SYNTAX AND SEMANTICS OF BABEL-17, V0.21 (DRAFT)

STEVEN OBUA

Contents

1. Introduction

The first question that someone who creates a new programming language will hear from others inevitably is: Why another programming language? Are there not already enough programming languages out there that you can pick from?

I always wanted a programming language

- in which I can express myself compactly and without jumping through too many hoops,
- that is easy to understand and to learn,

Date: December 7, 2010.

- in which purely functional programming is the default and not the exception,
- that supports exceptions,
- that has pattern matching,
- that has built-in support for laziness and concurrency,
- that has built-in support for purely functional lists, vectors, sets and maps,
- that seamlessly marries structured programming constructs like loops with purely functional programming,
- that supports objects, modules and data encapsulation,
- that has mature implementations for all major modern computing platforms and can (and should!) be used for real-world programming,
- that has a simple mechanized formal semantics,
- that is beautiful.

There is no such language out there. Therefore I decided to create one. When Babel-17 will have reached v1.0 then the above goals will have been achieved. The version described in this document is Babel-17 v0.21, so there is still some distance to go.

Babel-17 is not a radically new or revolutionary programming language. It picks those raisins I like out of languages like Standard ML, Scala, Alice ML, Java, Javascript, Erlang, Haskell, Lisp, Clojure and Mathematica, and tries to evolve them into a beautiful new design.

Babel-17 is a functional language. It is not a pure functional language because it contains sources of non-determinism like the ability to pick one of two values randomly. If you take these sources of non-determinism out of the language (there are currently only two such sources, **choose** and **random**), it becomes pure.

In Babel-17, every value has a type. These types are not checked statically, but only dynamically.

Babel-17 is also object-oriented in that many aspects of Babel-17 can be explained in terms of sending messages to objects.

The default evaluation order of Babel-17 is strict. But optional laziness is built into the heart of Babel-17, also, and so is concurrency.

Babel-17 has pattern matching and exception handling. Both are carefully integrated with each other and with the laziness and concurrency capabilities of Babel-17.

Furthermore, Babel-17 establishes the paradigm of *purely functional structured programming* via the notion of linear scope.

In this document I specify Babel-17 in a mostly informal style: formal notation is sometimes used when it supports clarity.

2. Reference Implementation and Bug Parade

At `www.babel-17.com` you will find a reference implementation of Babel-17 v0.21 that fully supports the language as described in this document. There you will also find a Netbeans plugin that has syntax and error-highlighting for Babel-17 programs. As you find your way through this document, it might be helpful to directly try out Babel-17 in Netbeans to see if the interpreter behaves as you would guess from this spec. If you find implausible behavior, email your findings to `bugparade@babel-17.com`. Each proper bug will be listed together with your name (unless you don't wish so; then don't give your name in the email; it's also OK not to give your real name but an alias) and date

of your discovery forever at `bugparade.babel-17.com`. You can also provide a link to your blog or whatever.

3. Lexical Matters

Babel-17 source code is always written UTF-8 encoded. If it ain't UTF-8, it ain't Babel-17.

Constructors are alphanumeric strings which can also contain underscores, and which must start with a capital letter. Identifiers are alphanumeric strings which can also contain underscores, and which must start with a non-capital letter. The distinction between two constructors or two identifiers is not sensitive to capitalization.

These are the keywords of Babel-17:

```
begin   end   object   with      if       then      else     elseif
while   for     do    choose   random    yield     match     case
  as    val     def     in    exception   lazy   concurrent  memoize
  to  downto   true   false     nil     infinity    force     this
```

Note that keywords are written always in lower case. For example, `bEGIN` is not a legal keyword, but it also isn't an identifier (because of case insensitivity). Also note that `Begin` and `BEGIN` denote the same legal constructor. When we talk about identifiers in the rest of the paper, we always mean identifiers which are not keywords.

You can write down decimal, hexadecimal, binary and octal numbers in Babel-17. The following notations all denote the same number:

```
15  0xF  0b1111  0o17
```

Babel-17 also knows the number **infinity**.

Strings start and end with a quotation mark. Between the quotation mark, any valid UTF-8 string is allowed that does not contain newline, backslash or quotation mark characters, for example:

```
"hello world"  "$1, 2%"  "1 <= 4! <= 5^2"
```

Newline, backslash and quotation mark characters can be written down in escaped form. Actually, any unicode character can be written down in its escaped form via the hexadecimal value of its codepoint:

String consisting only of a quotation mark	`"\""`
String consisting only of a backslash	`"\\"`
String consisting only of a line feed	`"\n"`
String consisting only of a carriage return	`"\r"`
String consisting only of line feed via 16-bit escaping	`"\u000A"`
String consisting only of line feed via 32-bit escaping	`"\U0000000A"`

Comments in Babel-17 are written between matching pairs of `/*` and `*/` and can span several lines. Currently, comments cannot be nested.

Finally, here are all ASCII special symbols that can occur in a Babel-17 program:

```
=  ==  !=  <  <=  >  >=  +  -  *  /  ^  ;
|  &  !  ++  --  **  //  ,  ::  ->  =>  ?  ...
(  )  [  ]  {  }  .
```

Table 1 lists all other Unicode symbols that have a meaning in Babel-17.

Table 1. Further Unicode Symbols

Unicode Hex Code	Display	ASCII Equivalent
2261	$\equiv$	`==`
2262	$\not\equiv$	`!=`
2264	$\leq$	`<=`
2265	$\geq$	`>=`
221E	∞	**infinity**
2227	$\wedge$	`&`
2228	$\vee$	`|`
00AC	$\neg$	`!`
2237	::	`::`
2192	$\rightarrow$	`->`
21D2	$\Rightarrow$	`=>`
2026	...	...

Table 2. Built-in Types of Babel-17

Type	Description
Integer	the type of arbitrary size integer numbers
Infinity	the type consisting of the values **infinity** and $-$**infinity**
Boolean	the type consisting of the values **true** and **false**
String	the type of valid Unicode strings
List	the type of lists
Vector	the type of vectors / tuples
Set	the type of sets
Map	the type of maps
CExpr	the type of constructed expressions
Object	the type of user-defined objects
Function	the type of functions
PersistentException	the type of persistent exceptions
DynamicException	the type of dynamic exceptions

4. Overview of Built-in Types

Each value in Babel-17 has a unique type. All built-in types are depicted in Table 2. There is currently no way to explicitly test the type of a value. There is also currently no way to annotate the type of a variable. Future versions of Babel-17 will include such capabilities.

5. Objects and Functions

All values in Babel-17 are objects. This means that you can send messages to them. Many values in Babel-17 are functions. This means that you can apply them to an argument, yielding another value. Note that for a value to be an object, it does not need to have type *Object*, and for a value to be a function, it does not need to have type *Function*.

The result of sending message m to object v is written as

$$v.m$$

Here m is an identifier. The special object **nil** does not respond to any message.

The result of applying function f to value x is written as

$$f\ x$$

Note that $f\ x$ is equivalent to

$$(f.\text{apply\_})\ x$$

Therefore any object that responds to the apply_ message can act as a function. In the above we could also leave out the brackets because sending messages binds stronger than function application.

Repeated function application associates to the left, therefore

$$f\ x\ y$$

is equivalent to $(fx)\ y$.

6. Exceptions

Exception handling in Babel-17 mimicks exception handling mechanisms like those which can be found in Java or Standard ML, while adhering at the same time to the functional paradigm.

There are two types of exceptions: *PersistentException* and *DynamicException*. The difference is that a *PersistentException* can be treated as part of any data structure, and is passed around just like any other value, while a *DynamicException* can never be part of a data structure and has special evaluation semantics.

Exceptions in Babel-17 are not that special at all but mostly just another type of value. Let us write *PersistentException* v for a *PersistentException* with parameter v, and *DynamicException* v for a *DynamicException* with parameter v. We also write *Exception* v for an exception with parameter v. The parameter v is a *non-exceptional* value; this means that v can be any Babel-17 value except one of type *DynamicException*. Note that a value of type *PersistentException* is therefore non-exceptional.

The following deterministic rules govern how exceptions behave with respect to sending of messages, function application, laziness and concurrency:

$$\begin{aligned}
\texttt{(}\textit{Exception}\ v\texttt{).}m &\rightsquigarrow \textit{DynamicException}\ v \\
\texttt{(}\textit{DynamicException}\ v\texttt{)}\ x &\rightsquigarrow \textit{DynamicException}\ v \\
f\ \texttt{(}\textit{DynamicException}\ v\texttt{)} &\rightsquigarrow \textit{DynamicException}\ v \text{ where } f \text{ is non-exceptional} \\
\texttt{(}\textit{PersistentException}\ v\texttt{)}\ g &\rightsquigarrow \textit{DynamicException}\ v \text{ where } g \text{ is non-exceptional} \\
\texttt{exception}\ v &\rightsquigarrow \textit{DynamicException}\ v \\
\texttt{lazy (}\textit{Exception}\ v\texttt{)} &\rightsquigarrow \textit{PersistentException}\ v \\
\texttt{concurrent (}\textit{Exception}\ v\texttt{)} &\rightsquigarrow \textit{PersistentException}\ v \\
\texttt{force}\ v &\rightsquigarrow v \text{ for all } v\text{, including exceptions}
\end{aligned}$$

Exceptions in Babel-17 are created with the expression **exception** v. Creating exceptions in Babel-17 corresponds to *raising* or *throwing* exceptions in other languages. There is no special construct for catching exceptions in Babel-17. Instead, catching an exception can be done via the **match** expression which will be described later.

In the next section, we will describe the **lazy** and **concurrent** expressions of Babel-17. They are the reason why exceptions are divided into dynamic and persistent ones.

7. Laziness and Concurrency

The default evaluation mechanism of Babel-17 is strict. This basically means that the arguments of a function are evaluated before the function is applied to them. Babel-17 has two constructs to change this default behaviour.

The expression

lazy e

is not evaluated until it is actually needed. When it is needed, it evaluates to whatever e evaluates to, with the exception of dynamic exceptions, which are converted to persistent ones.

The expression

concurrent e

also evaluates to whatever e evaluates to, again with the exception of dynamic exceptions which are converted to persistent ones. This evaluation will happen concurrently.

One could think that apart from obvious different performance characteristics, the expressions **lazy** e and **concurrent** e are equivalent. This is not so. If e is a non-terminating expression then, even if the value of **concurrent** e is never needed during program execution, it might still lead to a non-terminating program execution. In other words, the behaviour of **concurrent** e is unspecified for non-terminating e.

Sometimes you do not want to proceed until an expression has been completely evaluated so that it does not contain any lazy or concurrent subcomponents any more. In those situations you use the expression

force e

which evaluates to e. So semantically, **force** is just the identity function.

We mentioned before that lazy and concurrent expressions are the reason why exceptions are divided into dynamic and persistent ones. To motivate this, look at the expression

$$\text{fst}\ \ (0,\ \textbf{lazy}\ (1\ \textbf{div}\ 0))$$

Here the function fst is supposed to be a function that takes a pair and returns the first element of this pair. So what would above expression evaluate to? Obviously, fst does not need to know the value of the second element of the pair as it depends only on the first element, so above expression evaluates just to 0. Now, if **lazy** was semantically just the identity function, then we would have

$$\begin{aligned} 0 = \text{fst}\ \ (0,\ \textbf{lazy}\ (1\ \textbf{div}\ 0)) &= \text{fst}\ \ (0,\ 1\ \textbf{div}\ 0) = \text{fst}\ \ (0,\ \textbf{exception}\ \text{DomainError}) \\ &= \text{fst}\ \ (\textbf{exception}\ \text{DomainError}) = \textbf{exception}\ \text{DomainError} \end{aligned}$$

Obviously, 0 should not be the same as an exception, and therefore **lazy** cannot be the identity function, but converts dynamic exceptions into persistent ones. For a dynamic exception e the equation

$$(0,\ e) = e$$

holds. For a persistent exception e this equation does not hold, and therefore the above chain of equalities is broken.

8. Lists and Vectors

For $n \geq 0$, the expression

$$[e_1, \ldots, e_n]$$

denotes a *list* of n elements.

The expression

$$(e_1, \ldots, e_n)$$

denotes a *vector* of n elements, at least for $n \neq 1$. For $n = 1$, there is a problem with notation, though, because (e) is equivalent to e. Therefore there is the special notation $(e,)$ for vectors which consist of only one element.

The only difference between lists and vectors is that they have different performance characteristics for the possible operations on them. Lists behave like simply linked lists, and vectors behave like arrays. Note that all data structures in Babel-17 are immutable. Apart from performance characteristics, anywhere you can use a list, you can use a vector, and vice versa.

Another way of writing down lists is via the right-associative :: constructor:

$$h::t$$

Here h denotes the *head* of the list and t its *tail*. Actually, note that the expression $[e_1,\ldots,e_n]$ is just syntactic sugar for the expression

$$e_1::e_2::\ldots::e_n::[]$$

Dynamic exceptions cannot be part of a list but are propagated:

$$\texttt{(DynamicException } v\texttt{)::}t \quad \rightsquigarrow \quad \texttt{DynamicException } v$$

$$h\texttt{::(DynamicException } v\texttt{)} \quad \rightsquigarrow \quad \texttt{DynamicException } v \quad \text{where } h \text{ is non-exceptional}$$

Note that in cases where the tail t is neither a list nor a vector, we identify $h::t$ with $h::t::[]$.

9. CExprs

A *CExpr* is a constructor c together with a parameter p, written $c\ p$. It is allowed to leave out the parameter p, which then defaults to **nil**. For example, HELLO is equivalent to HELLO **nil**. A constructor c cannot have a dynamic exception as its parameter, therefore we have:

$$c\ \texttt{(DynamicException } v\texttt{)} \quad \rightsquigarrow \quad \texttt{DynamicException } v$$

10. Pattern Matching

Maybe the most powerful tool in Babel-17 is pattern matching. You use it in several places, most prominently in the **match** expression which has the following syntax:

match e
  case p_1 => b_1
  $\vdots$
  case p_n => b_n
end

TABLE 3. General Patterns

Syntax	Description
_	the underscore symbol matches anything but a dynamic exception
x	an identifier x matches anything but a dynamic exception and binds the matched expression to x
$(x$ as $p)$	matches p, and binds the successfully matched value to x; the match fails if p does not match or if the matched value is a dynamic exception
z	a number z, like 0 or 42 or -10 or **infinity** or $-$**infinity** , matches just that number z
$c\ p$	matches a *CExpr* with constructor c if the parameter of the *CExpr* matches p
s	a string s, like "hello", matches just that string s
(p)	same as p
$(p$ **if** $e)$	matches any non-exceptional value that matches p, but only if e evaluates to **true**; identifiers bound in p can be used in e
$(\mathbf{val}\ e)$	matches anynon-exceptional value which is equivalent to e
$(f\ ?\ p)$	here f is a function; f is applied to the value to be matched; the match succeeds if the result of the application is not an exception amd matches p
$(d\ ?\ p)$	here d is a not a function and not an exception; short for ((v => v.deconstruct_ d) ? p)
$(e\ ?)$	short for (e ? **true**)
$\{m_1 = p_1, \ldots, m_n = p_n\}$	matches an object that has exactly the messages m_1,.., m_n such that the message values match the given patterns
$\{m_1 = p_1, \ldots, m_n = p_n, \delta\}$	matches an object that has the messages m_1,.., m_n such that the message values match the given patterns
nil	matches the empty object
exception p	matches any exception such that its parameter matches p

Given a value e, Babel-17 tries to match it to the patterns p_1, p_2, ... and so on sequentially in that order. If p_i is the first pattern to match, then the result of **match** is given by the block expression b_i. If none of the pattern matches then there are two possible outcomes:

(1) If e is a dynamic exception, then the value of the match is just e.
(2) Otherwise the result is a dynamic exception with parameter NoMatch.

A few of the pattern constructs incorporate arbitrary value expressions. When these expressions raise exceptions, the whole pattern they are contained in is rendered non-matching.

So what does a pattern look like? Table 3 and Table 4 list all ways of building a pattern.

TABLE 4. Collection Patterns

Syntax	Description
$[p_1, \ldots, p_n]$	matches a list/vector consisting of $n \geq 0$ elements, such that element e_i of the list is matched by pattern p_i
$(p_1, \ldots, p_n)$	matches a vector/list consisting of $n = 0$ or $n \geq 2$ elements, such that element e_i of the list is matched by pattern p_i
$[p_1, \ldots, p_n, \delta]$	matches a list/vector consisting of at least $n \geq 1$ elements, such that the first n elements e_i of the list/vector are matched by the patterns p_i
$(p_1, \ldots, p_n, \delta)$	matches a vector/list consisting of at least $n \geq 1$ elements, such that the first n elements e_i of the vector/list are matched by the patterns p_i
$(p,)$	matches a vector/list consisting of a single element that matches the pattern p.
$(h::t)$	matches a non-empty list/vector such that h matches the head of the list/vector and t its tail
$\{p_1, \ldots, p_n\}$	see section 17
$\{p_1, \ldots, p_n\ , \delta\}$	see section 17
$\{q_1 -> p_1, \ldots, q_n -> p_n\}$	see section 17
$\{q_1 -> p_1, \ldots, q_n -> p_n, \delta\}$	see section 17
$\{->\}$	matches the empty map (see section 17)
(**for** $p_1, \ldots, p_n$ **end**)	see section 17
(**for** $p_1, \ldots, p_n, \delta$ **end**)	see section 17

In this table of pattern constructions we use the δ-pattern δ. This pattern stands for "the rest of the entity under consideration" and can be constructed by the following rules:

(1) The ellipsis ... is a δ-pattern that matches any rest.
(2) If δ is a δ-pattern, and x an identifier, then (x as δ) is a δ-pattern.
(3) If δ is a δ-pattern, and e an expression, then (δ **if** e) is a δ-pattern.

Note that pattern matching does not distinguish between vectors and lists. A pattern that looks like a vector can match a list, and vice versa.

11. Non-Determinism in Babel-17

One source of non-determinism in Babel-17 is *probabilistic* non-determism. The expression

random n

returns for an integer $n > 0$ a number between 0 and $n - 1$ such that each number between 0 and $n - 1$ has equal chance of being returned. If n is a dynamic exception, then this exception is propagated, if n is a non-exceptional value that is not an integer > 0 then the result is an exception with parameter DomainError.

The other source of non-determinism is choice. The expression

choose l

takes a non-empty collection l and returns a member of the collection. The choice operator makes it possible to optimize evaluation in certain cases. For example, in

choose (**concurrent** a, **concurrent** b)

the evaluator could choose the member that evaluates quicker.

Babel-17 has been designed such that if you take the above two constructs out of the language, it becomes purely functional. If in your semantics of Babel-17 you replace the concept of value by the concept of a probability distribution over values and a set of values, you might be able to view Babel-17 as a purely functional language even *including* **random** and **choose**.

12. Block Expressions

So far we have only looked at expressions. We briefly mentioned the term *block expressions* in the description of the **match** function, though. We will now introduce and explain block expressions.

Block expressions can be used in several places as defined by the Babel-17 grammar. For example, they can be used in a **match** expression to define the value that corresponds to a certain case. But block expressions can really be used just everywhere where a normal expression is allowed:

begin
  b
end

is a normal expression where b is a block expression. A block expression has the form

s_1
$\vdots$
s_n

where the s_i are *statements*. In a block expression both newlines and semicolons can be used to separate the statements from each other. The other means of separation are semicolons.

Statements are Babel-17's primary tool for introducing identifiers. There are several kinds of statements. Three of them will be introduced in this section.

First, there is the **val**-statement which has the following syntax:

val $p = e$

Here p is a pattern and e is an expression. Its meaning is that first e gets evaluated. If this results in a dynamic exception, then the result of the block expression that the **val**-statement is part of will be that dynamic exception. Otherwise, the result of evaluating e is matched to p. If the match is successful then all identifiers bound by the match can be used in later statements of the block expression. If the match fail, then the value of the containing block expression is the dynamic exception NoMatch.

Second, there is the **def**-statement which obeys the following syntax for defining the identifier id:

def $id\ arg = e$

TABLE 5. Legal and illegal definitions

val x = y	**def** x = y	**val** x = y	**def** x = y
val y = 0	**val** y = 0	**def** y = 0	**def** y = 0
illegal	*illegal*	*legal*	*legal*

The *arg* part is optional. If *arg* is present, then it must be a pattern. Let us call those definitions where *arg* is present a *function definition*, and those definitions where *arg* is not present a *simple definition*.

Per block expression and identifier *id* there can be either a single simple definition, or there can be several function definitions. If there are multiple function definitions for the same identifier in one block expression, then they are bundled in that order to form a single function.

The defining expressions in **def**-statements can recursively refer to the other identifiers defined by **def**-statements in the same block expression. This is the main difference between definitions via **def** and definitions via **val**. Only those **val**-identifiers are in scope that have been defined *previously*, but **def**-identifiers are in scope throughout the whole block expression. Table 5 exemplifies this rule.

Let us assume that a block expression contains multiple function definitions for the same identifier f:

def f $p_1 = e_1$
⋮
def f $p_n = e_n$

Then this is (almost) equivalent to

def f x =
  match x
    case p_1 => e_1
      ⋮
    case p_n => e_n
  end

where x is fresh for the p_i and e_i. The slight difference between the two notations is that arguments that match none of the patterns will result in a DomainError exception for the first notation, and in a NoMatch exception for the second.

While definitions only define a single entity per block and identifier, it is OK to have multiple **val**-statements for the same identifier in one block expression, for example like that:

val x = 1
val x = (x, x)
x

The above block expression evaluates to (1,1); later **val**-definitions overshadow earlier ones. But note that neither

val x = 1

```
def x = 1
```

nor

```
def x = 1
val x = 1
```

are legal.

Another difference between **val** and **def** is observed by the effects of non-determinism:

```
val x = random 2
(x, x)
```

will evaluate either to (0, 0) or to (1, 1). But

```
def x = random 2
(x, x)
```

can additionally also evaluate to (0, 1) or to (1, 0) because x is evaluated each time it is used.

Let us conclude this section by describing the third kind of statement. It has the form

yield e

where e is an expression. There is also an abbreviated form of the **yield**-statement which we have already used several times in this section:

e

The semantics of the **yield**-statement is that it appends a further value to the value of the block expression it is contained in. Block expressions have a value just like all other expressions in Babel-17. It is obtained by "concatenating" all values of the **yield**-statements in a block expression. The value of

```
begin
end
```

is the empty vector (). The value of

```
begin
  yield a
end
```

is a. The value of

```
begin
  yield a
  yield b
end
```

is the vector (a, b) of length 2 and so on.

In this section we have introduced the notions of block expressions and statements. Their full power will be revealed in later sections of this document.

13. Anonymous Functions

So far we have seen how to define named functions in Babel-17. Sometimes we do not need a name for a certain function, for example when the code that implements this function is actually just as easy to understand as any name for the function. We already have the tools for writing such nameless, or anonymous, functions:

```
begin
  def sqr x = x * x
  sqr
end
```

is an expression denoting the function that squares its argument. There is a shorter and equivalent way of writing down the above:

```
x => x * x
```

In general, the syntax is

p => e

where p is a pattern and e an expression. The above is equivalent to

```
begin
  def f p = e
  f
end
```

where f is fresh for p and e.

There is also a syntax for anonymous functions which allows for several cases:

```
(case p1 => b1
   ⋮
 case pn => bn)
```

is equivalent to

```
begin
  def f p1 = begin b1 end
    ⋮
  def f pn = begin bn end
  f
end
```

where f is fresh for the p_i and b_i.

14. Objects

Object expressions have the following syntax:

```
object
  s1
  ⋮
  sn
end
```

The s_i are arbitrary statements, with the exception that no **yield**-statements are allowed. Those s_i that are **def** statements define the set of messages that the object responds to.

Often you do not want to create objects from scratch but by modifying other already existing objects:

object + parents
  s_1
  $\vdots$
  s_n
end

The expression parents must evaluate to either a list, a vector or a set. The members of parents are considered to be the parents of the newly created object, in the order induced by the collection. The idea is that the created object not only understands the messages defined by the s_i, but also the messages of the parents. Messages defined via s_i shadow messages of the parents. The messages of an earlier parent shadow the messages of a later one. The parents must have been created via object expressions, too.

The keyword **this** has currently no meaning in Babel-17. This might change in future versions of Babel-17.

There is one more way to denote record-like objects:

$\{\ m_1 = v_1, \ldots, m_n = v_n\ \}$

This is equivalent to:

begin
  val $(w_1, \ldots, w_n) = (v_1, \ldots, v_n)$
  object
    def $m_1 = w_1$
    $\ldots$
    def $m_n = w_n$
  end
end

The empty object is denoted by

nil

which is equivalent to

object end

15. Boolean Operators

Babel-17 provides the usual boolean operators. They are just syntactic sugar for certain **match** expressions; the exact translations are given in Table 6. Babel-17 also has **if**-expressions with the following syntax:

if b_1 **then**
  e_1
elseif b_2 **then**
  e_2
  $\vdots$

TABLE 6. Boolean Operators

! a	a & b	a | b

match a
  case true =>
  false
  case false =>
  true
  case _ =>
  exception DomainError
end
	match a
	  case true =>
	  match b
	  case true =>
	  true
	  case false =>
	  false
	  case _ =>
	  exception DomainError
	  end
	  case false =>
	  false
	  case _ =>
	  exception DomainError
	end
		match a
		  case false =>
		  match b
		  case true =>
		  true
		  case false =>
		  false
		  case _ =>
		  exception DomainError
		  end
		  case true =>
		  true
		  case _ =>
		  exception DomainError
		end

elseif b_n **then**
  e_n
else
  e_{n+1}
end

The **elseif**-branches are optional. Also, they can be eliminated in the obvious manner via nesting, so that we only need to give the semantics for the expression

if b **then** e_1 **else** e_2 **end**

The meaning of above expression is defined to be

match b
  case true => e_1
  case false => e_2
  case _ => **exception** DomainError
end

Actually, the **else** branch is also optional. The notation

if b **then** e **end**

is shorthand for

if b **then** e **else end**

16. ORDER

Babel-17 has a built-in partial order $\leq$. All other relational operators are defined in terms of this partial order (Table 7). It is possible to chain relational operators like this:

$a <= b <= c > d \mathrel{!}= e$

TABLE 7. Relational Operators

Syntax	**Semantics**
$a >= b$	$b <= a$
$a == b$	$a <= b$ & $b <= a$
$a < b$	$a <= b$ & $!(a >= b)$
$a > b$	$a >= b$ & $!(a <= b)$
$a \mathrel{!}= b$	$!(a == b)$

Intuitively, the above means

$$a <= b \;\&\; b <= c \;\&\; c > d \;\&\; d \mathrel{!}= e\ .$$

Note that we always evaluate the operands of relational operators, even chained ones, only once. For example, the precise semantics of $a <= b <= c <= d <= e$ is

```
begin
  val t = a
  val u = b
  t <= u &
  begin
    val v = c
    u <= v &
    begin
      val w = d
      v <= w & w <= e
    end
  end
end
```

In the above, t, u, v and w are supposed to be fresh identifiers. Also note that if there are operands that are exceptions, then the result of a comparison is a dynamic exception with the same parameter as the first such operand (from left to right).

The partial order is defined by the following laws:

- **nil** $< -\infty < z < \infty < s < lv < cexpr < set < map < b$ holds for all integers z, strings s, lists and vectors lv, constructed expressions $cexpr$, sets set, maps map and booleans b
- booleans are totally ordered and **false** $<$ **true** holds
- integers are totally ordered in the obvious way
- $-\infty == -\infty$ and $\infty == \infty$ both hold
- strings are totally ordered via lexicographic ordering
- lists and vectors are partially ordered by the lexicographic ordering
- constructed expressions are partially ordered by representing them by the pair consisting of constructor and parameter
- $f <> g$ holds for all functions f and g
- sets are partially ordered by representing them by sorted lists that consist of the elements of the set

TABLE 8. Set Operations

S `+` e	set created by inserting the element e into the set S
S `-` e	set created by removing the element e from the set S
S `++` T	set created by inserting all elements of T into the set S
S `--` T	set created by removing all elements of T from the set S
S `*` T	cartesian product of the set S and the elements of T
S `**` T	set created by removing all elements from the set S which are not an element of T
S e	tests wether e is an element of S
S/f	set created by applying the function f to the elements of S

- maps are partially ordered by representing them by sorted lists that consist of the (key, value) pairs of the map
- let U consist of all user-defined objects (i.e., defined via **object**) that cannot receive the message `representative`; then we have **nil** $<=$ u $<$ $-\infty$ for all $u \in U$; furthermore U is partially ordered by considering its elements as sorted lists that consist of (message, value) pairs.
- let a be a user-defined object (i.e., defined via **object**) that has the message `representative`; let ra be the result of sending `representative` to a; then $a <= b$ is defined by $ra <= b$, and $b <= a$ is defined via $b <= ra$ for all b.

The order does not distinguish between lists and vectors. For example, we have

(1, 2, 3) == [1, 2, 3]

17. Sets and Maps

Sets and maps are built into Babel-17. For example, the set consisting of 3, 42 and ∞ can be written as

{42, ∞, 3}

Sets are always sorted. The sorting order is Babel-17's built-in partial order, and every set forms a totally ordered subdomain of this partial order. Future versions of Babel-17 might allow the set order to be different from the built-in partial order. Adding or removing an element e from a set S is only well-defined when the elements of S together with e are totally ordered by the partial order of S. The same holds for testing if an element is in a set. Table 8 shows the built-in operations for sets.

Maps map finitely many keys to values. For example, the map that maps 1 to 2 and 4 to 0 is written as

{1 -> 2, 4 -> 0}

The empty map is denoted by

{->}

Maps also have always a partial order associated with them, in the current version of Babel-17 this is always Babel-17's built-in partial order. Operations on maps are only

Table 9. Map Operations

M `+` (k, v)	map created from the map M by associating k with v
M `-` k	map created from the map M by removing the key k
M `++` N	map created from the map M by adding the key/value pairs that are element of N
M `--` S	map created from M by removing all keys that are elements or keys of S
M `*` N	map created by using the partial order of M; if (k_M, v_M) is contained in M and (k_N, v_N) is contained in N such that $k_M == k_N$, then $(k_M, (v_M, v_N))$ is contained in the new map
M `**` T	map created by removing all keys from the map M which are not in the set/map T
M k	returns the value v associated with k in M, or returns a dynamic exception with parameter `OutOfDomain` if no such value exists
$M\%f$	map created by applying the function f to the values of M; f takes a key/value pair and returns a value
M/f	map created by applying the function f to the values of M; f takes a value and returns a value
M.reorder p	map created by reordering M according to the partial order p; in case of duplicate keys, the greater of the keys relative to the order of M is kept

well-defined if all keys of the map together with all other involved keys are totally ordered by the associated order. Table 9 shows the built-in operations for maps.

18. API and Runtime Environment

Although we have already provided the specification of many operations for the built-in types of Babel-17, this document does not contain the complete API for all types yet. This API will be provided together with the reference implementation for Babel-17. Here is a quick preview of some of its features:

- Integers are arbitrary precision and support the usual operations, throwing exceptions in cases which are not well-defined.
- Division and modulo of integers are euclidean.
- Strings support the usual operations like concatenation; the basic unit is a code point, not a character. Advanced operations like the Unicode collation algorithm are not part of Babel-17 currently, but might be introduced in later versions of Babel-17.

This specification also does not contain anything about the runtime environment in which Babel-17 is supposed to run. Depending on the runtime environment, there may be additional libraries available to the programmer. The medium term goal is to have Babel-17 running on and interfacing with the following platforms:

- Java and Android

TABLE 10. Syntactic Sugar

Sugared	**Desugared**
a `+` b	a.plus_ b
a `-` b	a.minus_ b
`-` a	a.uminus_
a `*` b	a.times_ b
a `/` b	a.quotient_ b
a `div` b	a.div_ b
a `mod` b	a.mod_ b
a `^` b	a.pow_ b
a `++` b	a.plus__ b
a `--` b	a.minus__ b
a `**` b	a.times__ b
a `//` b	a.quotient__ b
a `to` b	a.to_ b
a `downto` b	a.downto_ b
f x	f.apply_ x

- iOS
- HTML5 / Javascript

19. Syntactic Sugar

One of the goals of Babel-17 is that Babel-17 code is easy to read and understand. I have found that allowing arbitrary user-specific extensions to the syntax of code are definitely not helping to achieve this goal. Nevertheless, a bare minimum of syntactic sugar is necessary to support the most basic conventions known from arithmetic; for example, one should be able to write `3 + 5` to denote the addition of the integer 3 and the integer 5.

The programmer can use some of this syntactic sugar when defining her own objects. For example, `3 + 5` is just syntactic sugar for

```
3.plus 5
```

Table 10 lists all syntactic sugar of Babel-17 that is also available to the programmer. The availability of syntactic sugar for function application means that you can let your own objects behave as if they were functions.

20. Memoization

Babel-17 supports *memoization*. In those places where **def**-statements can be used, **memoize**-statements can be used, also. A **memoize**-statement must always refer to **def**-statements in the same scope. Babel-17 differentiates two kinds of memoization, *strong* and *weak*. A **memoize**-statement has the following syntax:

memoize $ref_1 \ldots ref_n$

The ref_i are either of the form id to indicate strong memoization, or of the form (id) to signal weak memoization. In both cases id refers to the **def**-statement being memoized. As an example, here is the definition of the weakly memoized fibonacci-sequence:

```
memoize (fib)
def fib 0 = 0
def fib 1 = 1
def fib n = fib (n−1) + fib (n−2)
```

The difference between weak and strong memoization is that strong memoization guarantees that the memoized function will not be executed twice for the same argument; weak memoization does not make this guarantee, so that garbage collection can reclaim all or part of the memoization cache when it needs to. Note that all arguments to the same memoized function must be totally ordered by the built-in partial order, and that memoization can only be requested for definitions that take an argument.

21. Linear Scope

We extend now the syntax of **val**-statements such that it is actually legal to leave out the keyword **val** if all variables defined by it have already been defined previously in its surrounding *linear scope*. The following two expressions are equivalent:

```
begin                    begin
  val x = a                val x = a
  val y = b                val y = b
  val x = c                x = c
  d                        d
end                      end
```

Furthermore, **begin** ... **end** can not only be used as an expression, but also as a statement; the following expressions exemplify nested **begin** ... **end** statements:

begin
  val x = 1
  val y = 2
  begin
  val x = 3
  val y = 4 ∗ x
  end
  (x, y)
end
	begin
	  val x = 1
	  val y = 2
	  begin
	  val x = 3
	  y = 4 ∗ x
	  end
	  (x, y)
	end
		begin
		  val x = 1
		  val y = 2
		  begin
		  val x = 3
		  val y = 0
		  y = 4 ∗ x
		  end
		  (x, y)
		end

evaluates to $(1, 2)$	evaluates to $(1, 12)$	evaluates to $(1, 2)$

The above examples show that the effect of dropping the **val** in a **val** statement is that the binding of an identifier becomes visible at that level within the linear scope where it has last been introduced via a **val** statement that includes the **val** keyword or via a pattern match in a **match** or **def**.

The linear scope of a statement is defined to be the block expression b that directly contains the statement, together with the linear scope of the expression or statement

that contains b. The linear scope of an expression e is only defined when e is a *control expression* that appears in another **val** statement of the form

val $pat = e$

or

$pat = e$

Then the linear scope of e is defined as the linear scope of that **val** statement. These are all *control expressions* of Babel-17:

- **begin** ... **end**
- **if** ... **end**
- **match** ... **end**
- **for** ... **end**
- **while** ... **end**

The last two control expressions are loops and explained in the next section.

Here are three example code snippets that illustrate linear scope:

```
begin                  begin                  begin
  val x = 1              val x = 1              val x = 1
  val y =                val y = 3 *            val y = 3 *
    begin                  begin                  begin
      x = 2                  x = 2                  val x = 2
      x+x                    x+x                    x+x
    end                    end                    end
  (x, y)                 (x, y)                 (x, y)
end                    end                    end
```

evaluates to $(2, 4)$	illegal	evaluates to $(1, 12)$

Just as the **begin** ... **end** expression may be used as a statement, you can also use all other control expressions as statements, for example:

```
begin
  val x = random 2
  if x == 0 then
    x = 100
  else
    x = 200
  end
  x + x
end
```

This expression will evaluate either to 200 or to 400.

For **if**-statements the **else**-branch is optional, and **match**-statements do not throw a NoMatch exception if none of the patterns matches.

22. Yield

Control expressions always return a value. With the help of the **yield** statement and, optionally, *collector annotations*, the programmer has flexible control about how this value is computed. The syntax of a **yield** statement is simply

yield e

where e is an arbitrary expression. The semantics is that the value of the control expression is the collection of all the values that have been issued via **yield**. For example, the expression

```
begin
  yield 1
  val x = 2
  yield x*x
  yield 3
  10
end
```

evaluates to `(1, 4, 3, 10)`. Note how the normal return value of **begin** ... **end** just counts as an additional **yield**.

By default, all yielded values are collected in a list which collapses in case of only a single element. The programmer might want to deviate from this default and collect the yielded values differently, for example to get rid of the collapsing behavior. *Collector annotations* allow her to do just that. A collector annotation has the following syntax:

with c : e

where c is a *collector* and e is a control expression. A collector is any object that

- responds to the message `collector_close`,
- and responds to the message `collector_add` x with another collector.

Lists, sets and maps are built-in collectors which the programmer can use out-of-the-box; apart from that she can implement her own collectors, of course.

Here is an example where we use a set as a collector:

```
with {4} : begin
  yield 1
  yield 2
  yield 1
  10
end
```

Above expression evaluates to $\{1,\ 2,\ 4,\ 10\}$.

23. Loops

Unlike the other control expressions, *loops* become useful only in combination with *linear scope* and **yield**.

This is the syntax for the **while**-loop:

```
while c do
  b
end
```

Here c must evaluate to a boolean and b is a control block. For example, here is how you could code the euclidean algorithm for calculating the greatest common divisor using **while**:

```
def gcd (a,b) = begin
  while b != 0 do
    (a, b) = (b, a mod b)
  end
  a
end
```

There is also the **for**-loop. It has the following syntax:

```
for p in C do
  b
end
```

In the above p is a pattern, C is a *collection* C, and b is a control block. The idea is that above expression iterates through those elements of the *collection* C which match p; for each successfully matched element, b is executed.

An object C is a collection if it handles the message `iterate`

- by returning `()` if it represents the empty collection,
- by returning `(`e`,` C'`)` such that C' is also a collection otherwise.

Here is an example of a simple **for**-loop expression:

```
begin
  val s = [10, (5, 8), 7, (3,5)]
  with {->} : for (a,b) in s do
    yield (b,a)
  end
end
```

evaluates to $\{8 -> 5, 5 -> 3\}$.

Using **for**-loops it is possible to formulate all of those *fold*-related functionals known from functional programming in a way which is easier to parse for most people.