SYNTAX AND SEMANTICS OF BABEL-17

STEVEN OBUA

Contents

Date: August 8, 2011.

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,
- 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 v0.3.1, 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 [1], Scala [2], Isabelle [5] (which is not a language, but a system for interactive theorem proving), 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 few sources of non-determinism out of the language, 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* [4] 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.

Not part of this document, but also part of the overall Babel-17 specification is its formal ANTLR v3.0 grammar; and the (small, but growing) set of unit tests that correspond to the different sections of this document. Both artifacts are available at [3].

2. Reference Implementation

At [3] you will find a reference implementation of Babel-17 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`. Preferably you would also send along a unit test that fails because of the observed behavior, but which you think should succeed.

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`	`unittest`	`force`	`this`
`try`	`catch`	`typedef`	`typeof`	`module`	`private`	`import`	`not`
`and`	`or`	`xor`	`native`	`root`	`lens`	`min`	`max`

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 integers in Babel-17. The following notations all denote the same integer:

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

There is also a decimal syntax for floating point numbers in Babel-17:

```
15.0  1.5E1  1.5E+1  150E-1  15e0
```

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"`

Multi-line comments in Babel-17 are written between matching pairs of `#(` and `)#` and can span several lines. Single-line comments start with `##`.

Pragmas start with `#`, just like comments. There are the following pragmas: `#assert`, `#catch`, `#log`, `#print`, `#profile`.

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.

4. Overview of Built-in Types

Each value in Babel-17 has a unique type. All built-in types are depicted in Table 2. Often we use the name of a type to refer to this type in this specification, but the representation of it in actual Babel-17 is given in the type column. The type of a value v can be obtained via

TABLE 1. Further Unicode Symbols

Unicode Hex Code	**Display**	**ASCII Equivalent**
2261	$\equiv$	`==`
2262	$\not\equiv$	`<>`
2264	$\leq$	`<=`
2265	$\geq$	`>=`
2237	$::$	`::`
2192	$\rightarrow$	`->`
21D2	$\Rightarrow$	`=>`
2026	$\ldots$	`...`

TABLE 2. Built-in Types of Babel-17

Name	**Type**	**Description**
Integer	`int`	the type of arbitrary size integer numbers
Real	`real`	the type of real numbers
Boolean	`bool`	the type consisting of the values **true** and **false**
String	`string`	the type of valid Unicode strings
List	`list`	the type of lists
Vector	`vect`	the type of vectors / tuples
Set	`set`	the type of sets
Map	`map`	the type of maps
CExpr	`cexp`	the type of constructed expressions
Object	`obj`	the type of user-defined objects
Function	`fun`	the type of functions
PersistentException	`exc`	the type of persistent exceptions
DynamicException		the type of dynamic exceptions
Type	`type`	the type of types
Module	`module_`	the type of modules
Lens	`lens_`	the type of lenses
Native	`native_`	the type of native values

typeof v

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 `obj`, and for a value to be a function, it does not need to have type `fun`.

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

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

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 $(f\ x)\ y$.

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:

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

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. Catching an exception can be done via **match** or via **try−catch**. Both constructs 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

$$\textbf{lazy}\ e$$

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

$$\textbf{concurrent}\ e$$

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 want to explicitly force the evaluation of an expression. In those situations you use the expression

$$\textbf{force}\ e$$

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

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

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

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

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 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.

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:

$$\begin{array}{lcll} (\mathtt{DynamicException}\ v)\mathtt{::}t & \rightsquigarrow & \mathtt{DynamicException}\ v & \\ h\mathtt{::}(\mathtt{DynamicException}\ v) & \rightsquigarrow & \mathtt{DynamicException}\ v & \text{where } h \text{ is non-exceptional} \end{array}$$

Note that when the tail t is not a list, 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:

$$c\ (\mathtt{DynamicException}\ v) \rightsquigarrow \mathtt{DynamicException}\ v$$

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
  ⋮
  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.

A few of the pattern constructs incorporate arbitrary value expressions. When these expressions raise exceptions, they are propagated up.

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

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.

Besides the **match** construct, there is also the **try** construct. While **match** can handle exceptions, most of the time it is more convenient to use **try** for this purpose. The syntax is

try
  s_1
  ...
  s_m
catch
  case p_1 => b_1
  ⋮
  case p_n => b_n
end

The meaning of the above is similar to the meaning of

match
  begin
    s_1
    ...
    s_m
  end
case (**exception** p_1) => b_1
  ⋮
case (**exception** p_n) => b_n
case x => x
end

except for two differences:

(1) the latter expression might not be legal Babel-17 because of linear scoping violations,
(2) **try** does not catch persistent exceptions.

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	an integer number z, like 0 or 42 or -10, matches just that number z
$c\ p$	matches a *CExpr* with constructor c if the parameter of the *CExpr* matches p; instead of $c\ \_$ you can just write c
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
(**val** e)	matches any non-exceptional value which is equivalent to e; in case an `Unrelated`-exception is thrown, this pattern just does not match, and does *not* propagate up the exception
$(c\ !\ p)$	let v be the value to be matched; then the match succeeds if the result v.destruct_ c matches p
$(c\ !)$	short for $(c\ !\ \_)$
$(f\ ?\ p)$	f is applied to the value to be matched; the match succeeds if the result of the application matches p
$(f\ ?)$	short for (f ? **true**)
$\{m_1 = p_1,\ \ldots,\ m_n = p_n\}$	matches a value of type `obj` 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 a value of type `obj` 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
$(p : t)$	matches anything that has (or is auto-convertible to) type t and matches p
$(p : (e))$	matches anything that matches p and is or auto-converts to the type that e evaluates to
$(t\ p)$	inner-value pattern; see Section 27.2

Table 4. Collection Patterns

Syntax	**Description**
$[p_1, \ldots, p_n]$	matches a list consisting of $n \geq 0$ elements, such that element e_i of the list is matched by pattern p_i
$(p_1, \ldots, p_n)$	matches a vector 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 consisting of at least $n \geq 1$ elements, such that the first n elements e_i of the list are matched by the patterns p_i
$(p_1, \ldots, p_n, \delta)$	matches a vector consisting of at least $n \geq 1$ elements, such that the first n elements e_i of the vector are matched by the patterns p_i
$(p,)$	matches a vector consisting of a single element that matches the pattern p.
$(h{::}t)$	matches a non-empty list such that h matches the head of the list and t its tail
$\{p_1, \ldots, p_n\}$	see section 17
$\{p_1, \ldots, p_n\ , \delta\}$	see section 17
$\{q_1 \text{ −> } p_1, \ldots, q_n \text{ −> } p_n\}$	see section 17
$\{q_1 \text{ −> } p_1, \ldots, q_n \text{ −> } p_n, \delta\}$	see section 17
$\{\text{−>}\}$	see section 17
(**for** $p_1, \ldots, p_n$ **end**)	see section 25
(**for** $p_1, \ldots, p_n, \delta$ **end**)	see section 25

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.

Another 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.

There is a third source of non-determinism which has its roots in the module loading mechanism (see Section 27).

Babel-17 has been designed such that if you take the above sources of non-determinism 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

$$\begin{array}{l} s_1 \\ \vdots \\ s_n \end{array}$$

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

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 fails, 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 $id\ arg = e$

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

TABLE 5. Legal and illegal definitions

val x = y **val** y = 0	**def** x = y **val** y = 0	**val** x = y **def** y = 0	**def** x = y **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*, 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** p_1 => b_1
  ⋮
 case p_n => b_n)

is equivalent to

begin
  def f p_1 = **begin** b_1 **end**
  ⋮
  def f p_n = **begin** b_n **end**
  f
end

where f is fresh for the p_i and b_i.

14. Object Expressions

Object expressions have the following syntax:

object
  s_1
  ⋮
  s_n
end

The s_i are statements. 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
  ⋮
  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

Table 6. Boolean Operators

not a	a **and** b	a **or** 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

shadow messages of the parents. The messages of an earlier parent shadow the messages of a later one.

The keyword **this** can be used only in **def**-statements of an object and points to the object that the original message has been sent too.
There is also a 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 $m_1 = w_1$
    ...
    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. Furthermore, there is the **xor** operator which is defined in the usual way. Babel-17 also has **if**-

TABLE 7. Relational Operators

Syntax	**Semantics**
$a < b$	$(a \sim b) < 0$
$a == b$	$(a \sim b) == 0$
$a > b$	$(a \sim b) > 0$
$a <= b$	$(a \sim b) <= 0$
$a >= b$	$(a \sim b) >= 0$
$a <> b$	$(a \sim b) <> 0$

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

The **elseif**-branches are optional. 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 which is defined in terms of the operator $\sim$. The expression $a \sim b$ returns an *Integer*; the usual relational operators are defined in Table 7.

It is possible to chain relational operators like this:

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

Intuitively, the above means

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

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

```
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 dynamic exceptions, then the result of a comparison is a dynamic exception with the same parameter as the first such operand (from left to right).

It is possible that two values a and b are not related with respect to the built-in order. In this case, $a \sim b$ throws an *Unrelated*-exception. Taking this into account, the definition of $a\ op\ b$ for $op \in \{<, >, <=, >=\}$ is

```
match a ~ b
  case (u if u op 0) => true
  case _ => false
end
```

The cases $op \in \{==, <>\}$ do not propagate the *Unrelated*-exception, and are defined like this:

```
match a ~ b
  case (u if u op 0) => true
  case _ => false
  case (exception Unrelated) => true op false
end
```

In principle, two values a and b are unrelated if they have different types. This statement is weakened by the existence of automatic type conversion (Section 27.7): if a and b have different types, it is checked if a can be automatically converted to something of type **typeof** b, or if b can be automatically converted into something of type **typeof** a. If so, then one of these conversions is done and the result is compared with the other value.

The built-in partial order has the following properties:

- if a and b have the same type, and this type is `obj` or user-defined (see Section 27.2 for how to define your own types), then $a \sim b$ is equivalent to

```
match a.compare_ b
```

```
        case u : int => u
        case _ => exception Unrelated
        case (exception _) => exception Unrelated
      end
```

if a has a `compare_` message; if not then a and b are compared according to the number, names and contents of their messages
- persistent exceptions are partially ordered by their parameter
- booleans are totally ordered and **false** < **true** holds
- integers are totally ordered in the obvious way
- reals are partially ordered as described in Section 18
- strings are totally ordered via lexicographic ordering
- lists are partially ordered by the lexicographic ordering
- vectors are partially ordered by the lexicographic ordering
- constructed expressions are partially ordered by representing them by the pair consisting of constructor and parameter
- $f \neq g$ for all values f and g of type `fun`
- sets are partially ordered by first comparing their sizes, and then their elements
- maps are partially ordered by first comparing their sizes, then their keys, and then their corresponding values
- types are totally ordered by their names
- modules are totally ordered by their names

There are built-in operators **min** and **max** that compute the minimum and the maximum of a *collection* (Section 25 defines what a collection is). For example,

min (1, 2) == **max** (−1, 1, 0) == 1

17. Sets and Maps

Sets and maps are built into Babel-17. For example, the set consisting of 3, 42 and 15 can be written as

{42, 15, 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.

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 well-defined if all keys of the map together with all other involved keys are totally ordered by the associated order.

Pattern matching is available also for sets and maps. The pattern

$\{p_1, \ldots, p_n\}$

matches a set that has n elements $e_1, \ldots, e_n$ such that p_i matches e_i and $e_i < e_j$ for $i < j$. The pattern

$\{p_1, \ldots, p_n, \delta\}$

matches a set that has $m \geq n$ elements such that its first n elements match the patterns p_i, and the set consisting of the other $m - n$ elements matches the δ-pattern.

Similarly, the pattern

$\{p_1 \text{ -> } q_1, \ldots, p_n \text{ -> } q_n\}$

matches a map consisting of n key/value pairs such that the key/value pairs match the pattern pairs in order. The pattern $\{\text{ -> }\}$ matches the empty map. Map patterns can have a δ pattern, too.

18. Reals and Interval Arithmetic

Babel-17 is radical in its treatment of floating point arithmetic: there is *only* interval arithmetic. What that means is that reals are represented in Babel-17 as closed real intervals, i.e. as pairs $[a; b]$ where a and b are floating point numbers and $a \leq b$. The usual floating point numbers f can then be represented as $[f; f]$. For example, the notation 1.5 is just short for $[1.5; 1.5]$.

The general form of this interval notation is

$[a;\ b]$

Its value is formed by evaluating a and b, resulting in `reals` $[a_1; a_2]$ and $[b_1; b_2]$, respectively, and then forming the interval $[\min(a_1, b_1), \max(a_2, b_2))]$. In case the evaluation does not yield reals, a *DomainError*-exception is thrown.

The general rule of interval arithmetic in Babel-17 is as follows: If f is a mapping from $\mathbb{R}^n$ to $\mathbb{R}$, then for its interval arithmetic implementation F the following must hold for all $x_i \in \mathbb{R}$ and all `reals` $y_i = [u_i; v_i]$ such that $u_i \leq x_i \leq v_i$:

$$f(x_1, \ldots, x_n) \in F(y_1, \ldots, y_n)$$

The built-in order on `reals` is defined such that

$$[a_1; a_2] \leq [b_1; b_2] \quad \text{iff} \quad a_2 < b_1 \vee (a_1 = b_1 \wedge a_2 = b_2)$$

Note that certain seemingly obvious inequalities like $[1.0; 2.0] \leq 2.0$ do *not* hold!

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 is 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.

TABLE 8. 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.slash_ 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.slash__ b
a `to` b	a.to_ b
a `downto` b	a.downto_ b
f x	f.apply_ x

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 8 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 $\mathit{ref}_1, \ldots, \mathit{ref}_n$

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 always remembers a value once it has been computed; weak memoization instead may choose not to remember computed results in certain situations, for example in order to free memory in low memory situations. Note that all arguments to the same memoized function must be totally ordered by the built-in partial order.

21. Linear Scope

We are now ready to explore the full power of block expressions and statements in Babel-17. We approach the topic of this and the following sections, *linear scope*, in an example-oriented way. A more in-depth treatment can be found in my paper *Purely Functional Structured Programming* [4].

First, we extend the syntax of **val**-statements such that it is legal to leave out the keyword **val** in certain situations. Whenever the situation is such that it is legal to write

$$a = expr$$

for a variable identifier a, we say that a is in *linear scope*. We call the above kind of statements *assignments*. A necessary condition for a to be in linear scope is for a to be in scope because of a previous **val**-statement or a pattern match. The idea behind linear scope is that when the program control flow has a linear structure, then we can make assignments to variables in linear scope without leaving the realm of purely functional programming.

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
```

So far, **val**-statements and assignments have indistinguishable semantics. The differences start to show when we look at nested block expressions:

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 or via a pattern match.

Linear scope spreads along the statements and nested block expressions of a block expression. It usually does not spread into expressions. An exception are certain expressions that can also be viewed as statements. We call these expressions *control expressions*. For example, in

```
begin
  val x = 1
  x = 2
  val y = a * b
 (x, y)
end
```

the linear scope of x does not extend into the expressions a and b, because $a * b$ is not a control expression. Therefore the above expression will always evaluate (assuming there is no exception) to a pair which has 2 as its first element. But for example in

```
begin
  val x = 1
  x = 2
  val y =
     begin
       s1
       ⋮
       sn
     end
  (x, y)
end
```

the linear scope of x extends into s_1 and spreads then along the following statements. Therefore the first element of the pair that is the result of evaluating above expression depends on what happens in the s_i. Here are three example code snippets that further illustrate linear scope:

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

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

These are the control expressions that exist in Babel-17:

- **begin** ... **end**
- **if** ... **end**
- **match** ... **end**
- **try** ... **catch** ... **end**

- **for** ... **do** ... **end**
- **while** ... **end**

The last two control expressions are loops and explained in the next section. The first four have already been treated without delving too much into their statement character. We have already seen how **begin** ... **end** is responsible for nesting block expressions, when it is used as a statement. 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, but **match**-statements throw a NoMatch exception if none of the patterns matches.

22. Record Updates

Linear scope makes it possible to have purely functional record updates in Babel-17. Let us assume you have defined u via

```
val u = {x = 10, y = 20, z = −4}
```

and now you want to bind u to another record that differs only in the x-component. You could proceed as follows:

```
u = {x = 9, y = u.y, z = u.z}
```

but this clearly does not scale with the number of components of u. A more scalable alternative would be to write

```
u =
  object + [u]
    def x = 9
  end
```

Babel-17 allows to write down the above statement in a more concise form:

```
u.x = 9
```

In general, for a value u of type `obj`,

u.m = t

is shorthand notation for

```
u =
  begin
    val evaluated_t = t
    object + [u]
```

```
        def m = evaluated_t
      end
    end
```

Actually, there is an exception to the above rule: if u has a message m_`putback_`, then

```
u.m = t
```

is actually shorthand for

```
u = u.m_putback_ t
```

The reason for this exception is that it allows us to generalize the concept of a purely functional record update to the concept of a *lens*.

23. Lenses

Lenses [6] allow us to generalize the thing we did with updating records in a purely functional way. Lenses and linear scope work together well; this is very similar to how lenses and the state monad work together well [7].

A *lens* is basically a pair of functions (g, p), where g is called the *get* function and p is called the *putback* function of the lens. In the special case where a lens represents a field m of a record u, this pair would be defined via

$$g\ u = u.m, \quad p\ u\ t = \textit{a copy of } u \textit{ where the field } m \textit{ has been set to } t$$

Generalizing from this special case, a lens should obey the following laws:

$$g\ (p\ u\ t) = t, \quad p\ u\ (g\ u) = u$$

In Babel-17, these laws are just recommendations for how to construct a lens as there are no mechanisms in place to ensure that a lens actually obeys them.

You can define a lens in Babel-17 directly via providing *get* and *putback* explictly:

```
def g u = u.m
def p u = t => begin u.m = t; u end
val l = lens (g, p)
```

The above code constructs a new lens l. Lenses actually form their own type, so we have **typeof** l == (: lens_).

What can you do with a lens once you got one? You can apply l to a value u via

```
l u
```

This applies the *get* function of l to u, so for our specific l we have that l u is equivalent to u.m. In order to emphasize the aspect that l can be regarded as a field accessor for u, instead of l u you can also write

```
u.(l)
```

This notation has the advantage that it can also be used on the left hand side of an assignment:

```
val u = {m = 10, n = 12}
u.(l) = 23
u == {m = 23, n = 12}
```

TABLE 9. Modifying Assignment Operators

Syntax	Semantics	Syntax	Semantics	Syntax	Semantics
x += y	x = x + y	x *= y	x = x * y	x ^= y	x = x ^ y
x =+ y	x = y + x	x =* y	x = y * x	x =^ y	x = y ^ x
x ++=y	x = x ++ y	x **=y	x = x ** y	x div= y	x = x div y
x =++ y	x = y ++ x	x =** y	x = y ** x	x =div y	x = y div x
x −= y	x = x − y	x /= y	x = x / y	x mod= y	x = x mod y
x =− y	x = y − x	x =/ y	x = y / x	x =mod y	x = y mod x
x −−= y	x = x −− y	x //= y	x = x // y	x min= y	x = min (x, y)
x =−− y	x = y −− x	x =// y	x = y // x	x max= y	x = max (x, y)
x xor= y	x = x xor y	x and= y	x = x and y	x or= y	x = x or y
x =xor y	x = y xor x	x =and y	x = y and x	x =or y	x = y or x

The above evaluates to **true**, because the line u.(l) = 23 is short for u = l.putback u 23.

Lenses come automatically equipped with a *modify* operation, l.modify u f is short for l.putback u (f (l u)). There is special notation that supports this operation:

> **val** u = {m = 10, n = 12}
> u.(l) += 2
> u == {m = 12, n = 12}

This again evaluates to **true**, because the line u.(l) += 2 is short for

> u = l.modify u (m => m + 2)

The *identity lens*

> **val** id = **lens** (x => x, x => y => y)

blurs the distinction between normal assignment and lens assignment, because the meaning of

> **val** x = 10
> x = 12

is exactly the same as the meaning of

> **val** x = 10
> x.(id) = 12

In particular, our new assignment operators like += and ∗= also work in the context of normal assignment:

> **val** x = 10
> x += 2
> x == 12

This evaluates to **true**. Tables 9 list all modifying assignment operators.

Besides via directly giving the pair of functions that form a lens, a lens can also be defined by its *access path*. The access path of a lens is basically just how you would write down the *get* function of the lens if it could be defined using message passing (possibly with arguments) and lens application only. For example, instead of defining a lens via

```
def g u = u.m
def p u = t => begin u.m = t; u end
val l = lens (g, p)
```

you could just write

```
val l = lens u => u.m
```

The identity lens becomes even simpler in this form:

```
val id = lens u => u
```

Another valid legal lens definition is:

```
val h = lens x => ((x.mymap.lookup 10) + 40).(l)
```

You could define the above lens directly via a pair of functions as follows:

```
def g x = ((x.mymap.lookup 10) + 40).(l)
def p x = t =>
  begin
    val u1 = x.mymap
    val u2 = u1.lookup 10
    val u3 = u2 + 40
    val new_u3 = l.putback u3 t
    val new_u2 = u2.plus__putback_ 40 new_u3
    val new_u1 = u1.lookup_putback_ 10 new_u2
    val new_x = x
    new_x.mymap = new_u1
    new_x
  end

val h = lens (g, p)
```

The cool thing about lenses is that they are *composable.* Given two lenses a and b, we can define a new lens c via

```
val c = lens x => x.(a).(b)
```

Multiplication of lenses is defined to be just this composition, so we could also just write

```
val c = a * b
```

Here is an example that illustrates composition:

```
val a = lens x => x.a
val b = lens x => x.b

val x = {a = {a = 1, b = 2}, b = {a = 3, b = 4}}

x.(a*b) = 10
x.(b*a) = 20

x == {a = {a = 1, b = 10}, b = {a = 20, b = 4}}
```

This evaluates to **true**. Note that if the left hand side of an assignment is an access path, then the corresponding lens is automatically constructed. Therefore, in the above context the following five lines have identical meaning:

```
x.(a*b) = 10
x.(a).(b) = 10
x.a.b = 10
x.(a).b = 10
x.a.(b) = 10
```

24. With

By default, all yielded values of a block expression are collected into a vector which collapses in the case of 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. The **with** expression allows her to do just that. Its syntax is:

```
with c do
  b
end
```

where c is a *collector* and b is a block expression. A collector c is any object that

- responds to the message `collector_close_`,
- and returns via c.`collector_add_` x another collector.

It is recommended that collectors also support the message `empty` to represent the empty collector that has not collected anything yet.

Lists, vectors, sets, maps and strings 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} do
  yield 1
  yield 2
  yield 1
  10
end
```

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

25. Loops

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

```
while c do
  b
end
```

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

```
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*, and b is a block expression. 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,
- or otherwise by returning `(`e`, `C'`)` such that C' is also a collection.

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 in combination with linear scope it is possible to formulate all of those *fold*-related functionals known from functional programming in a way which is easier to parse (and remember) for most people. Let us for example look at a function that takes a list m of integers $[a_0, \ldots, a_n]$ and an integer x as arguments and returns the list

$$[q_0, \ldots, q_n] \quad \text{where} \quad q_k = \sum_{i=0}^{k} a_i\, x^i$$

The implementation in Babel-17 via a loop is straightforward, efficient and even elegant:

```
m => x =>
  with [] do
    val y = 0
    val p = 1
    for a in m do
      y = y + a*p
      p = p * x
      yield y
    end
  end
```

The built-in collections that can be used with for-loops are the usual suspects: lists, vectors, sets, maps and strings. Of course you can define your own custom collections. There is even a pattern you can use for matching against an arbitrary collection:

(**for** p_1, ..., p_n **end**)

and the corresponding δ pattern

(**for** p_1, ..., p_n, δ **end**)

match collections with exactly n or at least n elements, respectively.

26. Pragmas

Pragmas are statements that are not really part of the program, but inserted in it for pragmatical reasons. They are useful for testing, debugging and profiling a program. There are currently four different kinds of pragmas available:

`#log` e evaluates the expression e and logs it.

`#print` e evaluates the expression e such that it has no internal lazy or concurrent computations any more, and then logs it.

`#profile` e evaluates the expression e, gathers profiling information while doing so, and logs both.

`#assert` e evaluates the expression e and signals an error if e does not evaluate to **true**.

`#catch` p `try` e evaluates the expression e and signals an error if e does not evaluate to an exception with a parameter that matches p.

The Babel-17 interpreter / compiler is free to ignore pragmas, typically if instructed so by the user.

27. Modules and Types

Modules give Babel-17 code a static structure and improve on object expressions by providing more sophisticated encapsulation mechanisms via types. Modules are introduced with the syntax

module $p_1.p_2.\ldots.p_n$
  s_1
  $\vdots$
  s_n
end

where $p = p_1.p_2.\ldots.p_n$ is the *module path.*

The statements s_i that may appear in modules are basically those that may also appear in object expressions. Additionally, modules can contain *type definitions*; this is the topic of the next section.

Identifiers introduced via definitions become the messages that this module responds to except when hidden via **private**.

Modules can be nested. Basically,

module $p_1.p_2.\ldots.p_n$
  S_1
  module $q_1.q_2.\ldots.q_m$

```
    ⋮
    end
    S2
  end
```

means the same as

```
  module p1.p2.....pn
    S1
    S2
  end
  module p1.p2.....pn.q1.q2.....qm
  ⋮
  end
```

A module cannot access its surrounding scope, with the exception of those identifiers that have been introduced via **import**. So the following is not allowed, because inside module b the identifier u cannot be accessed:

```
module a
  val u = 2
  module b
    def message = u
  end
end
```

But the following *is* allowed:

```
module a
  import someCoolModule.importantValue => u
  module b
    def message = u
  end
end
```

27.1. **Loading Modules.** Before a module can respond to messages, it must be loaded. A module is in one of the following three states: DOWN, LOADING or UP. There are three things that one might suspect to change the state of the module:

A: the module receives a message m that corresponds to a definition inside of it,
B: the module receives a message m that corresponds to a submodule $p.m$,
C: the module receives an invalid message m.

Of these three things, neither B nor C change the state of the module, so we are looking now only at case A.

In the following we distinguish modules which need initialization from modules which don't. A module does not need initialization iff it consists only of the following types of statements: **def**, **typedef**, **import**, **private**, **memoize**.

If module p needs initialization and is in state DOWN, then sending it a message causes it to go to state LOADING; after p has been loaded, that is all statements of the module have been executed, p goes from state LOADING to state UP. If the module needs no initialization, the state goes directly from DOWN to UP.

If p is in state LOADING, then sending it a message will make the sender wait until p is in state UP. Note that this can lead to a deadlock situation. In case the Babel-17 runtime can detect this deadlock, a *DeadLock*-exception is thrown.

If p is already in state UP, then A does not change the state of the module.

Note that the state of a module is not directly accessible and only sometimes indirectly observable via the *DeadLock*-exception.

Here is an example that leads to a deadlock:

```
module deadlock
  def x = 10
  val a = deadlock.x + 1
  def y = a * a
end
deadlock.y
```

Note that the following example does *not* lead to a deadlock:

```
module noDeadlock
  def x  = 10
  val a = x + 1
  def y = a * a
end
noDeadlock.y
```

The most straightforward way to avoid deadlock problems is to write modules which need no initialization.

27.2. **Type Definitions.** A module may contain type definition statements. A new type t is introduced via

typedef t *pat* = *expr*

where t is an identifier, *pat* a pattern and *expr* an expression. Like a **def** statement, the above introduces a new function t. Unlike a **def** statement, it also introduces a new type t. The new function t has the property that all non-exceptional values v it returns will have type t and will consist of two components: an *inner value*, and an *outer value*. The inner value i is just the argument that was passed to t. The outer value o is the result of evaluating and forcing *expr* in case this does not lead to an exception. If *pat* does not match, then a *DomainError*-exception is thrown.

The point of all this is that the value v behaves just like its outer value o, i.e. v responds to the same messages as o, and does so in the same way. Nevertheless, the type of v is t, whatever the type of o might be. This means that you can implement the behavior of v by choosing an appropriate o which accesses the hidden encapsulated state i. This hidden state can also be accessed in the module in which the type has been defined via the *inner-value pattern* (t p) which matches v if it has type t and its inner value i matches p.

Let's look at a first type definition example:

```
module cards
  typedef rank i =
    match i
      case 14 => Ace
      case 13 => King
      case 12 => Queen
      case 11 => Jack
      case x if 2 <= x <= 10 => Number x
    end
end
```

In the above we define the type *cards.rank*. We can create a value of this type like this:

```
val k = cards.rank 13
```

The expression (:cards.rank) evaluates to the type *cards.rank*, therefore

```
(typeof k) == (:cards.rank)
```

evaluates to **true**.

Note that there is no other way to create a value of type *cards.rank* than via the *rank* function in the module *cards*. If you want to access the inner value of a value of type *cards.rank*, you can do so *within* the module *cards* via the inner-value pattern:

```
module cards
  typedef rank ...
   ...
  def rank2num(rank n) = n
   ...
end
```

Now you can access the inner value of a rank even outside of the *cards* module by calling the *rank2num* function we just defined. The following evaluates to **true**:

```
cards.rank2num (cards.rank 14) == 14
```

You can use the fact that a rank value behaves like its outer value to do pattern matching on rank values:

```
def rank2str(r : rank) =
  match r
    case (Ace !) => "ace"
    case (King !) => "king"
    case (Queen !) => "queen"
    case (Number ! n) => "number"+n
  end
```

It is possible to distribute the definition of a type t over several statements:

typedef t pat_1 = $expr_1$

$\vdots$

typedef t pat_n = $expr_n$

So an alternative way of defining the rank type would have been:

```
module cards
  typedef rank 14 = Ace
  typedef rank 13 = King
  typedef rank 12 = Queen
  typedef rank 11 = Jack
  typedef rank (n if 2 <= n <= 10) = Number n
end
```

As a shortcut notation, several typedef clauses belonging to the same type t can be combined into a single one by separating the different cases by commas:

```
module cards
  typedef rank 14 = Ace, 13 = King, 12 = Queen, 11 = Jack,
               (n if 2 <=n <= 10) = Number n
end
```

Another shortcut notation allows you to write instead of

```
typedef t (x as p) = x
```

just

```
typedef t p
```

Finally, it is possible to identify a type with the module it is defined in by using the same name for the type and the module. For example:

```
module util.orderedSet
  typedef orderedSet ...
 ...
end
```

This defines the module *util.orderedSet* and the type *util.orderedSet.orderedSet*. Because of our convention of identifying modules and types that have the same name, you can refer to the type *util.orderedSet.orderedSet* also just by *util.orderedSet*. In particular, the following evaluates to **true**:

```
(: util .orderedSet.orderedSet) == (:util.orderedSet)
```

This is all there is to defining your own types in Babel-17. In the following we list a few common type definition patterns: *simple enumerations*, *algebraic datatypes*, and *abstract datatypes*.

27.3. **Simple Enumerations.** It is easy to define a type that is just an enumeration:

```
module cards
  typedef suit Spades, Clubs, Diamonds, Hearts
end
```

We may choose to represent our rank type rather as an enumeration, too:

```
module cards
  typedef rank Ace, King, Queen, Jack, Number (n if 2 <= n <= 10)
end
```

27.4. **Algebraic Datatypes.** For enumeration types, the inner value is always equal to the outer value. This property is also shared by the more complex algebraic datatypes which are typically recursively defined:

```
module cards
  typedef bintree Leaf _, Branch ( _ : bintree, _ : bintree)
end
```

27.5. **Abstract Datatypes.** The type definition facility of Babel-17 is powerful enough to allow you to define your own abstract datatypes. Currently there is no type in Babel-17 for representing sets that are ordered by a given (i.e., not necessarily the built-in) order. So let's define our own. To keep it simple, we use lists to internally represent sets:

```
module util.orderedSet

  private orderedSet, ins

  typedef orderedSet (leq, list ) = nil

  def empty (leq : fun) = orderedSet (leq, [])

  def insert (orderedSet (leq, list ), y) = (orderedSet (leq, ins (leq, list , y))

  def toList (orderedSet (leq, list )) = list

  def ins (leq, [], y) = [y]
  def ins (leq, x::xs, y) =
    if leq (y, x) then
        if leq (x, y) then x::xs else y::x::xs end
    else
        x::( ins (leq, xs, y))
    end
end
```

From outside the module, the only way to create values of type *orderedSet* is by calling *empty* and *insert*. The only way to inspect the elements of an *orderedSet* is via *toList*.

27.6. **Type Conversions.** It often makes sense to convert a value of one type into a value of another type. Probably the most prominent example of this is converting an *Integer* to a *Real* and rounding a *Real* to an *Integer*.

Because type conversions are so ubiquitous, there is special notation for it in Babel-17. To convert a value v canonically to a value of type t, the notation

$v :> t$

is used. If t is a value of type `type`, you can write

$v :> (t)$

to express the conversion operation.

Your own object can implement conversion to a type t via

```
object
  ...
  def this :> t = ...
  ....
end
```

In case your object cannot convert to t, it should throw a *DomainError*-exception.

You can annotate your *def*-statements with a *return type* t, for example:

```
def myfun pat : t = expr
```

This is just another notation for

```
def myfun pat = (expr :> t)
```

27.7. **Automatic Type Conversions.** Type conversions are actually differentiated into two separate classes: those which are automatically applied, and those which are only applied in conjunction with the `:>` operator.

Your own object can implement automatic type conversion to a type t via

```
object
  ...
  def this : t = ...
  ....
end
```

Note that an automatic type conversion takes precedence over a non-automatic one when used in conjunction with the `:>` operator. In particular,

```
object
  def this : int = 1
  def this :> int = 2
end :> int
```

will evaluate to 1.

27.8. **Import.** A module can be accessed from anywhere via its module path. For example will

```
module hello.world
  def x = 2
end
hello.world.x
```

result in the value 2.

To avoid typing long module paths you can *import* them:

```
import hello.world.x
(x, x, x)
```

will evaluate to $(2, 2, 2)$.

You can also import types this way:

```
import util.orderedSet.orderedSet
def e : orderedSet = util.orderedSet.empty
```

Alternatively, to make the above statement even shorter, import *both* the type and its enclosing module:

```
import util.orderedSet
def e : orderedSet = orderedSet.empty
```

It is not possible to just import all members of a module; Babel-17 is dynamically typed, and the danger of accidental mayhem just would be too big.

You can rename imports, for example:

```
import util.orderedSet => set
def e : set = set.empty
```

You can combine several imports into a single one:

```
import util.orderedSet.{empty, insert => orderedInsert}
def e : set = set.empty
```

All imports must be grouped together at the beginning of a block, and later **import**-statements in this group take earlier **import**-statements in this group (and outside this group, of course) into account.

There is the **root**-keyword that denotes the module root. Instead of **import** util.orderedSet you could just as well say

```
import root.util.orderedSet
```

You can also use it in expressions, like in

```
val e = root.util.orderedSet.empty
```

The **root**-keyword comes in handy in situations when you want to explicitly make sure that there is no name aliasing going on:

```
import com.util
import util.orderedSet
```

here the second import actually imports com.util.orderedSet. If you want to import util.orderedSet instead, write

```
import com.util
import root.util.orderedSet
```

27.9. **Abstract Datatypes, Continued.** We have seen previously how to define our own abstract datatype *orderedSet*. This is already fine, but we would like orderedSet to integrate more tightly with how things work in Babel-17. In particular,

(1) ordered sets should be comparable in a way that makes sense; right now, the expression

```
insert (empty leq, 1) == insert (empty leq, 2)
```

evaluates to **true** because both values have outer value **nil**,

(2) instead of insert (r, x) we just want to write r + x,

(3) instead of toList r we want to write r :> list ,

(4) we want to be able to traverse the set via

```
for x in r do ... end
```

(5) and we want to be able to write

```
with empty leq do
  yield a
  yield b
  yield c
end
```

for the set consisting of the elements a, b and c.

We can achieve all that by replacing the line

```
typedef orderedSet (leq, list ) = nil
```

with the following code:

```
typedef orderedSet (leq, list ) =
  object

    ## (1)
    def compare_ (orderedSet (leq2, list2 )) = list ~ list2

    ## (2)
    def plus_ x = insert (this, x)

    ## (3)
    def this : list = list

    ## (4)
    def iterate_ =
        match list
            case [] => ()
            case (x::xs) => (x, orderedSet (leq, xs))
        end

    ## (5)
    def collector_close_ = this
    def collector_add_ x = this + x
    def empty = orderedSet (leq, [])

  end
```

28. Unit Tests

Unit testing has become a corner stone of most popular software development methodologies. Therefore Babel-17 provides language support for unit testing in form of the unittest keyword. This keyword can be used in two forms.

First, it can be used as part of the name of a module, for example like in

```
module util.orderedSet.unittest

import util.orderedSet._
```

```
def leq (a, b) = a <= b
def geq (a, b) = a >= b

#assert insert (insert (empty leq, 3), 1) :> list == [1, 3]
#assert insert (insert (empty geq, 3), 1) :> list == [3, 1]

end
```

Second, it can be used to separate a module definition into two parts:

```
module util.orderedSet

typedef orderedSet ...

 ...

unittest

def leq (a, b) = a <= b
def geq (a, b) = a >= b

#assert insert (insert (empty leq, 3), 1) :> list == [1, 3]
#assert insert (insert (empty geq, 3), 1) :> list == [3, 1]

end
```

The meaning of the above is

```
module util.orderedSet

typedef orderedSet ...

 ...

def unittest =
  begin

    def leq (a, b) = a <= b
    def geq (a, b) = a >= b

    #assert insert (insert (empty leq, 3), 1) :> list == [1, 3]
    #assert insert (insert (empty geq, 3), 1) :> list == [3, 1]

  end

end
```

except for the fact that this is not legal Babel-17.

The idea is that Babel-17 can run in two modes. In *production mode*, all unit tests are ignored. In *unit testing mode*, you can run your unit tests. Usually in production mode, you would switch off assertions, and in unit testing mode you would turn them on, but that is up to you.

An important detail is that production code is separated from unit testing code, i.e. while unit testing code can import other unit testing code and production code, production code can only import other production code. For example,

```
module mystuff

import util.orderSet.unittest => test

test ()

end
```

is not legal Babel-17, because the production code module *mystuff* cannot import the unit testing function *util.orderSet.unittest*. On the other hand, the following is valid Babel-17:

```
module mystuff.unittest

import util.orderSet.unittest => test

test ()

end
```

29. Native Interface

Babel-17 exposes the platform it is running on through its *native interface*. To query which platform you are running on, use the expression

```
native Platform
```

If there is no underlying platform you can access, the value **nil** is returned.

29.1. The Java Platform.

In case of

```
native Platform == Java
```

there is a limited form of Java interoperability available. With

val v = **native** New (*classname*, x_1, ..., x_n)

you can create a value v of type `native_` that is a wrapper for a Java object of class *classname*; the x_i are the arguments that are passed to the constructor. Table 10 describes how values are converted between Babel-17 and Java.

You can access the fields of v and call its methods by sending messages to v. This holds also true for static fields and messages. So for example, you can do the following:

```
val ar = native New ("java.util.ArrayList", 5)
val _ =
  begin
```

TABLE 10. Conversions between Babel-17 and Java Values

Babel-17	**Java**
nil	`null`
native_	`Object`
`int`	`byte Byte short Short char Char int Integer long Long BigInteger`
`real`	`float Float double Double`
`bool`	`boolean Boolean`
`string`	`String char Char`
`vect`	Java array

```
    ar.add 1
    ar.add 10
    ar.add 5
    ar.add ((18, 13, 15),)
  end
ar.toArray ()
```

This evaluates to `(1, 10, 5, (18, 13, 15))`.

Ambiguities are resolved in *some* way. The only rule you can rely on is that access to methods shadows access to fields. Note that ambiguities not only arise because of typing ambiguities, but also because messages in Babel-17 are case-insensitive, but in Java method and field names are case-sensitive.

You can instantiate a native value that corresponds to a Java class *classname* via

val c = **native** Class *classname*

The value *c* will in addition to the normal class fields and methods also understand messages that correspond to the static fields and methods of the class. For example:

```
val c = native Class "java.lang.Integer"
c.parseInt "120"
```

will evaluate to 120.

30. Standard Library

The standard library is an important part of the language definition of Babel-17 and consists of the built-in types of Babel-17. In future versions of Babel-17, also the module *lang* and all its submodules are part of the standard library.

The messages implemented by the built-in types of the standard library can be grouped as follows:

- collector related messages (Section 24),
- collection related messages (Section 25),
- *putback* messages that implement lens functionality,
- those messages that are listed as syntactic sugar in Table 8,
- messages specific to the type.

As a convention, all messages that are normally only used via some sort of syntactic sugar end with an underscore in their name.

Table 11. Additional messages for collection/collector c

Message	**Description**
c.isEmpty	checks if c is an empty collection
c.empty	an empty collection that has the same type as c
c.size	the size of collection c
$c + x$	adds x as member to collection c
c ++ d	adds all members of d as members to c
c - x	removes from c all members that are equal to x
c -- d	removes from c all members that are equal to an element in d
c ** d	removes from c all members that are not equal to an element in d
c.head	the first element of c
c.tail	all elements of c except the first one
c.atIndex i	the i-th element of c
c.indexOf x	the lowest i such that c.atIndex i == x
c.contains x	checks if c contains x
c.take n	forms a collection out of the first n elements of c
c.drop n	forms a collection by dropping the first n elements of c
c/f	maps the function f over all elements of c
$(c * f)\ a_0$	folds f over c via $a_{i+1} = f\ (c_i,\ a_i)$
c ^ f	filters c by boolean-valued function f
c // f	map created by all key/value pairs $(x,\ f(x))$ where x runs through the elements of c; later pairs overwrite earlier pairs

30.1. **Collections and Collectors.** The built-in types that can be used as collectors and collections are: List, Vector, Set, Map and String. In the case of maps, the elements of the collection are pairs (vectors of length 2) where the first element of the pair represents the key, and the second element the value the key is mapped to. For strings, the elements of the collection are strings of length 1, consisting of a single Unicode code point.

All built-in collection/collector types implement the messages described in Table 11.

30.2. **Type Conversions.** Many built-in types have conversions defined between them. Some of these type conversions are automatic, and some are not. Table 12 lists all of them. Note that a "yes" and "auto" do not necessarily mean that *all* values of the source type can be converted to the destination type. For example, `5^1000` cannot be converted to `real`, and `5.1` cannot be automatically converted to `int`, but nevertheless `5.1 :> int == 5` holds.

In the rest of this section we enumerate for each built-in type all messages that are supported by this type. Collector and collection messages are excluded from this enumeration.

TABLE 12. Type Conversions `src :> dest`

src \ **dest**	`int`	`bool`	`real`	`string`	`list`	`vect`	`set`	`map`
`int`	-	yes	auto	yes	no	no	no	no
`bool`	yes	-	no	yes	no	no	no	no
`real`	auto	no	-	yes	no	no	no	no
`string`	yes	yes	yes	-	yes	yes	yes	no
`list`	no	no	no	no	-	auto	yes	yes
`vect`	no	no	no	no	auto	-	yes	yes
`set`	no	no	no	no	yes	yes	-	yes
`map`	no	no	no	no	yes	yes	yes	-

TABLE 13. List/Vector-specific messages

$l\ i$	same as l.atIndex i
$-l$	reverses l

30.3. **Integer.** Integers can be arbitrarily large and support the usual operations (`+`, binary and unary `-`, `*`, `^`, **div**, **mod**). Division and modulo are Euclidean.

The expression a `to` b denotes the list of values a, $a+1$, …, b. The expression a `downto` b denotes the list of values a, $a-1$, …, b.

30.4. **Reals.** Reals are intervals bounded by floating point numbers. For more information on them see Section 18.

30.5. **String.** Implements no messages specific for strings. The semantics of the indexOf and contains messages is extended with respect to the usual collection semantics to not only search for strings of length 1, but arbitrarily long strings.

30.6. **List and Vector.** Those messages specific to lists and vectors are listed in Table 13.

30.7. **Set.** There are no messages specific to sets except that for a set s the expression $s\ x$ is equivalent to s.contains x and therefore tests if x is an element of s.

30.8. **Map.** The messages specific to maps are listed in Table 14. Note that the operators `-`, `--`, `**` and `//` deviate in their semantics from the usual collection semantics, but that the operators `+` and `++` *do* behave according to the usual collection semantics.

30.9. **Object.** Custom objects respond exactly to the set of messages defined in their body, with one exception: if the custom object implements all four of these messages:

- collector_add_
- collector_close_
- empty
- collector_iterate_

Table 14. Map-specific Messages

c.contains x	checks if c contains x
c.containsKey k	checks if c contains (k, v) for some v
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 elements of n
m -- n	map created from m by removing all keys that are elements of n
m ** n	map created from m by removing all keys that are not elements of n
m k	returns the value v associated with k in m, or returns a dynamic exception with parameter `DomainError` if no such value exists
m // f	map created by applying the function f to the key/value pairs $(k,\ v)$ of m, yielding key/value pairs $(k,\ f(k,\ v))$

then the custom object automatically inherits standard implementations for all messages listed in Table 11 and for the three type conversions to `list`,`vector` and `set`. These standard implementations are shadowed by actual implementations the custom object might provide.

31. What's Next?

Where will Babel-17 go from here? There are two dimensions along which Babel-17 has to grow in order to find adoption.

Maturity and breadth of implementations While there will probably always be a reference implementation of Babel-17 that strives for simplicity and clarity and sacrifices performance to achieve this, Babel-17 will need quality implementations on major computing platforms like JavaVM, Android, iOS and HTML 5. A lot of the infrastructure of these implementations can be shared, so once there is a speedy implementation of Babel-17 running for example on the Java Virtual Machine, the other implementations should follow more quickly. For example, there are many many opportunities for both static and particularly dynamic optimizations of the standard library implementation.

Development of the language Babel-17 is a dynamically typed purely functional programming language with a module system that has powerful encapsulation mechanisms. Some way of dealing with state, user interfaces, and communication with the outside world will be added to future versions of Babel-17.

References

[1] Milner; Tofte, Harper, MacQueen. *The Definition of Standard ML.*MIT Press, 1997.
[2] Odersky, Spoon, Venners. *Programming in Scala.*Artima Press, 2008.
[3] Babel-17. `http://www.babel-17.com`
[4] Mini Babel-17. `http://www.babel-17.com/Mini-Babel-17`
[5] Isabelle. `http://isabelle.in.tum.de`

[6] Foster, Greenwald, Moore, Pierce, Schmitt. Combinators for Bi-Directional Tree Transformations. *ACM Symposium on Principles of Programming Languages*, 2005.
[7] Edward Kmett. Lenses: A Functional Imperative. `http://www.youtube.com/watch?v=efv0SQNde5Q`
[8] Samuel R. Delany. *Babel-17*. Ace Books, 1966.