 Sequences

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

Figure 33 for instance guarantees that the code will first `write "A"`, then `write "B"`, then write the result of the sum `f(100)+f(200)`. However, the implementation is entirely free to compute `f(100)` and `f(200)` in any order, including in parallel.

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

Figure 33. Code writing `A`, then `B`, then `f(100)+f(200)`

Items in a sequence can be *declarations* or *statements*. Declarations include rewrite declarations, data declarations, type declarations and assignments to a type declaration. All other items in a sequence are statements.

3.1.7 Index operators

The notation `A[B]` and `A.B` are used as index operators, i.e. to refer to individual items in a collection. The `A[B]` notation is intended to represent array indexing operations, whereas the `A.B` notation is intended to represent field indexing operations.

For example, consider the declarations in Figure 34.

```
MyData ->
    Name  -> "Name of my data"
    Value -> 3.45
    1 -> "First"
    2 -> "Second"
    3 -> "Third"
```

Figure 34. Structured data

In that case, the expression `MyData.Name` results in the value `"Name of my data"`. The expression `MyData[1]` results in the value `"First"`.

The two index operators differ when their right operand is a name. The notation `A.B` evaluates `B` in the context of `A`, whereas `A[B]` first evaluates `B` in the current context, and then applies `A` to it (it is actually nothing more than a regular tree rewrite). Therefore, the notation `MyData.Value` returns the value `3.45`, whereas the value of `MyData[Value]` will evaluate `Value` in the current context, and then apply `MyData` to the result. For example, if we had `Value->3` in the current context, then `MyData[Value]` would evaluate to `"Third"`.

Warning 9. Index operators are only partially implemented. They work for simple examples, but may fail for more complex use cases. In particular, it is not currently possible to update a context by writing to an indexed value.

Comparison with C Users familiar with languages such as C may be somewhat disconcerted by ELIOT's index operators. The following points are critical for properly understanding them:

- Arrays, structures and functions are all represented the same way. The entity called `MyData` can be interpreted as an array in `MyData[3]`, as a structure in `MyData.Name`, or as a function if one writes `MyData 3`. In reality, there is no difference between `MyData[3]` and `MyData 3`: the former simply passes a block as an argument, i.e. it is exactly equivalent to `MyData(3)`, `MyData{3}`. Writing `MyData[3]` is only a way to document an intent to use `MyData` as an array, but does not change the implementation.
- Data structures can be extended on the fly. For example, it is permitted to assign something to a non-existent binding in `MyData`, e.g. by writing `MyData[4]:=3`. The ability to add "fields" to a data structure on the fly makes it easier to extend existing code.
- Data structures can include other kinds of rewrites, for example "functions", enabling object-oriented data structures. This is demonstrated in Section 5.6.
- Since the notation `A.B` simply evaluates `B` in the context of `A`, the value of `MyData.4` is... `4`: there is no rewrite for `4` in `MyData`, therefore it evaluates a itself.

3.1.8 C interface

A *C interface* is is a rewrite where the implementation is a prefix of two names, the first one being `C` and the second one being the name of a C function. A C interface can also be specified using a special `extern` syntax. The name of the C function can also be specified as text if it does not obey ELIOT naming rules, e.g. to interface to a function named `_foobar_`.

Figure 35 shows two ways of making the `sin` function of the C standard library available to an ELIOT program. The first one uses an ELIOT-style rewrite, whereas the second one uses a C-style syntax:

```
sin X:real as real -> C sin
extern double sin(double);
```

Figure 35. Creating an interface for a C function

The C-like syntax used for `extern` declaration is defined by the file `C.syntax`, and applies for anything between delimiters `extern` and `;` as indicated in the `xl.syntax` file. While extremely simplistic relative to the real C syntax, it is sufficient to import most functions.

Table 1 shows which types can be used in a C interface and what C type they map to:

ELIOT type	C type
integer	int
real	double
text	const char *
tree	Tree *
infix	Infix *
prefix	Prefix *
postfix	Postfix *
block	Block *
name	Name *
boolean	bool

Table 1. Type correspondances in a C interface

Warning 10. The C interface syntax is only available in optimized mode.

3.1.9 Machine Interface

A *machine interface* is a rewrite where the implementation is a prefix of two names, the first one being `opcode`. Figure 36 shows how a specific tree rewrite can be connected to the generation of machine-level opcodes:

```
X:integer+Y:integer as integer -> opcode Add
```

Figure 36. Generating machine code using opcode declarations

Machine-level opcodes are provided by the LLVM library (http://llvm.org). Opcodes available to ELIOT programs are described in Section 6.14.

Warning 11. The machine-level interface is only available in optimized mode.

3.2 Binding References to Values

A *rewrite declaration* of the form `Pattern->Implementation` is said to *bind* its pattern to its implementation. A sequence of rewrite declarations is called a *context*. For example, the block `{x->3;y->4}` is a context that binds `x` to `3` and `y` to `4`.

Warning 12. The idea of formalizing the context and making it available to programs was only formalized after the standard and optimized mode were implemented. It is not currently working, but should be implemented in a future release. However, many notions described in this section apply internally to the existing implementations, i.e. the context order is substantially similar even if it is not made visible to programs in the way being described here.

3.2.1 Context Order

A context may contain multiple rewrites that hide one another.

For example, in the context `{x->0;x->1}`, the name `x` is bound twice. The evaluation of `x` in that context will return `0` because rewrites are tested in order. In other words, the declaration `x->0` *shadows* the declaration `x->1` in that context.

For the purpose of finding the first match, a context is traversed depth first in left-to-right order, which is called *context order*.

3.2.2 Scoping

The left child of a context is called the *local scope*. The right child of a context is the *enclosing context*. All other left children in the sequence are the local scopes of expressions currently being evaluated. The first one being the *enclosing scope* (i.e. the local scope of the enclosing context) and the last one being the *global scope*.

This ensures that local declarations hide declarations from the surrounding context, since they are on the left of the right child, while allowing local declarations in the left child of the context to be kept in program order, so that the later ones are shadowed by the earlier ones.

The child at the far right of a context is a catch-all rewrite intended to specify what happens when evaluating an undefined form.

3.2.3 Current context

Any evaluation in ELIOT is performed in a context called the *current context*. The current context is updated by the following operations:

1. Evaluating the implementation of a rewrite creates a scope binding all arguments to the corresponding parameters, then a new context with that scope as its left child and the old context as its right child. The implementation is then evaluated in the newly created context.
2. Evaluating a sequence initializes a local context with all declarations in that sequence, and creates a new current context with the newly created local context as its left child and the old context as its right child. Statements in the sequence are then evaluated in the newly created context.
3. Evaluating an assignment changes the implementation of an existing binding if there is one in the current context, or otherwise creates a new binding in the local scope.

3.2.4 References

An expression that can be placed on the left of an assignment to identify a particular binding is called a *reference*. A reference can be any pattern that would go on the left of a rewrite. In addition, it can be an index operator:

- If `A` refers to a context, assigning to `A.B` will affect the binding of `B` in the context `A`, and not the binding of `A.B` in the current context.
- If `A` refers to a context, assigning to `A[B]` (or, equivalently, to `A{B}`, `A(B)` or `A B`) will affect the binding corresponding to `B` in the context of `A`, not the binding of `A B` in the current context. The index `B` will be evaluated in the current context if required to match patterns in `A`, as explained in Section 3.3.
- Special forms described in Section 4.1.9, such as `left X` refer to children of `infix`, `prefix`, `postfix` or `block` trees. They can be used as a reference in an assignment, and will modify the tree being referred to. This can be used to directly manipulate the program structure.

3.3 Evaluation

Evaluation is the process through which a given tree is rewritten.

3.3.1 Standard evaluation

Except for special forms described later, the evaluation of ELIOT trees is performed as follows:

1. The tree to evaluate, `T`, is matched against the available data and rewrite pattern. `3*4+5` will match `A*B+C` as well as `A+B` (since the outermost tree is an infix `+` as in `A+B`).

2. Possible matches are tested in *context order* (defined in Section 3.2) against the tree to evaluate. The first matching tree is selected. For example, in Figure 1, `(N-1)!` will be matched against the rules `0!` and `N!` in this order.
3. Nodes in each candidate pattern `P` are compared to the tree `T` as follows:
   - Constant symbols or names in `P` are compared to the corresponding element in `T` and must match exactly. For example, the `+` symbol in pattern `A+B` will match the plus `+` symbol in expression `3*4+5` but not the minus `-` symbol in `3-5`.
   - Variables names in `P` that are not bound to any value and are not part of a type declaration are bound to the corresponding fragment of the tree in `T`. For example, for the expression `3!`, the variable `N` in Figure 1 will be bound to `3`.
   - Variable names in `P` that are bound to a value are compared to the corresponding tree fragment in `T` after evaluation. For instance, if `true` is bound at the point of the declaration in Figure 21, the test `if A<3 then X else Y` requires the evaluation of the expression `A<3`, and the result will be compared against `true`.
   - This rule applies even if the binding occured in the same pattern. For example, the pattern `A+A` will match `3+3` but not `3+4`, because `A` is first bound to `3` and then cannot match `4`. The pattern `A+A` will also match `(3-1)+(4-2)`: although `A` may first be bound to the unevaluated value `3-1`, verifying if the second `A` matches requires evaluating both `A` and the test value.
   - Type declarations in `P` match if the result of evaluating the corresponding fragment in `T` has the declared type, as defined in Section 3.4. In that case, the variable being declared is bound to the evaluated value.
   - Constant values (integer, real and text) in `P` are compared to the corresponding fragment of `T` after evaluation. For example, in Figure 1, when the expression `(N-1)!` is compared against `0!`, the expression `(N-1)` is evaluated in order to be compared to `0`.
   - Infix, prefix and postfix nodes in `P` are compared to the matching node in `T` by comparing their children in depth-first, left to right order.

   The comparison process, called *pattern matching*, may cause fragments of the tree to be evaluated. Each fragment is evaluated at most once for the process of evaluating the tree `T`. Once the fragment has been evaluated, the evaluated value will be *memoized* and used in any subsequent comparison or variable binding. For example, when computing `F(3)!`, the evaluation of `F(3)` is required in order to compare to `0!`, guaranteeing that `N` in `N!` will be bound to the evaluated value if `F(3)` is not equal to `0`.
4. If there is no match found between any pattern `P` and the tree to evaluate `T`:
   - Integer, real and text terminals evaluates as themselves.
   - A block evaluates as the result of evaluating its child.
   - If the tree is a prefix with the left being a name containing `error`, then the program immediatly aborts and shows the offending tree. This case corresponds to an unhandled error.
   - For a prefix node or postfix tree, the operator child (i.e. the left child for prefix and the right child for postfix) is evaluated, and if the result is different from the original operator child, evaluation is attempted again after replacing the original operator child with its evaluated version.
   - In any other case, the tree is prefixed with `evaluation_error` and the result is evaluated. For example, `$foo` will be transformed into `evaluation_error $foo`. A prefix rewrite for `evaluation_error` is supposed to handle such errors.
5. If a match is found, variables in the first matching pattern (called *parameters*) are bound to the corresponding fragments of the tree to evaluate (called *arguments*).
   - If an argument was evaluated (including as required for comparison with an earlier pattern), then the corresponding parameter is bound with the evaluated version.

- If the argument was not evaluated, the corresponding parameter is bound with the tree fragment in context, as explained in Section 3.2. In line with the terminology used in functional languages, this context-including binding is called a *closure*.

6. Once all bindings have been performed, the implementation corresponding to the pattern in the previous step is itself evaluated. The result of the evaluation of the original form is the result of evaluating the implementation in the new context created by adding to the original context the bindings of parameters to their arguments. For a data form, the result of evaluation is the pattern after replacing parameters with the corresponding arguments.

3.3.2 Special forms

Some forms have a special meaning and are evaluated specially:

1. A terminal node (integer, real, type, name) evaluates as itself, unless there is an explicit rewrite rule for it[8].
2. A block evaluate as the result of evaluating its child.
3. A rewrite rule evaluates as itself.
4. A data declaration evaluates as itself
5. An assignment binds the variable and evaluates as the named variable after assignment
6. Evaluating a sequence creates a new local context with all declarations in the sequence, then evaluates all its statements in order in that new local context. The result of evaluation is the result of the last statement, if there is one, or the newly created context if the sequence only contains declarations.
7. If `C` is a context and `E` is an expression, evaluating `C E` is equivalent to evaluating `E` in the current context, then evaluating the result in the context of `C`. For example, `(0->3)(1-1)` will evaluate `1-1`, resulting in `0`, then evaluate the result in the context `0->3`, resulting in the value `3`.
8. If `C` is a context and `E` is an expression, evaluating `C.E` is equivalent to evaluating `E` in the context of `C`. For example, `(foo->1;bar->2).bar` will return `2`.

3.3.3 Lazy evaluation

When an argument is bound to a parameter, it is associated to a context which allows correct evaluation at a later time, but the argument is in general not evaluated immediately. Instead, it is only evaluated when evaluation becomes necessary for the program to execute correctly. This technique is called *lazy evaluation*. It is intended to minimize unnecessary evaluations.

Evaluation of an argument before binding it to its parameter occurs in the following cases, collectively called *demand-based evaluation*:

1. The argument is compared to a constant value or bound name, see Section 3.3.1, and the static value of the tree is not sufficient to perform the comparison. For example, in Figure 37, the expression `4+X` requires evaluation of `X` for comparison with `4` to check if it matches `A+A`; the expression `B+B` can be statically bound to the form `A+A` without requiring evaluation of `B`; finally, in `B+C`, both `B` and `C` need to be evaluated to compare if they are equal and if the form `A+A` matches.

```
A+A -> 2*A
4+X  // X evaluated
B+B  // B not evaluated
B+C  // B and C evaluated
```

Figure 37. Evaluation for comparison

2. The argument is tested against a parameter with a type declaration, and the static type of the tree is not sufficient to guarantee a match. For example, in Figure 38, the expression `Z+1` can statically be found to match the form `X+Y`, so `Z` needn't be evaluated. On the other hand, in `1+Z`, it is necessary to evaluate `Z` to type-check it against `integer`.

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

```
X:tree + Y:integer -> ...
Z + 1 // Z not evaluated
1 + Z // Z evaluated
```

Figure 38. Evaluation for type comparison

3. A specific case of the above scenario is the left side of any index operator. In `A.B` or `A[B]`, the value `A` needs to be evaluated to verify if it contains `B`.

When lazy evaluation happens, the expression being bound is a closure as explained in Section 3.3.1, i.e. it will be an expression of the form `C.E` where `C` is the original evaluation context and `E` is the original expression to evaluate.

Warning 13. Lazy evaluation was formalized after the compilers were implemented, and is not entirely consistent in the current implementations. This should be fixed in future versions.

3.3.4 Explicit evaluation

Expressions are also evaluated in the following cases, collectively called *explicit evaluation*:

1. An expression on the left or right of a sequence is evaluated. For example, in `A;B`, the names `A` and `B` will be evaluated.
2. The prefix `do` forces evaluation of its argument. For example, `do X` will force the evaluation of `X`.
3. The program itself is evaluated. Most useful programs are sequences.

The explicit evaluation of a name does not change the value bound to that name in the current context. For example, if the current context contains `A->write "Hello"`, each explicit evaluation of `A` will cause the message `Hello` to be written.

3.3.5 Memoization

Whenever a parameter is evaluated, the evaluated result may be used for all subsequent demand-based evaluations of the same tree, a process called *memoization*. What is memoized is associated with the original tree.

Memoization does not happen for explicit evaluations. This is illustrated with the example in Figure 39:

```
foo X ->
    X
    if X then writeln "X is true"
    if do X then writeln "X is true"
    X
bar ->
    writeln "bar evaluated"
    true
foo bar
```

Figure 39. Explicit vs. lazy evaluation

In Figure 39, evaluation happens as follows:

1. The expression `foo bar` is evaluated explicitly, being part of a sequence. This matches the rewrite for `foo X` on the first line.
2. The first reference to `X` in the implementation is evaluated explicitly. This causes the message bar evaluated to be written to the console.
3. The second reference to `X` is demand-based, but since `X` has already been evaluated, the result `true` is used directly. The message X is true is emitted on the console, but the message bar evaluated is not.
4. The third reference to `X` is an argument to `do`, so it is evaluated again, which writes the message bar evaluated on the console.

5. The last reference to `X` is another explicit evaluation, so the message bar evaluated is written on the console again.

The purpose of these rules is to allow the programmer to pass code to be evaluated as an argument, while at the same time minimizing the number of repeated evaluations when a parameter is used for its value. In explicit evaluation, the value of the parameter is not used, making it clear that what matters is the effect of evaluation itself. In demand-based evaluation, it is on the contrary assumed that what matters is the result of the evaluation, not the process of evaluating it. It is always possible to force evaluation explicitly using `do`.

Warning 14. Like lazy evaluation, memoization is not fully consistent in the current implementations.

3.4 Types

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

Warning 15. Like contexts, the type system was largely redesigned based on experience with the first implementations of the language. As a result, the current implementations implement a very weak type system compared to what is being described in this section. At this point, user-defined types do not work as descried in either the standard or optimized implementation. This section defines the future implementation.

3.4.1 Predefined types

The following types are predefined:

- `integer` matches integer values
- `real` matches real values
- `text` matches text values
- `symbol` matches names and operator symbols
- `name` matches names only
- `operator` matches operator symbols only
- `infix` matches infix nodes
- `prefix` matches prefix nodes
- `postfix` matches postfix nodes
- `block` matches block nodes
- `tree` matches any abstract syntax tree, i.e. any representable XL value
- `boolean` matches the names `true` and `false`.

3.4.2 Type definition

A *type definition* for type `T` is a special form of tree rewrite declaration where the pattern has the form `type X`. A type definition declares a type name, and the pattern that the type must match. For example, Figure 40 declares a type named `complex` requiring two real numbers called `re` and `im`, and another type named `ifte` that contains three arbitrary trees called `Cond`, `TrueC` and `FalseC`.

```
complex -> type (re:real, im:real)
ifte -> type {if Cond then TrueC else FalseC}
```

Figure 40. Simple type declaration

The outermost block of a type pattern, if it exists, is not part of the type pattern. To create a type matching a specific block shape, two nested bocks are required, as illustrated with `paren_block_type` in Figure 41:

```
paren_block_type -> type((BlockChild))
```

Figure 41. Simple type declaration

Note that type definitions and type declarations should not be confused. A type *definition* defines a type and has the form `Name -> type TypePattern`, whereas a type *declaration* declares the type of an entity and has the form `Name:Type`. The type defined by a type definition can be used on the right of a type declaration. For example, Figure 42 shows how to use the `complex` type defined in Figure 40 in parameters.

```
Z1:complex+Z2:complex -> (Z1.re+Z2.re, Z1.im+Z2.im)
```

Figure 42. Using the `complex` type

Parameters of types such as `complex` are bound to contexts with declarations for the individual variables of the pattern of the type. For example, a `complex` like `Z1` in Figure 42 contains a rewrite for `re` and a rewrite for `im`. Figure 43 possible bindings when using the complex addition operator defined in Figure 42. The standard index notation described in Section 3.1.7 applies, e.g. in `Z1.re`, and these bindings can be assigned to.

```
// Expression being evaluated
(3.4, 5.2)+(0.4, 2.22)

// Possible resulting bindings
// in the implementation of +
Z1 ->
    re->3.4
    im->5.2
    re, im
Z2 ->
    re->0.4
    im->2.22
    re, im
```

Figure 43. Binding for a `complex` parameter

Figure 44 shows two ways to make type `A` equivalent to type `B`:

```
A -> B
A -> type X:B
```

Figure 44. Making type `A` equivalent to type `B`

3.4.3 Normal form for a type

By default, the name of a type is not part of the pattern being recognized. It is often recommended to make data types easier to identify by making the pattern more specific, for instance by including the type name in the pattern itself, as shown in Figure 45:

```
complex -> type complex(re:real, im:real)
```

Figure 45. Named patterns for `complex`

In general, multiple notations for a same type can coexist. In that case, it is necessary to define a form for trees that the other possible forms will reduce to. This form is called the *normal form*. This is illustrated in Figure 46, where the normal form is `complex(re;im)` and the other notations are rewritten to this normal form for convenience.

```
// Normal form for the complex type
complex -> type complex(re:real, im:real)

// Other possible notations that reduce to the normal form
i -> complex(0,1)
A:real + i*B:real -> complex(A,B)
A:real + B:real*i -> complex(A,B)
```

Figure 46. Creating a normal form for the complex type

3.4.4 Properties

Properties are types that match a number of trees, based not just on the shape of the tree, but on symbols bound in that tree. For instance, when you need a `color` type representing red, green and blue components, you care about the value of the components, but not the order in which they appear.

A *properties definition* is a rewrite declaration like the one shown in Figure 47 where:

1. The implementation is a prefix with the name `properties` followed by a block.
2. The block contains a sequence of type declarations `Name:Type` or assignments to type declarations `Name:Type:=DefaultValue`, each such statement being called a *property*.
3. The block optionally contains one or more `inherit` prefix (see Section 3.4.5)

```
color -> properties
    red   : real
    green : real
    blue  : real
    alpha : real := 1.0
```

Figure 47. Properties declaration

Properties parameters match any block for which all the properties are defined. Properties are defined either if they exist in the argument's context, or if they are explicitly set in the block argument, or if a *default value* was assigned to the property in the properties declaration. An individual property can be set using an assignment or by using the property name as a prefix.

For example, Figure 48 shows how the `color` type defined in Figure 47 can be used in a parameter declaration, and how a `color` argument can be passed.

```
write C:color ->
    write "R", C.red
    write "G", C.green
    write "B", C.blue
    write "A", C.alpha
write_color { red 0.5; green 0.2; blue 0.6 }
```

Figure 48. Color properties

Properties parameters are contexts containing local declarations called *getters* and *setters* for each individual property:

- The setter is a prefix taking an argument of the property's type, and setting the property's value to its argument.
- The getter returns the value of the property in the argument's context (which may be actually set in the argument's enclosing contexts), or the default value if the property is not bound in the argument's context.

This makes it possible to set default value in the caller's context, which will be injected in the argument, as illustrated in Figure 49, where the expression `C.red` in `write_color` will evaluate to `0.5`, and the argument `C.alpha` will evaluate to `1.0` as specified by the default value:

```
red := 0.5
write_color (blue 0.6; green 0.2)
```

Figure 49. Setting default arguments from the current context

It is sufficient for the block argument to define all required properties. The block argument may also contain more code than just the references to the setters, as illustrated in Figure 50:

```
write_color
    X:real := 0.444 * sin time
    if X < 0 then X := 1.0+X
    red X
    green X^2
    blue X^3
```

Figure 50. Additional code in properties

3.4.5 Data inheritance

Properties declarations may *inherit* data from one or more other types by using one or more `inherit` prefixes in the properties declaration, as illustrated in Figure 51, where the type `rgb` contains three properties called `red`, `green` and `blue`, and the type `rgba` additionally contains an `alpha` property:

```
rgb -> properties
    red   : real
    green : real
    blue  : real
rgba -> properties
    inherit rgb
    alpha : real
```

Figure 51. Data inheritance

Only declarations are inherited in this manner. The resulting types are not compatible, although they can be made compatible using automatic type conversions (see Section 3.4.7).

3.4.6 Explicit type check

Internally, a type is any context where a `contains` prefix can be evaluated. In such a context, the expression `contains X` is called a *type check* for the type and for value `X`. A type check must return a `boolean` value to indicate if the value `X` belongs to the given type.

Type checks can be declared explicitly to create types identifying arbitrary forms of trees that would be otherwise difficult to specify. This is illustrated in Figure 52 where we define an `odd` type that contains only odd integers and the text `"Odd"`. We could similarly add a type check to the definition of `rgb` in Figure 51 to make sure that `red`, `green` and `blue` are between `0.0` and `1.0`.

```
odd ->
    contains X:integer -> X mod 2 = 1
    contains "Odd" -> true
    contains X -> false
```

Figure 52. Defining a type identifying an arbitrary AST shape

The type check for a type can be invoked explicitly using the infix `contains` (with the type on the left) or `is_a` (with the type on the right). This is shown in Figure 53. The first type check `odd contains 3` should return `true`, since `3` belongs to the `odd` type. The second type check should return `false` since `rgb` expects the property `blue` to be set.

```
if odd contains 3 then pass else fail
if (red 1; green 1) is_a rgb then fail else pass
```

Figure 53. Explicit type check

3.4.7 Explicit and automatic type conversions

Prefix forms with the same name as a type can be provided to make it easy to convert values to type `T`. Such forms are called *explicit type conversions*. This is illustrated in Figure 54:

```
rgba C:rgb  -> (red C.red; green C.green; blue C.blue; alpha 1.0)
rgb  C:rgba -> (red C.red; green C.green; blue C.blue)
```

Figure 54. Explicit type conversion

An *automatic type conversion* is an infix `as` form with a type on the right. If such a form exists, it can be invoked to automatically convert a value to the type on the right of `as`. This is illustrated in Figure 55.

```
X:integer as real -> real X
1+1.5   // 1.0+1.5 using conversion above
```

Figure 55. Automatic type conversion

3.4.8 Parameterized types

Since type definitions are just regular rewrites, a type definition may contain a more complex pattern on the left of the rewrite. This is illustrated in Figure 56, where we define a `one_modulo N` type that generalizes the `odd` type.

```
one_modulo N:integer ->
    contains X:integer -> X mod N = 1
    contains X -> false
show X:(one_modulo 1)
```

Figure 56. Parameterized type

It is also possible to define tree forms that are neither name nor prefix. Figure 57 shows how we can use an infix form with the `..` operator to declare a range type.

```
Low:integer..High:integer ->
    contains X:integer -> X>=Low and X<=High
    contains X -> false
foo X:1..5 -> write X
```

Figure 57. Declaring a range type using an infix form

3.4.9 Rewrite types

The infix `->` operator can be used in a type definition to identify specific forms of rewrites that perform a particular kind of tree transformation. Figure 58 illustrates this usage to declare an `adder` type that will only match rewrites declaring an infix `+` node:

```
adder -> type {X+Y -> Z}
```

Figure 58. Declaration of a rewrite type

4 Standard XL library

The ELIOT language is intentionally very simple, with a strong focus on how to extend it rather than on built-in features. Most features that would be considered fundamental in other languages are implemented in the library in ELIOT. Implementing basic amenities that way is an important proof point to validate the initial design objective, extensibility of the language.

Warning 16. This describes the standard XL library for the core, text-only implementation of XL found in the open-source implementation. Since there is no real difference between built-in functions and library definitions, the XL language can be "embedded" in an application that will provide a much richer vocabulary. In particular, users of Tao Presentations should refer to the Tao Presentations on-line documentation for information about features specific to this product, such as 3D graphics, regular expressions, networking, etc.

4.1 Built-in operations

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

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

4.1.1 Arithmetic

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

Form	Description
`x+y`	Addition
`x-y`	Subtraction
`x*y`	Multiplication
`x/y`	Division
`x rem y`	Remainder
`x mod y`	Modulo
`x^y`	Power
`-x`	Negation
`x%`	Percentage (`x/100.0`)
`x!`	Factorial

Table 2. Arithmetic operations

4.1.2 Comparison

Comparison operators can take `integer`, `real` or `text` argument, both arguments being of the same type, and return a `boolean` argument, which can be either `true` or `false`. Text is compared using the lexicographic order[9].

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

Table 3. Comparisons

4.1.3 Bitwise arithmetic

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

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

Table 4. Bitwise arithmetic operations

4.1.4 Boolean operations

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

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

Table 5. Boolean operations

4.1.5 Mathematical functions

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

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

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

Table 6. Mathematical operations

4.1.6 Text functions

Text functions operate on `text` values.

Form	Description
`x&y`	Concatenation
`text_length x`	Length of the text
`text_range t, start, len`	Range of characters in `t`
`t[n]`	Character at index `n`
`t[n1..n2]`	Characters in range `n1..n2`

Table 7. Text operations

The first character in a text is numbered `0`.

4.1.7 Conversions

Conversions operations transform data from one type to another.

Form	Description
`real x:integer`	Convert integer to real
`real x:text`	Convert text to real
`integer x:real`	Convert real to integer
`integer x:text`	Convert text to real
`text x:integer`	Convert integer to text
`text x:real`	Convert real to text
`text n:name`	Convert name to text
`name t:text`	Convert text to name

Table 8. Conversions

A conversion from text that fails returns the value `0`. Conversions to text always use the format used for ELIOT source code, using dot as a decimal separator: `text 0.0` is `"0.0"`.

4.1.8 Date and time

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

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

Table 9. Date and time

4.1.9 Tree operations

Tree operations allow direct manipulation of abstract syntax trees.

Form	Description
`identity x`	Returns `x`
`do x`	Forces explicit evaluation of `x`
`x.y`	Evaluate `y` in context of `x`
`self`	The input form in a rewrite implementation
`left X`, `right X`	Left and right child for infix, prefix, postfix
`child X`	Child of a block
`symbol X`	Symbol for an infix or name as text
`opening X`, `closing X`	Opening and closing of text or blocks

Table 10. Tree operations

The prefix `left`, `right`, `child`, `symbol`, `opening` and `closing` can be assigned to, as described in Section 3.1.4.

4.1.10 List operations, map, reduce and filter

By convention, ELIOT lists use comma-separated lists, such as `1,3,5,6`, although similar operations can be built with any other data structure. The map, reduce and filter operations act on such lists. They also can take a range `Low..High` as input. An empty list is represented by the name `nil`. Basic list operations are shown in Table 11:

Form	Description
`nil`	The empty list
`head,tail`	A data form for lists
`length L`	The length of list `L`
`map F L`	Map function `F` to list `L`
`reduce F L`	Combine list elements in a single value
`filter F L`	Filter elements of a list
`x with y`	Convenience notation for Map, Reduce or Filter
`x..y`	Create a range of elements between `x` and `y`
`head L` or `L.head`	Head of the list
`tail L` or `L.tail`	Tail of the list (all but first element)
`L1 & L2`	Concatenation of lists

Table 11. List operations

The *map* operation builds a list by applying the first argument as a prefix to each element of the list in turn. For example, `map foo (1,3,5)` returns the list `foo 1, foo 3, foo 5`. Map can be used with anonymous functions: `map (x->x+1) (2,4,6)` returns `(2+1,4+1,6+1)`.

The *reduce* operation, sometimes called *fold* or *accumulate* in other functional languages, combines elements of the list two by two using a binary operation, and returns a single result. For example, `reduce (x,y->x+y) (1,3,5)` returns `1+3+5`.

The *filter* operation takes a predicate (i.e. a function taking a single argument and returning a `boolean`) and a list, and returns elements of the list for which the predicate returns `true`. For example, `filter (x->x<10) (1,12,17,2)` returns `(1,2)`.

The notation `(X where Predicate X) with L` corresponds to a filter operation on list `L` with predicate `Predicate`. For example, `(X where X<10) with (1,12,17,2)` returns `(1,2)`.

The notation `(X,Y -> ...) with L` corresponds to a reduce operation on list `L`. For example, `(X,Y -> X+Y) with (2,4,6)` returns `(2+1,4+1,6+1)`.

For other forms of `F`, the notation `F with L` corresponds to a map operation on list `L`. For example, `sin with (1,3,5)` returns `sin 1, sin 3, sin 5`.

The notation `x..y` is called a *range*. A range of integer, real numbers or text can be used as a type. A range of integers can also be used as a lazy enumeration of all elements as a comma-separated list. In other words, `1..5` is a short-hand notation for `1,2,3,4,5`.

4.2 Control structures

Control structures such as tests and loops are implemented in the ELIOT standard library.

Warning 17. The control structures described below are not necessarily all implemented at all optimization levels. Future implementations will add new control structures as soon as the compiler becomes smart enough to generate correct code for the definitions given in this section.

4.2.1 Tests

The defintion of the if-then-else statement in the library is as shown in Figure 59:

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

Figure 59. Library definition of if-then-else

This definition requires the value to be a `boolean`, i.e. `true` or `false`. The `good` function shown in Figure 60 provides a behavior closer to what is seen in languages such as C, where the value `0` is logically false and non-zero values are logically true.

```
good false -> false
good 0     -> false
good 0.0   -> false
good ""    -> false
good nil   -> false
good Other -> true
```

Figure 60. The `good` function

It is possible to add declarations of `good` for other data types. Such local declarations will precede the declarations for `good` in scoping order, so that they override that "default" implementation of `good` shown above.

4.2.2 Infinite Loops

The ELIOT standard library provides a number of loop constructs. Figure 4.2.3 shows an implementation for the simplest form of loop, the infinite loop. The repeated evaluation of `Body` illustrates the importance of explicit evaluation (see Section 3.3.4). Note that such a recursive implementation is only efficient if tail recursion optimization works correctly (see Section 6.12).

```
loop Body ->
    Body
    loop Body
```

Figure 61. Infinite loop

4.2.3 Conditional Loops (`while` and `until` loops)

Figure 62 shows an implementation for the `while` loop, which runs while a given condition is true. Explicit evaluation is not required for `Condition` because it is evaluated only once in the implementation of the `while` loop, preventing memoization of `Condition` (see Section 3.3.3). The parameter `Condition` is not given a `boolean` type because we want the expression, not the result of evaluating that expression.

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

Figure 62. While loop

The `until` loop shown in Figure 4.2.4 is very similar to the `while` loop except that it stops when the condition becomes `true` instead of `false`.

```
until Condition loop Body ->
    if not Condition then
        Body
        until Condition loop Body
```

Figure 63. Until loop

4.2.4 Controlled Loops (`for` loops)

The `for` loop is the most complex kind of loop. It exists in multiple variants. The simplest one, shown in Figure 64, iterates over a range of `integer` values. Notice that it creates a local `Index` variable to ensure it doesn't modify `Variable` unless the loop is actually executed.

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

Figure 64. For loop on an integer range

The `for` loop shown in Figure 65 iterates on all elements of a container such as a list or a range. It updates its `Variable` for each iteration with a new element in the container.

```
for Variable in Container loop Body ->
    C : tree := Container
    while good C loop
        Variable := head C
        Body
        C := tail C
```

Figure 65. For loop on a container

There are several other kinds of `for` loops, corresponding to the patterns shown in Figure 66:

```
for Variable in Low:integer..High:integer step Step:integer
for Variable in Low:real..High:real
for Variable in Low:real..High:real step Step:real
```

Figure 66. Other kinds of `for` loop

It is not difficult to create custom for loops to explore other data structures.

Warning 18. The standard mode implements hard-coded `for` loops. The optimized mode is not currently powerful enough to handle `for` loop definitions properly.

4.2.5 Excursions

4.2.6 Error handling

4.3 Library-defined types

A variety of types are defined in the library.

4.3.1 Range and range types

The notation `low..high` defines a *range*. A range can be used as a list by list operations, as explained in Section 4.1.10, but also as a type. The range type `low..high` accepts all values between `low` and `high` included. It is defined in a way substantially equivalent to Figure 67:

```
low..high ->
    contains X -> X>=low and X<=high
    self
```

Figure 67. Range and range type definition

Arithmetic operations are also defined on ranges of `integer` and `real` numbers, and operate simultaneously on the `low` and `high` part of the range. When `low` and `high` are `real`, operations are performed with different rounding for `low` and `high`, so as to implement proper interval arithmetic.

Ranges of `integer` can also be interpreted as lists, with `head` and `tail` operations implemented in a way substantially similar to Figure 68. Lazy evaluation ensures that very large ranges can be processed efficiently (see Section 5.7.7).

```
head low:integer..high:integer -> if low <= high then low else nil
tail low:integer..high:integer -> if low <  high then low+1..high else nil
```

Figure 68. Ranges as lists

A test is required to deal with the corner case of empty lists.

Warning 19. The `range` type is not currently implemented, pending improvements in the type system.

4.3.2 Union types

The notation `A|B` in types is a *union type* for `A` and `B`, i.e. a type that can accept any element of types `A` or `B`. It is pre-defined in the standard library as in Figure 69:

```
A|B ->
    contains X:A -> true
    contains X:B -> true
    contains X -> false
```

Figure 69. Union type definition

Union types facilitate the definition of functions that work correctly on a multiplicity of data types, but not necessarily all of them, as shown in Figure 70:

```
number                -> type X:(integer|real)
succ X:number         -> X + 1
pred X:(integer|real) -> X-1
```

Figure 70. Using union types

Warning 20. Union types are not implementede yet, pending improvements in the type system.

4.3.3 Enumeration types

An *enumeration type* accepts names in a predefined set. The notation `enumeration(A, B, C)` corresponds to an enumeration accepting the names `A`, `B`, `C`... This notation is pre-defined in the standard library as in Figure 71:

```
enumeration(A:name,Rest) ->
    contains X:name -> text X = text A or enumeration(Rest) contains X
    contains X -> false
```

Figure 71. Enumeration type definition

Unlike in other languages, enumeration types are not distinct from one another and can overlap. For example, the name `do` belongs to `enumeration(do,undo,redo)` as well as to `enumeration(do,re,mi,fa,sol,la,si)`. Also, as this enumeration example demonstrates, enumerations can use names such as `do` that are also used by standard prefix functions.

Warning 21. Enumeration types are not implemented yet.

4.3.4 A type matching type declarations

Type declarations in a type definition are used to declare actual types, so a type that matches type declarations cannot be defined by a simple pattern. Figure 72 shows how the standard library defines a `type_declaration` type using a type check.

```
type type_declaration ->
   contains X:infix -> X.name = ":"
   contains X -> false
```

Figure 72. Type matching a type declaration

Warning 22. This definition of `type_declaration` does not work yet, pending improvements in the type system implementation.

4.4 Modules

ELIOT modules make it possible to decompose a large ELIOT program in smaller units.

4.4.1 Import statement

The `import` prefix imports a source file or a module, as shown in Figure 73:

```
import "file.xl"
import MyModule
import OtherModule 1.2
import MOD = LongMessage 1.3
```

Figure 73. Import statements examples

An import statement can be followed by a file name or a module specification:

- A file name provides a system-dependent file name for an ELIOT source file. By convention, ELIOT source file names end in `.xl`. In order to improve compatibility between systems, backslash characters `\` in file names are converted to slash characters `/` on Unix systems, and slash characters in file names are converted to backslash on Windows. Drive specifications such as `C:` are not converted.
- A module can also be identified by a name, optionally followed by a real number representing a minimum required version number. Modules source files are located in a set of directories defined by a *module path*, and contain special module declarations specifying the module import name and the version number.

- Finally, the module being imported can locally be given a short name with the syntax `import M=ModSpec`. In that case, the contents of the module is only visible using the index notation with either the short name `M` or the long module name.

Importing a module or file has the following effects:

1. Any `syntax` statement in the imported module applies to the source code importing it.
2. A scope is created and populated with all declarations in the module.
3. Except if a short name is given, that scope is placed immediately to the right of the current context. In other words, it potentially shadows previously imported modules, but also is potentially shadowed by declarations in the current file.
4. If the module is identified by a name and not a file name, a binding of that module name to the newly created module scope, and another binding to the short name in case one was provided.

Warning 23. In the current implementation, the `import` statement does not make the syntax visible yet.

The rationale for these rules is to make different usage scenarios equally convenient:

- If declarations in a module are going to be used extensively, using `import Module` makes all declarations visible by default.
- If a local declaration `Foo` hides a declaration of `Foo` in the module, it is still possible to refer to the module's declaration as `Module.Foo`.
- If it is undesirable to see declarations from the module, using `import M=Module` will prevent the module from becoming visible, but will make it convenient to refer to entities declared in the module using the short name, as in `M.Foo`.

4.4.2 Declaring a module

A module is identified by a module description similar to Figure 74:

```
module_description
    id "B1E18CF6-0E3E-4992-98AD-0FD998C9C9CB"
    name "My Incredible Module"
    description "This is an example of module"
    import_name "MyModule"
    author "John Doe"
    website "http://www.taodyne.com"
    url "git://git.taodyne.com/MyModule"
    dependencies BaseLibrary 1.1, ELIOT 0.9
    version 1.0
```

Figure 74. Module definition

The module description contains information allowing the ELIOT compiler to identify the modules. Only the `import_name` is required for that purpose. It is however considered good practice to provide the rest of the information, which can be used by various applications to provide meaningful information to the user, or useful utilities such as module dependency management.

5 Example code

5.1 Minimum and maximum

The minumum and maximum can be defined as follows:

```
min x, y -> m:tree := min y; if m < x then m else x
min x    -> x
max x, y -> m:tree := max y; if m > x then m else x
max x    -> x
```

Figure 75. Computing a minimum and a maximum

The functions as defined will work with any number of arguments, as well as with lists of items.

5.2 Complex numbers

5.3 Vector and Matrix computations

5.4 Linked lists with dynamic allocation

5.5 Input / Output

5.6 Object-Oriented Programming

5.6.1 Classes

5.6.2 Methods

5.6.3 Dynamic dispatch

5.6.4 Polymorphism

5.6.5 Inheritance

5.6.6 Multi-methods

5.6.7 Object prototypes

5.7 Functional-Programming

5.7.1 Map

5.7.2 Reduce

5.7.3 Filter

5.7.4 Functions as first-class objects

5.7.5 Anonymous functions (Lambda)

5.7.6 Y-Combinator

5.7.7 Infinite data structures

Since arguments are evaluated lazily, the evaluation of one fragment of the form does not imply the evaluation of any other. This makes it possible to correctly evaluate infinite data structures, as illustrated in Figure 76.

```
integers_above N:integer -> N, integers_above N+1
head X,Y -> X
tail X,Y -> Y

// This computes 7 without evaluating integers_above 8
head tail tail tail integers_above 4
```

Figure 76. Lazy evaluation of an infinite list

6 Implementation notes

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

6.1 Lazy evaluation

6.2 Type inference

6.3 Built-in operations

6.4 Controlled compilation

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

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

Figure 77. Controlled compilation

Controlled compilation depends on low-level compilation primitives provided by the LLVM infrastructure[10], and assumes a good understanding of LLVM basic operations. Table 12 shows the correspondance between LLVM primitives and primitives that can be used during controlled compilation.

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

Table 12. LLVM operations

6.5 Tree representation

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

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

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

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

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

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

6.6 Evaluation of trees

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

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

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

6.7 Tree position

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

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

6.8 Actions on trees

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

6.9 Symbols

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

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

6.10 Evaluating trees

A tree is evaluated as follows:

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

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

6.11 Code generation for trees

Evaluation functions are functions with the signature shown in Figure 78:

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

Figure 78. Signature for rewrite code with two variables.

Unfortunately, the signature in Figure 78 is not valid in C or C++, so we need a lot of casting to achieve the desired effect.

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

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

6.11.1 Right side of a rewrite

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

For example, if a rewrite is `X+Y->foo X,Y`, then the `code` field for `foo X,Y` will have `X` as its second argument and `Y` as its third argument, as shown in Figure 78.

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

6.11.2 Closures

If a tree is passed as a `tree` argument to a function, then it is encapsulated in a *closure*. The intent is to capture the environment that the passed tree depends on. Therefore, the associated `code` will take additional arguments representing all the captured values. For instance, a closure for `write X,Y` that captures variables `X` and `Y` will have the signature shown in Figure 79:

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

Figure 79. Signature for rewrite code with two variables.

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

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

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

6.12 Tail recursion

6.13 Partial recompilation

6.14 Machine Interface

6.15 Machine Types and Normal Types

Table of contents

Index

List of figures

List of tables