# SYNTAX AND SEMANTICS OF BABEL-17

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?

*Date*: April 19, 2010.

My answer to this question is: Because I always wanted to create a programming language in which I can express myself compactly and without jumping through too many hoopes, that is easy to understand and to learn, which allows me to apply software engineering principles like encapsulation, and which yields efficient enough programs. And now I feel that I have gathered enough knowledge and experience to start walking towards this goal.

Babel-17 is not a radically new or revolutionary programming language. It picks those raisins I like out of languages like Standard ML, Alice ML, Java, Scala, Erlang, Haskell, Lisp, Clojure and Mathematica, and tries to evolve them into a beautiful new design. Of course, what some regard as beautiful, others might find unremarkable, or even ugly.

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, it becomes pure. Therefore I call Babel-17 a *clean* functional language.

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 *functional structured programming* via the notion of linear scope.

In this paper I specify Babel-17 v0.2 in a mostly informal style: formal notation is sometimes used when it supports clarity.

## 2. 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      module   package    yield      match       case
    as      val       def       in    exception    lazy    concurrent  memoize
    to    downto     true     false     this     random    infinity    await
 private  public  protected   here      root     external
```

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.

TABLE 1. Further Unicode Symbols

| Unicode Hex Code | Display | ASCII Equivalent |
|---|---|---|
| 2261 | ≡ | `==` |
| 2262 | ≢ | `!=` |
| 2264 | ≤ | `<=` |
| 2265 | ≥ | `>=` |
| 221E | ∞ | **infinity** |
| 2227 | ∧ | `&` |
| 2228 | ∨ | `\|` |
| 00AC | ¬ | `!` |
| 2237 | :: | `::` |
| 2192 | → | `->` |
| 21D2 | ⇒ | `=>` |
| 2026 | ... | `...` |

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 obey the same syntax as in C: There are multiline comments written between matching pairs of `/*` and `*/`, and there are comments that occupy the rest of the line which start with a `//`.

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.

## 3. Overview of Built-in Types

Each value in Babel-17 has a unique type. All built-in types are depicted in Table 2.

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 / 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 |
| *Module* | the type of modules |
| *ModuleKey* | the type of module keys |

## 4. 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 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 $(fx)\ y$.

## 5. Exceptions

Exception handling in Babel-17 mimicks exception handling mechanisms like those which can be found in Java or Standard ML or Erlang, 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 $v$ can be non-exceptional and of type *PersistentException* at the same time.

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

$$
\begin{array}{rcll}
\texttt{(}\textit{Exception}\ v\texttt{).}m & \rightsquigarrow & \textit{DynamicException}\ v & \\
\texttt{(}\textit{Exception}\ v\texttt{)}\ x & \rightsquigarrow & \textit{DynamicException}\ v & \\
f\ \texttt{(}\textit{DynamicException}\ v\texttt{)} & \rightsquigarrow & \textit{DynamicException}\ v & \text{where } f \text{ is non-exceptional} \\
\texttt{exception}\ v & \rightsquigarrow & \textit{DynamicException}\ v & \\
\texttt{lazy (}\textit{Exception}\ v\texttt{)} & \rightsquigarrow & \textit{PersistentException}\ v & \\
\texttt{concurrent (}\textit{Exception}\ v\texttt{)} & \rightsquigarrow & \textit{PersistentException}\ v &
\end{array}
$$

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 a later 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.

## 6. 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. It is guaranteed that $e$ is only evaluated once.

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. It is guaranteed that $e$ is only evaluated once.

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 $e$.

## 7. 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:

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

## 8. 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 $()$. For example, HELLO is equivalent to HELLO $()$. A constructor $c$ cannot have a dynamic exception as its parameter, therefore we have:

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

## 9. 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**

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 $e$.

A few of the pattern constructs are capable of raising dynamic exceptions because they incorporate arbitrary value expressions. When during a pattern match such an exception occurs, then the entire match is aborted and the dynamic exception is returned as the result of the **match**-expression. Furthermore note that persistent exceptions in $e$ or any of the patterns also lead to a dynamic exception if the value of the persistent exception is forced for pattern matching.

So what does a pattern look like? Table 3 lists all ways of building a pattern.

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

(1) The ellipsis ... is a $\delta$-pattern that matches any collection.
(2) If $\delta$ is a $\delta$-pattern, and $x$ an identifier, then ($x$ as $\delta$) is a $\delta$-pattern.
(3) If $\delta$ is a $\delta$-pattern, and $e$ an expression, then ($\delta$ **if** $e$) is a $\delta$-pattern.
(4) If $\delta$ is a $\delta$-pattern, and $e$ an expression, then ($e$ ? $\delta$) is a $\delta$-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.

## 10. Non-Determinism in Babel-17

The possibility of matching sets and maps introduces non-determinism into Babel-17. For example, the expression

```
match {1, 2}
  case {x, ...}  => x
end
```

can return both 1 and 2.

Another source of non-determinism 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 InvalidArgument $n$.

The third source of non-determinism is *concurrent* non-determism. The expression

**await** $l$

takes a finite list $l$ of possibly concurrently evaluated expressions and waits until at least one of the element expressions has finished computing. It returns a list of the element expressions that have been computed so far. Again, in case $l$ is non-exceptional and not a finite list, the result is an exception with parameter InvalidArgument $l$. If $l$ is a dynamic exception, then it is propagated such that the result is $l$.

TABLE 3. Pattern Constructs in Babel-17

| Syntax | General 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 $s$ |
| [$p_1$, ... , $p_n$] | matches a list consisting of $n \geq 0$ elements, such that element $e_i$ of the list is matched by pattern $p_i$ |
| ($p_1$, ... , $p_n$) | matches a list consisting of $n = 0$ or $n \geq 2$ elements, such that element $e_i$ of the list is matched by pattern $p_i$ |
| ($p$) | same as $p$ |
| [$p_1$, ... , $p_n$, $\delta$] | matches a list/vector consisting of at least $n \geq 0$ elements, such that the first $n$ elements $e_i$ of the list/vector are matched by the patterns $p_i$ |
| ($p_1$, ... , $p_n$, $\delta$) | matches a vector/list consisting of at least $n \geq 0$ 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 |
| ($p$ **if** $e$) | matches $p$, but only if $e$ evaluates to **true**; identifiers bound in $p$ can be used in $e$ |
| (**val** $e$) | matches any 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 (Some $r$) and $p$ matches $r$, or if the result is **true** and $p$ matches (). |
| ($d$ ? $p$) | here $d$ is a not a function and non-exceptional; calculate the result of ($v$.deconstruct $d$) where $v$ denotes the value to be matched; the match succeeds if the result is (Some $r$) and $p$ matches $r$, or if the result is **true** and $p$ matches () |
| ($e$ ?) | same as ($e$ ? _) |
| {$p_1$, ... , $p_n$} | matches any set such that every element of the set matches some $p_i$, and such that every $p_i$ is matched by some element of the set |
| {$p_1$, ... , $p_n$ , $\delta$} | matches any set such that every $p_i$ is matched by some element of the set |
| {$k_1$ -> $v_1$, ... , $k_n$ -> $v_n$} | matches any map that can be constructed from the $n$ key-value patterns |
| {$k_1$ -> $v_1$, ... , $k_n$ -> $v_n$, $\delta$} | matches any map that contains the $n$ key-value patterns |
| {->} | matches the empty map |

## 11. 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

$s_1$
$\vdots$
$s_n$
$e$

where the $s_i$ are statements and $e$ is the return expression of the block (a return expression is just a normal expression). In a block expression newlines can be used to separate the statements and the return expression from each other. The other means of separation are semicolons. Later on we will see how block expressions can be used to imitate imperative programming; on those occasions there will be no return expression $e$ but only statements.

Statements are Babel-17's primary tool for introducing identifiers. There are several kinds of statements. Two 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 $e$ gets evaluated and then is matched to $p$. If the match is successful then all identifiers bound by the match can be used in later statements and in the return expression of the block expression. Otherwise, some kind of dynamic exception must have happened which then becomes the value of the entire block expression.

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

**def** *id arg* $= e$

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, then they must appear sequentially in the block expression, without any other statements interrupting the sequence.

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

TABLE 4. Legal and illegal definitions

| | | | |
|---|---|---|---|
| **val** x = y<br>**val** y = 0 | **def** x = y<br>**val** y = 0 | **val** x = y<br>**def** y = 0 | **def** x = y<br>**def** y = 0 |
| *illegal* | *illegal* | *legal* | *legal* |

between definitions via **def** and definitions via **val**. Only those **val**-identifiers are in scope that have been defined *previously*. Table 4 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 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 $e_i$.

While only one simple definition per block and identifier is allowed, 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). The reason for this is that above code snippet is syntactic sugar for:

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

*ToDo: need to explain more about order of vals and defs, especially that vals cannot reference later vals, even not indirectly via defs. This will also clean up the parts about module and object initialization.*

## 12. 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 $b_i$.

## 13. Modules

Modules are Babel-17's way to structure programs and to define access boundaries. A module has the following syntax:

**module** $m$
  $s_1$
  $\vdots$
  $s_n$
**end**

The $s_i$ are **val** or **def** statements, or they are themselves again *nested* module definitions / declarations. Nested module definitions have the following syntax:

*attr* **module** $m$
  $s_1$
  $\vdots$
  $s_n$
**end**

Nested module declarations look as follows:

*attr* **external module** $m$

The $m$ always is a *module path*. A module path consists of a non-empty sequence of identifiers which are separated by periods. For example, `lang` is a module path, and so is `lang.math.taylor`. For nested module definitions/declarations, $m$ cannot contain any periods. Nested modules have an attribute *attr* which can be either **public** or **protected** or **private**. If there is no explicit attribute given, then the default attribute is **private**. These attributes are also used for those statements $s_i$ which are **def** statements, again with the default being **private** .

Before explaining what these attributes mean, let us first look at *inner* and *outer submodules*. Examples of both are given in Figure 1. In both example cases, the absolute module path of the submodule is `root.main.sub`. The absolute module path of the main module is `root.main`. This means that there is a *root module* which is the only module that has no *supermodule*. For every other module, the supermodule must exist. In other words, we can view the set of modules as a tree, such that the child of a node is a submodule of the module represented by this node, and such that for each node there is a unique module definition. If a node is an inner submodule, then all its children must be inner submodules, too.

The semantic difference between an inner and an outer submodule is that the inner submodule imports the namespace of its parent; more specifically, the inner submodule imports from the parent module all **def** and **module** definitions, but only those **val** definitions that come before the inner submodule definition.

The reserved keyword **here** denotes the current module. The reserved keyword **root** denotes the root module.

A module is an object, just like all other values in Babel-17. The messages it can successfully receive depend on its **def** and **module** statements, their attributes, and the module of the sender. For two modules $P$ and $Q$ we write $P = Q$ iff $P$ and $Q$ denote

FIGURE 1. Inner Submodule vs. Outer Submodule

| *Inner Submodule* | *Outer Submodule* |
|---|---|
| **module** main<br>$s_1$<br>⋮<br>$s_{i-1}$<br>*attr* **module** sub<br>⋮<br>**end**<br>$s_{i+1}$<br>⋮<br>$s_n$<br>**end** | **module** main<br>$s_1$<br>⋮<br>$s_{i-1}$<br>*attr* **external module** sub<br>$s_{i+1}$<br>⋮<br>$s_n$<br>**end**<br>**module** main.sub<br>⋮<br>**end** |

TABLE 5. Rules of Module Access

| | **here** = *sender* | **here** $\lhd_i$ *sender* | **here** $\lhd_o$ *sender* | otherwise |
|---|---|---|---|---|
| **private** | + | + | - | - |
| **protected** | + | + | + | - |
| **public** | + | + | + | + |

FIGURE 2. Module Initialization Example

```
module main
  val y = here.sqr 9
  val z = y
  def sqr x = x * x
end
```

the same module (i.e. they have the same absolute module path), and $P \lhd Q$ iff not $P = Q$ and $Q$ is a direct or indirect submodule of the module $P$. We write $P \lhd_i Q$ iff $P \lhd Q$ and the child of $P$ on the path to $Q$ is an inner submodule. We write $P \lhd_o Q$ iff $P \lhd Q$ and not $P \lhd_i Q$. In Table 5 **here** denotes the module to which the message is sent, and *sender* denotes the module from where the message is sent. The table shows if a message send is successful (+ means yes, − means no), depending on the relation between **here** and *sender*, and depending on the attribute of the defining statement.

Before a module can be used as an object, it must be initialized. The result of a complete initialization is that all **val** statements of the module have been executed, and that all inner submodules of the module have been initialized, too, all in the order of their appearance in the module. It must be possible to use a module as an object before it has been completely initialized, though, as the module shown in Figure 2 shows. To

initialize module *main*, we need to evaluate first $y$ and then $z$. Evaluating $y$ involves sending the message *sqr* to *main*. But at this point, *main* has not been completely initialized yet!

We solve this problem by keeping track of the progress of the initialization of a module. The progress is measured by the *initialization counter* of the module. When the initialization of the module starts, it is 0. Each time a **val** statement has been evaluated, this counter is increased by 1. For each message of the module we can determine *statically* which initialization counter is required so that all values directly used by this message are available. If during the initialization of the module a message arrives at the module that has a higher initialization counter requirement than the current initialization counter can provide, a dynamic exception is the result. It is not possible to send a message to a module that has not at least started its initialization because:

(1) if the module is **root**, then it always has been completely initialized;
(2) if the module is **here**, then it obviously has started initialization already;
(3) if the module is m.id, then the only other means apart from the above of obtaining a reference to it is by sending id to m; the result of this message send will be a completely initialized module if its initialization has not even started yet or has been completed already; otherwise the result will be a module currently initializing.

## 14. Packages

*ToDo: need to write this section*

## 15. Objects

Modules already give the developer the ability to define her own objects in Babel-17. This ability is strongly limited, though, because it is possible to determine statically how many module objects there are in a Babel-17 program. A more flexible way of creating objects is provided by the **object** expression which has the following syntax:

**object**
$s_1$
$\vdots$
$s_n$
**end**

The statements $s_i$ are either **val** definitions or optionally attributed **def** statements; the default attribute is **private**.

A keyword that can be used in object **def** statements is **this**. It denotes the current object. Because the current object exists only after the object has been completely initialized, it must not be used in the **val** statements of the object; neither may it be used by those **def** statements that involve messages which are sent directly or indirectly from the **val** statements of the same object.

Object initialization works similarly to module initialization. But unlike in the module case, no outside reference to the object currently being initialized exists. Because of above restrictions for **this**, we can furthermore determine *statically* if all initialization counters fulfill their requirements. If one doesn't, then this leads to a compile-time error.

TABLE 6. Boolean Operators

| ! a | a & b | a \| b |
|---|---|---|
| **match** a<br>  **case true** =><br>  **false**<br>  **case false** =><br>  **true**<br>  **case** _ =><br>  **exception** TypeError<br>**end** | **match** a<br>  **case true** =><br>  **match** b<br>  **case true** =><br>  **true**<br>  **case false** =><br>  **false**<br>  **case** _ =><br>  **exception** TypeError<br>  **end**<br>  **case false** =><br>  **false**<br>  **case** _ =><br>  **exception** TypeError<br>**end** | **match** a<br>  **case false** =><br>  **match** b<br>  **case true** =><br>  **true**<br>  **case false** =><br>  **false**<br>  **case** _ =><br>  **exception** TypeError<br>  **end**<br>  **case true** =><br>  **true**<br>  **case** _ =><br>  **exception** TypeError<br>**end** |

The returned object responds to those messages which have corresponding **def** statements. The rules for object access are those for module access. Therefore objects that have been defined in the same module can access all of each others messages, even those which have been declared as **private**.

There is no inheritance in Babel-17. I think that inheritance is actually *against* the practices of good software engineering, as it often is all about patching objects *after* they have been designed. On the other hand, if objects are extended via inheritance according to a given design, then I think Babel-17 is flexible enough to represent this design without inheritance.

## 16. 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**
  $e_2$
  ⋮
**elseif** $b_n$ **then**
  $e_n$
**else**
  $e_{n+1}$
**end**

TABLE 7. Relational Operators

| Syntax | Semantics |
|---|---|
| $a >= b$ | $b <= a$ |
| $a == b$ | $a <= b$ & $b <= a$ |
| $a <> b$ | $!(a <= b)$ & $!(b <= a)$ |
| $a < b$ | $a <= b$ & $!(a >= b)$ |
| $a > b$ | $a >= b$ & $!(a <= b)$ |
| $a != b$ | $!(a == b)$ |

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 => e1
  case false => e2
  case _ => exception TypeError
end
```

## 17. 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:

$a <= b <= c > d <> e$

The semantics of the above is

$a <= b$ & $b <= c$ & $c > d$ & $d <> e$ .

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 not $a <= b$ & $b <= c$ & $c <= d$ & $d <= e$, but rather

```
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:

- $-\infty < b < z < s < mod < modkey < list < cexpr < f < set < map < \infty$ holds for all booleans $b$, integers $z$, strings $s$, module keys $modkey$, modules $mod$, lists $l$, constructed expressions $cexpr$, functions $f$, sets $set$, maps $map$
- 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
- modules are totally ordered by representing them by the lists created from separating the absolute module path by periods
- module keys are totally ordered by the ordering on the modules they belong to
- lists 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
- maps are partially ordered by representing them by sorted lists that consist of the (key, value) pairs of the map
- let $a$ be a user-defined object (i.e., defined via **object**); if $a$ cannot receive the message `representative`, then $a <> b$ holds for all $b$; otherwise let $ra$ be the result of sending `representative` to $a$ from within the module where $a$ was defined; then $a <= b$ is defined by $ra <= b$, and $b <= a$ is defined via $b <= ra$ for all $b$.

The purpose of *module keys* is to provide a secret that can only be obtained by code within the module. Inside a module, the module key can be obtained via

```
@
```

The main application of module keys is to create abstract datatypes that cannot be confused with other types with respect to the built-in partial order of Babel-17.

## 18. Sets and Maps

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

```
{42, ∞, 3}
```

Sets are always sorted. The default order is Babel-17's built-in partial order, but it is also possible to create sets that work with arbitrary partial orders; an arbitrary partial order is represented as a function that takes a pair and returns a boolean. 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 most commonly used notation for sets.

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 the set/list $T$ into the set $S$ |
| $S$ -- $T$ | set created by removing all elements of the set/list $T$ from the set $S$ |
| $S$ * $T$ | cartesian product of the set $S$ and the set $T$, sorted by lexicographic ordering |
| $S$ ** $T$ | set created by removing all elements from the set $S$ which are not in the set $T$ |
| $S$ $e$ | tests wether $e$ is an element of $S$ |
| $S/f$ | set created by applying the function $f$ to the elements of $S$ |
| $S$.reorder $p$ | set created by reordering $S$ according to the partial order $p$; in case of duplicate elements, the greater of the elements relative to the order of $S$ is kept |

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, by default this is Babel-17's built-in partial order. Operations on maps are only 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 most commonly used notation for maps.

## 19. 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 a reference implementation for Babel-17 v0.2, which is currently scheduled for the summer of 2010. Until then, let me just make a few comments about it:

- 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 v0.2, but will maybe 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

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 of the map/set/list $N$ |
| $M$ `--` $S$ | map created from $M$ by removing all keys that are in the set/list/map $S$ |
| $M$ `*` $N$ | 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 |

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
- Android
- iPhone / iPad SDK
- LLVM
- HTML5 / Javascript

## 20. 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. syntactic_plus  5
```

TABLE 10. Syntactic Sugar

| Sugared | Desugared |
|---|---|
| $a$ `+` $b$ | $a$.syntactic_plus $b$ |
| $a$ `-` $b$ | $a$.syntactic_minus $b$ |
| `-` $a$ | $a$.syntactic_uminus |
| $a$ `*` $b$ | $a$.syntactic_times $b$ |
| $a$ `/` $b$ | $a$.syntactic_div $b$ |
| $a$ `%` $b$ | $a$.syntactic_mod $b$ |
| $a$ `^` $b$ | $a$.syntactic_pow $b$ |
| $a$ `++` $b$ | $a$.syntactic_plusplus $b$ |
| $a$ `--` $b$ | $a$.syntactic_minusminus $b$ |
| $a$ `**` $b$ | $a$.syntactic_timestimes $b$ |
| $a$ `to` $b$ | $a$.syntactic_to $b$ |
| $a$ `downto` $b$ | $a$.syntactic_downto $b$ |
| $f$ $x$ | $f$.apply $x$ |

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. To simplify the implementation of this, there is the restriction that apply methods cannot be implemented via a **def** that takes no parameter.

## 21. 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.

## 22. 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                    begin                    begin
  val x = 1                val x = 1                val x = 1
  val y = 2                val y = 2                val y = 2
  begin                    begin                    begin
    val x = 3                val x = 3                val x = 3
    val y = 4 * x            x = 4 * x                val y = 0
  end                      end                        y = 4 * x
  (x, y)                   (x, y)                   end
end                      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**<br>  **val** x = 1<br>  **val** y =<br>  **begin**<br>  x = 2<br>  x+x<br>  **end**<br>  (x, y)<br>**end** | **begin**<br>  **val** x = 1<br>  **val** y = 3 ∗<br>  **begin**<br>  x = 2<br>  x+x<br>  **end**<br>  (x, y)<br>**end** | **begin**<br>  **val** x = 1<br>  **val** y = 3 ∗<br>  **begin**<br>  **val** x = 2<br>  x+x<br>  **end**<br>  (x, y)<br>**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.

## 23. 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 $\{1,\ 2,\ 4,\ 10\}$.

## 24. 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.

## 25. Notes

This section mentions things that have not been explained (well) before and that must be integrated into the main text later on. These things pop up during implementation work and looking closer at the spec.

25.1. **'apply' always takes a parameter.** This is a restriction that should hurt nobody. It ensures that an implementation does not need to look recursively for an 'apply' message.

25.2. **'representative' never takes a parameter.** This makes sense because functions cannot be compared to anything. Of course, if you really want to, you can still return a function.