ℵ Programming Language

Volume 1 - Programmer's guide

Revision 0.8.1

Contents

Preface

This manual is part of the *Aleph Programming Language Series*, a multi volume set, describing the programming environment of the Aleph system.

- *Volume 0 - Aleph Installation Guide* is the distribution installation manual.
- *Volume 1 - Aleph Programmer Guide* is the first volume of this set. It is both an introduction and an advanced guide for the the developer.
- *Volume 2 - Aleph Library Reference* is the second volume of this set. It is a complete description of the Aleph standard library.

The Aleph programming language

Aleph is a multi-threaded functional programming language with dynamic symbol bindings that support the object oriented paradigm. **Aleph** features a state of the art runtime engine that supports both 32 and 64 bits platforms. **Aleph** comes with a rich set of libraries that are designed to be platform independent. **Aleph** is a free software. A flexible license has been designed for both individuals and corporations. Everybody is encouraged to use, distribute and/or modify the aleph engine for any purpose.

Features

The **Aleph** engine is written in C++ and provides runtime compatibility with it. Such compatibility includes the ability to instantiate C++ classes, use virtual methods and raise or catch exceptions. A comprehensive API has been designed to ease the integration of foreign libraries.

- Builtin objects
  More than 50 reserved keywords and predicates. Various containers like list, vector, hash table, bitset, and graphs.
- Functional programming
  Support for *lambda expression* with explicit closure. Symbol scope limitation with *gamma expression*. Form like notation with an easy block declaration.
- Object oriented
  Single inheritance object mechanism with dynamic symbol resolution. Native class derivation and method override. Static class data member and methods.
- Multi-threaded engine
  True multi-threaded engine with automatic object protection mechanism against concurrent access. Read and write locking system and thread activation via condition objects.
- Original regular expression
  Builtin regular expression engine with group matching, exact or partial match and substitution.

Aleph is a core language and libraries. The libraries are a specific set of classes and functions which are structured per application domains. **Aleph** is delivered with a set of standard libraries.

- **aleph-sys**
  The `aleph-sys` library is the system calls library. Standard classes and functions are provided to interact with the running machine.

- **aleph-sio**
  The `aleph-sio` library is the standard input/output All input/output operations are performed with this library.

- **aleph-net**
  The `aleph-net` library is the networking library. The library is based on the standard *Internet Protocol* and provides various classes to manipulates IP address, client or server sockets.

- **aleph-www**
  The `aleph-www` library is the World Wide Web library. The library provides various classes that ease the development of web applications or CGI scripts.

Aleph provides *extensions.* An extension is a library or an application which is not installed by default. The user selects during the installation process which extension is needed. For example, the static version of the interpreter is an extension.

Aleph engine

Aleph is an interpreted language. When used interactively, commands are entered on the command line and executed when a complete and valid syntactic object has been constructed. Alternatively, the interpreter can execute a source file. **Aleph** does not have a garbage collector. **Aleph** operates with a lazy, scope based, object destruction mechanism. Each time an object is no longer visible, it is destroyed automatically. At this time, the **Aleph** interpreter is unable to reclaim memory with circular structures. This is a well known problem when using a reference count mechanism. In the future, the **Aleph** engine will provide some mechanisms to resolve this problem.

Flexible Distribution

Aleph is a free software. A flexible license model encourages individuals or corporations to use, copy, modify and/or distribute this software. **Aleph** is designed by software professionals. Quality is one the driving force of the development effort. This is reflected in this distribution by the extensive documentation. A large test suite is used to assess the quality of the distribution. Right now, the engine has been successfully tested on most Linux platforms, Free BSD and Solaris.

License

Aleph is a free software. It can be used, modified and redistributed by anybody for personal or commercial use. The only restriction is altering the copyright notice associated with the material. Individual or corporation are permitted to use, include or modify the Aleph engine. All material developed with the Aleph language belongs to their respective copyright holder.

This program is free software. You can redistribute it and/or modify it provided that this copyright notice is kept intact. This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose. In no event shall the copyright holder be liable for any direct, indirect, incidental or special damages arising in any way out of the use of this software.

Chapter 1

Getting Started

This chapter is a quick introduction to the Aleph Programming Language, simply called **aleph**. The chapter describes the fundamental syntax without entering into the details which are described in later chapters.

1.1 First programs

The fundamental aleph syntactic object is a form. A form is parsed and immediately executed by the Aleph engine. A form is generally constructed with a function name and a set of arguments. The process of executing a form is called the **evaluation**. As a simple program, the traditional `"hello world"` program is shown below.

1.1.1 Hello world

The ”hello world” program is *de rigueur* when introducing a new programming language. Here is the aleph version of it.

```
aleph >println "hello world"
```

Aleph is an interpreted language. You can therefore invoke the interpreter and enter the above commands, or use your favorite editor and executes the file. By convention, an aleph source file has the extension *.als*. A simple session to run the above program (assuming the source file is called *hello.als*) is shown below.

```
zsh >aleph hello.als
hello world
```

You can also simply invoke **aleph** and enter the command interactively. The result will be the same. Simply type `Ctrl-D` to exit the session. Another way to operate is to call the **Aleph** compiler called **axc**, and then invoke the interpreter with the compiled file, even let the interpreter to figure this out. Note that the interpreter assume the *.axc* for compiled file.

```
zsh >axc   hello.als
zsh >aleph hello.axc
hello world
```

```
zsh >aleph hello
hello world
```

The order of search is determined by a special system called the *file resolver*. Its behavior is described in a special chapter of this manual.

1.1.2 Interpreter command

The **aleph** interpreter can be invoked with several options, a file to execute and program arguments. The `-h` option prints the various interpreter options.

```
zsh >aleph -h
usage: aleph [options] [file] [arguments]
             [-h]         print this help message
             [-v]         print version information
             [-i] path    add a path to the resolver
             [-f] assert  enable assertion checking
```

The `-v` option prints the interpreter version and operating system. The `-f` option turn on or off some additional options like assertion checking. The program arguments are illustrated later in this chapter. The `-i` option add a path to the interpreter file path resolver. Several `-i` options can be specified. The order of search is determined by the option order. The use of the resolver combined with the *librarian* is described in a specific chapter.

1.1.3 Interactive line editing

Line editing capabilities are supported when the interpreter is used interactively. Error messages are displayed in red if the terminal supports colors. The following table is a resume of the default key bindings.

Binding	**Description**
backspace	erase the previous character
delete	erase at the cursor position
insert	toggle insert with in-place
left	move the cursor to the left
rigth	move the cursor to the right
up	move up in the history list
down	move down in the history list
ctrl-a	move to the beginning of the line
ctrl-e	move to the end of the line
ctrl-u	clear the input line
ctrl-k	clear from the cursor position
ctrl-l	refresh the line editing

Table 1.1: Terminal default bindings

1.1.4 Command line arguments

The command line arguments to the interpreter are stored in a vector called `argv` which is part of the `interp` class. A complete discussion about class data member is covered in chapter 4. At this stage, just look at the example below which illustrates the use of the vector argument.

```
# argv.als
# print the argument length and the first one

println "argument length: " (interp:argv:length)
println "first argument : " (interp:argv:get 0)

zsh >aleph argv.als hello world
2
hello
```

1.1.5 Loading a source file

The interpreter class provides also the `load` method to load a source file. The argument must be a valid file path or an exception is raised. The `load` method returns `nil`. When the file is loaded, the interpreter input, output and error streams are used. The load operation read one form after another and executes them sequentially.

```
# load the source file fred.als
aleph >interp:load "fred.als"
```

If the file has been compiled the *axc* extension can be used instead. This force the interpreter to load the compiled version. If you are not sure, or do not care about which file is loaded, the extension can be omitted.

```
# load the compiled file fred.axc
aleph >interp:load "fred.axc"
# load whatever is found
aleph >interp:load "fred"
```

Without extension, the compiled file is searched first. If it is not found the source file is searched and loaded.

1.1.6 The compiler

axc is the *Aleph cross compiler*. It generates a binary file that can be run across platforms. The `-h` option prints the various interpreter options.

```
usage: axc [options] [files]
           [-h]      print this help message
           [-v]      print version information
           [-i] path add a path to the resolver
```

One or several files can be specified on the command line. The source file can be searched by the resolver by using the `-i` option.

1.1.7 Builtin objects

Aleph provides several builtin objects, namely Boolean, Integer, Real, Character and String. A builtin object can be constructed literally for each of these types. The best way to build such object is to bind it to a *symbol*. The **const** and **trans** reserved keywords are used to declare a new symbol. A symbol is simply a binding between a name and an object. Almost any standard characters can be used to declare a symbol.

```
const boolean   true      → true
const integer   1999      → 1999
const real      2000.0    → 2000.0
const string    "aleph"   → aleph
const char      'a'       → a
```

None of the symbols (or names) used in the program above are reserved keywords. In fact, the capitalize names are builtin objects (like Integer or String). The **const** reserved keyword creates a const symbol and returns the last evaluated object. As a consequence, nested **const** constructs are possible like `trans b (const a 1)`. The **trans** reserved keyword declare a new non-constant symbol. That is, the symbol can be changed. Note that it is the symbol which is marked constant, not the object.

```
trans a-symbol "hello world"  → "hello world"
trans a-symbol 2000           → 2000
println a-symbol              → 2000
```

1.1.8 Comments

Comments starts with the character '#'. All characters until the end of line are consumed. Comments can be placed anywhere in the source file. Comments entered during an interactive session are discarded.

1.1.9 Forms

The previous program was an illustration of the simplest form declaration, referred as *implicit form*. An implicit form is a single line command. When a command is becoming complex, the use of the standard form notation is more readable. The standard form uses the '(' and ')' characters to start and close a form. The previous programs could have been written with the standard form notation instead of the implicit one. The use of standard form notation versus the implicit is one is a matter of style and readability.

A form causes an *evaluation*. When a form is evaluated, each symbol in the form are evaluated to their corresponding internal object. Then the interpreter treats the first object of the form as the object to execute and the rest is the argument list for the calling object. The use of form inside a form is the standard way to perform recursive evaluation for complex expression.

```
const three (+ 1 2) → 3
```

The previous program defines a symbol which is initialized with the integer 3, that is the result of the computation (+ 1 2). The program shows also that Polish notation is used for arithmetic. If fact, '+' is a builtin operator which causes the arguments to be summed (if possible).

Evaluation can be nested as well as definition and assignation. When a form is evaluated, the result of the evaluation is made available to the calling form. If the result is obtained at the top level, that result is discarded.

```
const    b (trans a (+ 1 2))
assert a 3
assert b 3
trans  a 4
assert b 3
```

This program illustrates the mechanic of the evaluation process. The evaluation is done recursively. The `(+ 1 2)` form is evaluated as 3 and the result transmitted to the form `(trans a 3)`. This form not only creates the symbol 'a' and binds to it the integer 3, but returns also 3 (i.e. the result of the previous evaluation). Finally, the form `(const b 3)` is evaluated, that is, b is created and the result discarded. Internally, things are a little more complex, but the idea remains the same. This program illustrates also the usage of the **assert** keyword.

1.2 Lambda expression

A *lambda expression* is a function in the aleph terminology. The term come historically from Lisp to express the fact that a lambda expression is analog to the concept of expression found in the lambda calculus. There are various ways to create a lambda expression. A lambda expression is created with the **trans** reserved keywords. A lambda expression takes 0 or more arguments and return an object. A lambda expression is also an object by itself. When a lambda expression is called, the arguments are evaluated from left to right and placed on the interpreter eval stack. The function is then called and the object result is transmitted to the calling form. The use of **trans** vs **const** is explain later. As an example, we define the factorial of an integer in a recursive way.

```
# declare the factorial function
trans fact (n) (
  if (== n 1) 1 (* n (fact (- n 1))))

# compute factorial 5
println "factorial 5 = " (fact 5)
```

This program calls for several comments. First the **trans** keyword defines a new function object with one argument called `n`. The body of the function is defined with the **if** reserved keyword and can be easily understood. The function is called in the next form when the **println** reserved keyword is executed. Note that here, the call to `fact` produces an integer object, which is converted automatically by the **println** keyword.

1.2.1 Block form

The notation used in the `fact` program is the standard form notation originating from Lisp and the Scheme dialect. **Aleph** offers another notation called the *block form* notation with the use of the { and } characters. A block form is a syntactic notation where each form in the block form is executed sequentially. The form can be either an implicit or a regular form. The `fact` procedure can be rewritten with the block notation as illustrated below.

```
# declare the factorial procedure
trans fact (n) {
  if (== n 1) 1 (* n (fact (- n 1)))
}

# compute factorial 5
println "factorial 5 = " (fact 5)
```

Another way to create a lambda expression is via the reserved keyword **lambda**. Recall that a lambda expression is an object. So when such object is created, it can be bounded to a symbol. The factorial example could be rewritten with an explicit lambda call.

```
# declare the factorial procedure
const fact (lambda (n) (
  if (== n 1) 1 (* n (fact (- n 1)))))

# compute factorial 5
println "factorial 5 = " (fact 5)
```

Note that here, the symbol `fact` is a constant symbol. The use of `const` is rather reserved for *gamma expression*.

1.2.2 Gamma expression

A lambda expression can somehow becomes very slow during the execution, since the symbol evaluation is done within a set of nested call to resolve the symbols. In other words, each recursive call to a function creates a new symbol set which is linked with its parent. When the recursion is becoming deep, so is the path to traverse from the lower set to the top one. Aleph provides another mechanism called *gamma expression* which binds only the function symbol set to the top level one. The rest remains the same. Using a gamma expression can speedup significantly the execution. Here is the factorial program.

```
# declare the factorial procedure
const fact (n) (
  if (== n 1) 1 (* n (fact (- n 1))))

# compute factorial 5
println "factorial 5 = " (fact 5)
```

We will come back later to the concept of *gamma expression*. The use of the reserved keyword **const** to declare a *gamma expression* makes now sense. Since most function definitions are constant with one level, it was a language choice to implement this syntactic sugar. Note that **gamma** is a reserved keyword and can be used to create a gamma expression object. On the other hand, note that the gamma expression mechanism does not work for instance method. We will illustrate this point later in the book.

1.2.3 Lambda generation

A lambda expression can be used to generate another lambda expression. In other word, a function can generate a function, hence the term that Aleph is a *functional programming* language. Suppose one might want to write a function which take an argument and generate a function which add this argument to the generated function argument (got that!). Here is the program.

```
# a gamma which creates a lambda
const gen (n) (
  lambda (x) (n) (+ x n))

# create a function which add 2 to its argument
const add-2 (gen 2)

# call add-2 with an argument and check
println "result = " (add-2 3)
```

The interesting part in the previous program is the concept of closed variables. Looking at the lambda expression inside `gen`, notice that the argument to the gamma is `x` while `n` is marked in a form before the body of the gamma. This notation indicates that the gamma should retain the value of the argument `n` when the closure is created. In the literature, you might discover a similar mechanism referenced as a *closure*. A closure is simply a variable which is closed under a certain context. When a variable is reference in a context without any definition, such variable is called a *free variable*. We will see later more programs with closures. Note that in the Aleph terminology, it is the object created by the gamma call which is called a closure. Note also that the same mechanism apply with lambda.

In short, a lambda expression is a function with or without closed variables, which works with nested symbol sets (or namesets). A gamma expression is a function with or without closed variable which is binded to the top level nameset. The reserved keyword **trans** binds a lambda expression. The reserved keyword **const** binds a gamma expression. A gamma expression cannot be used as an instance method.

1.2.4 Multiple arguments binding

A lambda or gamma expression can be defined to work with extra arguments using the special `args` binding. During a lambda or gamma expression execution, the special symbol `args` is defined with the extra arguments passed at the call. For example, a gamma expression with 0 formal argument and 2 actual arguments has `args` defined as a cons cell.

```
const proc-nilp nil {
  trans result 0
  for (i) (args) (result:+= i)
  eval result
}
assert 3 (proc-nilp 1 2)
assert 7 (proc-nilp 1 2 4)
```

`args` can also be defined with formal arguments. In that case, `args` is defined as a cons cell with the remaining actual arguments.

```
# check with arguments
const proc-args (a b) {
  trans result (+ a b)
  for (i) (args) (result:+= i)
  eval result
}
assert 3 (proc-args 1 2)
assert 7 (proc-args 1 2 4)
```

1.3 Nameset and bindings

A nameset is a container of bindings between a name and *symbolic variable*. We use the term *symbolic variable* to denote any binding between a name and an object. There are various ways to express such bindings. The common one in Aleph is called a symbol. Another type of binding is an argument. Despite the fact they are different, they share a set of common properties, like being settable. Another point to note is the nature of the nameset. As a matter of fact, Aleph has various type of namesets. The top level nameset is called a *global set* and is designed to handle a large number of symbols. In a lambda or gamma expression, the nameset is called a *local set* and is designed to be fast with a small number of symbols. The moral of this little story is to think always in terms of namesets, no matter how it is implemented. All namesets support the concept of parent binding. When a nameset is created (typically during the execution of a lambda expression), this nameset is linked with its parent one. This means that a symbol lookup is done by traversing all nameset from the bottom to the top and stopping when one is found. In term of Aleph notation, the *current nameset* is referenced with the special symbol **.**. The *parent nameset* is referenced with the special symbol **..**. The *top level nameset* is referenced with the symbol **...**.

1.3.1 Symbol

A symbol is an object which defines a binding between a name and an object. When a symbol is evaluated, the evaluation process consists in returning the associated object. There are various ways to create or set a symbol, and the different reserved keywords account for the various nature of binding which has to be done depending on the current nameset state. One of the symbol property is to be *const* or not. When a symbol is marked as a constant, it cannot be modified. Note here that it is the symbol which is constant, not the object. A symbol can be created with the reserved keywords **const** or **trans**.

1.3.2 Creating a nameset

A nameset is an object which can be constructed directly by using the object construction notation. Once the object is created, it can be binded to a symbol. Here is a nameset called `example` in the top level nameset.

```
# create a new nameset called example
const example (nameset .)

# bind a symbol in this nameset
const example:hello "hello"
println example:hello
```

1.3.3 Qualified name

In the previous example, a symbol is referenced in a given nameset by using a *qualified name* like `example:hello`. A qualified name define a path to access a symbol. The use of qualified name is a powerful notation to reference an object in reference to another object. For example, the qualified name `.:hello` refers to the symbol `hello` in the current nameset. The qualified name `...:hello` refers to the symbol `hello` in the top level nameset. There are other use for qualified names, like method call with an instance.

1.3.4 Symbol binding

The `trans` reserved keyword has been shown in all previous example. `trans` creates or set a symbol in the current nameset. For example, the form `trans a 1` is evaluated as follow. First, a symbol named `a` is searched in the current nameset. At this stage, two situations can occur. If the symbol is found, it is set with the corresponding value. If the symbol is not found, it is created in the current nameset and set. The use of qualified name is also permitted (and encouraged) with **trans**. The exact nature of the symbol binding with a qualified name depends on the partial evaluation of the qualified name. For example, `trans example:hello 1` will set or create a symbol binding in reference to the `example` object. If *example* refers to a nameset, the symbol is binded in this nameset. If `example` is a class, `hello` is binded as a static symbol for that class. In theory, there is no restriction to use **trans** on any object. If the object does not have a symbol binding capability, an exception is raised. For example, if `n` is an integer object, the form `trans n:i 1` will fail.

With 3 or 4 arguments, `trans` defines automatically a lambda expression. This notation is a syntactic sugar. The lambda expression is constructed from the argument list and bounded to the specified symbol. The rule used to set or define the symbol are the same as described above.

```
# create automatically a lambda expression
trans min (x y) (if (< x y) x y)
```

1.3.5 Constant binding

The `const` reserved keyword is similar to `trans`, except that it creates a *constant* symbol. Once the symbol is created, it cannot be changed. This `const` property is hold by the symbol itself. When trying to set a `const` symbol, an exception is raised. **const** works also with qualified names. The rules described previously are the same. When a partial evaluation is done, the partial object is called to perform a constant binding. If such capability does not exist, an exception is raised.

With 3 or 4 arguments, `const` defines automatically a *lambda* lambda expression. Like **trans** the rule are the same except that the symbol is marked constant.

```
# create automatically a const lambda expression
const max (x y) (if (> x y) x y)
```

1.3.6 Arguments

An expression argument is similar to a symbol, except that it is used only with function argument. The concept of binding between a name and an object is still the same, but with an argument, the object is not stored as part of the argument, but rather at another location (in fact, it is the execution stack). An argument can also be *constant.* On the other hand, a single argument can have multiple bindings. Such situation is found during the same function call in two different threads. An argument list is part of the lambda or gamma expression declaration. If the argument is defined as a constant argument a sub form notation is used to defined this matter. For example, the `max` gamma expression is given below.

```
# create a const gamma expression with const argument
const max (gamma ((const x) (const y)) (if (> x y) x y))
```

A special symbols named **args** is defined during a lambda or gamma expression evaluation with the remaining arguments passed at the time the call is made. The symbol can be either `nil` or bound to a list of objects.

```
const proc-args (a b)
  trans result (+ a b)
  for (i) (args) (result:+= i)
  eval result

assert 3 (proc-args 1 2)
assert 7 (proc-args 1 2 4)
```

1.4 Control flow

Aleph provides various reserved keywords which can be seen as standard *imperative statements*. Such statements are useful to write *readable* program but are not necessary the best in terms of efficiency. In most cases, a statement returns the last evaluated object. Most of the statements are *control flow* statements.

1.4.1 If statement

The **if** reserved keyword takes two or three arguments. The first argument is the boolean condition to check. If the condition evaluates to `true` the second argument is evaluated. The form return the result of such evaluation. If the condition evaluates to `false`, the third argument is evaluated or nil is returned if it does not exist. An interesting example which combines the `if` reserved keyword and a deep recursion is the computation of the *Fibonacci* sequence.

```
const fibo (gamma (n) (
  if (< n 2) n (+ (fibo (- n 1)) (fibo (- n 2))))
```

1.4.2 While statement

The **while** reserved keyword takes 2 arguments. The first is the loop condition. The second argument is the loop body. The first argument must evaluate to a boolean. The body is executed as long as the boolean condition is true. An interesting example related to integer arithmetic with a `while` loop is the computation of the *greatest common divisor or gcd*. We illustrate here its computation as describe by Knuth in the volume 2 of the *Art of Computer Programming*.

```
const gcd (u v) {
    while (!= v 0) {
      trans r (u:mod v)
      u:= v
      v:= r
    }
    eval u
}
```

Note in this previous example the use of the symbol =. The qualified name `u:=` is in fact a *method call*. Here, the integer `u` is assigned with a value. In this case, the symbol is not changed. It is the object which is muted.

1.4.3 Do statement

The **do** reserved keyword is similar to the **while** reserved keyword, except that the loop condition is evaluated after the body execution. The syntax call is opposite to the **while**. The first argument is the loop body and the second argument is the exit loop condition.

```
# count the number of digits in a string
const number-of-digits (s) {
    const len (s:length)
    trans index 0
    trans count 0
    do {
      trans c (s:get index)
```

```
    if (c:digit-p) (count:++)
  } (< (index:++) len)
  eval count
}
```

1.4.4 Loop statement

The **loop** reserved keyword is another form of loop. It take four arguments. The first is the initialize form. The second is the exit condition. The third is the step form and the fourth is the form to execute at each loop step. Unlike the `while` and `do` loop, the `loop` statement creates its own nameset, since the initialize condition generally creates new symbol for the loop only.

```
# a simple loop from 0 to 10
loop (trans i 0) (< i 10) (i:++) (println i)
```

1.4.5 Switch statement

The **switch** reserved keyword is a condition selector. The first argument is the switch selector. The second argument is a list of various value which can be matched by the switch value. A special symbol called **else** can be used to match any value.

```
# return the primary color in a rgb
const get-primary-color (color value) (
  switch color (
    ("red"   (return (value:substr 0 2))
    ("green" (return (value:substr 2 4))
    ("blue"  (return (value:substr 4 6))
  )
)
```

1.4.6 Return statement

The **return** reserved keyword indicates an exceptional condition in the flow of execution within a lambda or gamma expression. When a return is executed, the associated argument is returned and the execution terminates. If **return** is used at the top level, the result is simply discarded.

```
# initialize a vector with a value
const vector-init (length value) {
  # treat nil vector first
  if (<= length 0) return (Vector)
  trans result (Vector)
  do (result:add value) (> (length:--) 0)
}
```

1.4.7 Eval and protect

The **eval** reserved keyword forces the evaluation of the object argument. **eval** is typically used in a function body to return a particular symbol value. **eval** can also be used to force the evaluation of a *protected* object. In many cases, `eval` is more efficient than **return**. The **protect** reserved keyword constructs an aleph object without evaluating it. Typically when used with a form, **protect** return the form itself. **protect** can also be used to prevent a symbol evaluation. When used with a symbol, the symbol object itself is returned.

```
const add (protect (+ 1 2)) → cons cell
(eval add)                  → 3
```

1.4.8 Assert statement

The **assert** reserved keyword check for equality between the two arguments and abort the execution in case of failure. By default, the assertion checking is turn off, and can be activated with the command option `-f assert`. Needless to say that **assert** is used for debugging purpose.

```
assert true   (> 2 0)
assert 0      (- 2 2)
assert "true" (String true)
```

1.4.9 Block statement

The **block** reserved keyword executes a form in a new local set. The local set is destroyed at the completion of the execution. The **block** reserved keyword returns the value of the last evaluated form. Since a new local set is created, any new symbol created in this nameset is destroyed at the completion of the execution. In other word, the **block** reserved keyword allows the creation of a local scope.

```
trans a 1
block {
  assert    a 1
  trans     a (+ 1 1)
  assert    a 2
  assert ..:a 1
}
assert 1 a
```

1.5 Builtin objects

Aleph provides several builtin objects and builtin operators for arithmetic and logical operations. The `Integer` and `Real` classes are primarily used to manipulate numbers. The `Boolean` class is used to for boolean operation. Other builtin objects include `Character` and `String`. The exact usage of these classes is described in the next chapter.

1.5.1 Arithmetic operations

Aleph provides various ways to perform arithmetic operations. Chapter 2 gives a thorough discussion on the subject. The *global* arithmetic is mostly done with the **+** for add, **-** for substract, ***** for mult and **/** for divide. Each of these operators works with both integer and real number.

```
(+ 1 2)    →  3
(- 1)      → -1
(* 3 5.0)  → 15.0
(/ 4.0 2)  →  2.0
```

Predicate	Description
nil-p	return true with nil object
real-p	return true with real object
regex-p	return true with regex object
string-p	return true with string object
number-p	return true with number object
boolean-p	return true with boolean object
integer-p	return true with integer object
character-p	return true with character object

Table 1.2: Aleph builtin predicates

Predicate	Description
class-p	return true with class object
thread-p	return true with thread object
promise-p	return true with promise object
lexical-p	return true with lexical object
literal-p	return true with literal object
closure-p	return true with closure object
nameset-p	return true with nameset object
instance-p	return true with instance object
qualified-p	return true with qualified object

Table 1.3: Aleph special predicates

1.5.2 Logical operations

The `Boolean` class is used to represent the boolean value **true** and **false**. These last two symbols are builtin in the interpreter as `const` symbols. Aleph provides also some reserved keywords like `not`, `and` and `or`. Their usage is self understandable.

```
not true              → false
and true (== 1 0)     → false
or (< -1 0) (> 1 0)  → true
```

1.5.3 Predicates

A *predicate* is a function which returns a boolean object. Aleph provides several predicates to check for some builtin objects. By convention, a predicate terminates with the sequence `-p`. The **nil-p** predicate is a special predicate which returns true if the object is nil. Aleph provides a predicate for each builtin objects.

For example, one can write a function which returns true if the argument is a number, that is, an integer or a real number.

```
# return true if the argument is a number
const number-p (n) (
  or (integer-p n) (float-p n))
```

Predicates for *functional* and *symbolic* programming are also builtin into the Aleph engine.

Finally, for each object, a predicate is also associated. For example, `cons-p` is the predicate for the `Cons` object.

Predicate	Description
cons-p	return true for a cons object
list-p	return true for a list object
node-p	return true for a node object
edge-p	return true for an edge object
graph-p	return true for a graph object
queue-p	return true for a queue object
bitset-p	return true for a bitset object
vector-p	return true for a vector object

Table 1.4: Aleph special predicates

1.6 Class and Instance

Aleph provides support for the object oriented programming paradigm. A *class* in the aleph terminology is a nameset which can be binded automatically when an *instance* of that class is created. Compared to other language, there is no need to declare the data member for a particular class. Data members are created during the instance construction. A class allows an instance to call function with the instance nameset visible for that function.

1.6.1 Class and members

A class is declared with the reserved keyword **class**. The class acts like a nameset. Functions can be bounded to this class.

```
const Color (class)
const Color:BLACK "#000000"
const Color:WHITE "#FFFFFF"
```

Any object can be bounded as a data member, including lambda or gamma expressions.

```
const Color (class)
const Color:get-primary-from-string (color value) {
  trans val "0x"
  val:+= (switch color (
           ("red"   (value:substr 1 3))
           ("green" (value:substr 3 5))
           ("blue"  (value:substr 5 7))
         )
  )
  Integer val
}
```

1.6.2 Instance

An instance of a class is created like any builtin object. If a method called `initialize` is defined for that class, the method is used as a constructor of that instance.

```
const Color (class)
trans Color:initialize (red green blue) {
  const this:red   (Integer red)
```

```
  const this:green (Integer green)
  const this:blue  (Integer blue)
}

const red   (Color 255   0   0)
const green (Color   0 255   0)
const blue  (Color   0   0 255)
```

1.6.3 Instance method

When a lambda expression is bound to the class or the instance, that lambda can be invoked as an instance method. When an instance method is invoked, the instance nameset is set as the parent nameset for that lambda. This is the main reason why a gamma expression cannot be used as an instance method. The instance nameset defines the instance data members and the special symbol `this`.

```
const int-max (x y)
  if (> x y) (Integer x) (Integer y))

const Color:RED-FACTOR   0.75
const Color:GREEN-FACTOR 0.75
const Color:BLUE-FACTOR  0.75
trans Color:get-darker nil {
  trans red   (int-max (this:red:*   Color:RED-FACTOR)   0)
  trans green (int-max (this:green:* Color:GREEN-FACTOR) 0)
  trans red   (int-max (this:blue:*  Color:BLUE-FACTOR)  0)
  Color red green blue
}
# get a darker color than red
const dark-red (red:get-darker)
```

1.7 Miscellaneous features

Aleph provides several facilities for control flow and exceptional operations. Most of these features are available via the use of reserved keywords.

1.7.1 Iteration

An iteration facility is provided for some objects known as *iterable* objects. `Cons`, `List` and `Vector` are typical iterable objects. There are two ways to iterate with these objects. The first method uses the **for** reserved keyword. The second method uses an explicit iterator which can be constructed by the object.

```
# compute the scalar product of two vectors
const scalar-product (u v) {
  trans result 0
  for (x y) (u v) (result:+= (* x y))
  eval result
}
```

The **for** reserved keyword iterate on both object `u` and `v`. For each iteration, the symbol `x` and `y` are set with their respective object value. In the example above, the result is obtained by summing all intermediate products.

```
# test the scalar product function
const v1 (Vector 1 2 3)
const v2 (Vector 2 4 6)
(scalar-product v1 v2)  → 28
```

The iteration can be done explicitly by creating an iterator for each vectors and advancing steps by steps.

```
# scalar product with explicit iterators
const scalar-product (u v) {
  trans result 0
  trans u-it   (u:get-iterator)
  trans v-it   (v:get-iterator)
  while (u:valid-p) {
    trans x (u:get-object)
    trans y (v:get-object)
    result:+= (* x y)
    u:next
    v:next
  }
  eval result
}
```

In the example above, two iterators are constructed for both vectors `u` and `v`. The iteration is done in a `while` loop by invoking the `valid-p` predicate. The `get-object` method returns the object value at the current iterator position.

1.7.2 Exception

An *exception* is an unexpected change in the execution flow. The Aleph model for exception is based on a mechanism which throws the exception to be caught by a handler. The mechanism is also designed to be compatible with the native "C++" implementation.

An exception is thrown with the reserved keyword **throw**. When an exception is thrown, the normal flow of execution is interrupted and an object used to carry the exception information is created. Such exception object is propagated backward in the call stack until an exception handler catch it.

The reserved keyword **try** executes a form and catch an exception if one has been thrown. With one argument, the form is executed and the result is the result of the form execution unless an exception is caught. If an exception is caught, the result is the exception object. If the exception is a native one, the result is nil.

```
try (+ 1 2)                         → 3
try (throw)                         → nil
try (throw "hello")                 → nil
try (throw "hello" "world")         → nil
try (throw "hello" "world" "folks") → "folks"
```

The exception mechanism is also designed to install an exception handler and eventually retrieve some information from the exception object. The reserved symbol **what** can be used to retrieve some exception information.

```
# protected factorial
const fact (n) {
  if (not (integer-p n)) (throw "number-error" "invalid argument")
  if (== n 0) 1 (* n (fact (- n 1)))
}
# exception handler
const handler nil
  errorln what:eid ',' what:reason


(try (fact 5)       handler) → 120
(try (fact "hello") handler) → number-error, invalid argument
```

1.7.3 Delayed evaluation

The **Aleph** interpreter provides a special mechanism to delay an evaluation. The reserved keyword **delay** creates a special object called a *promise* which records the form to be later evaluated. The reserved keyword **force** causes a *promise* to be evaluated. Subsequent call with **force** will produce the same result.

```
trans   y 3
const   l ((lambda (x) (+ x y)) 1)
assert  4 (force l)
trans   y 0
assert  4 (force l)
```

1.7.4 Regular Expressions

The **Aleph** interpreter provides a builtin mechanism for regular expression. A *regex* is an object which is used to match certain text patterns. Regular expressions are built implicitly by the **Aleph** reader wit the use of the `[` and `]` characters.

```
if (== (const re [($d$d):($d$d)]) "12:31") {
  trans hr (re:get 0)
  trans mn (re:get 1)
}
```

In the previous example, *regex* is bind to the symbol `re`. The *regex* contains two groups as defined by the `(` and `)` characters. The call to the operator == returns `true` if the *regex* matches the argument string. The `get` method can be used to retrieve the group by index.

1.8 Threads

The Aleph interpreter provides a powerful mechanism which allows the concurrent execution of forms and the synchronization of shared objects. There are two types of threads, namely *normal thread* and *daemon thread*. They differ only by the interpreter exit condition. The interpreter will wait until all normal threads are completed. On the other hand, the interpreter will not wait for daemon threads. They are automatically stopped when all normal threads are finished. Normal threads are created with the reserved keyword **launch**, and daemon threads are created with the reserved keyword **daemon**. When threads are used, the interpreter manages automatically the shared objects and protect them against concurrent access. Chapter 7 describes in details the shared object behavior.

```
# shared variable access
const var 0

const decr nil (while true (var:= (- var 1)))
const incr nil (while true (var:= (+ var 1)))
const prtv nil (while true (println "value = " var))

# start 3 threads
launch (prtv)
launch (decr)
launch (incr)
```

1.8.1 Form synchronization

Although, Aleph provides an automatic synchronization mechanism for reading or writing an object, it is sometimes necessary to control the execution flow. There are basically two techniques to do so. First, protect a form from being executed by several threads. Second, wait for one or several threads to complete their task before going to the next execution step. The reserved keyword **sync** can be used to synchronize a form. When a form, is synchronized, the Aleph engine guarantees that only one thread will execute this form.

```
const print-message (code mesg) (
  sync {
     errorln "error  : " code
     errorln "message: " mesg
  }
)
```

The previous example create a gamma expression which make sure that both the error code and error message are printed in one group, when several threads call it.

1.8.2 Thread completion

The other piece of synchronization is the thread completion indicator. The thread descriptor contains a method called `wait` which suspend the calling thread until the thread attached to the descriptor has been completed. If the thread is already completed, the method returns immediately.

```
# simple flag
const flag false

# simple shared tester
const ftest (val) (flag) (assert val (flag:shared-p))

# no thread mean not shared
ftest false
```

```
# in a thread it is shared
const thr (launch (ftest true))
thr:wait
assert true (flag:shared-p)
```

This example is taken from the test suites. It checks that a closed variable becomes shared when started in a thread. Note the use of the `wait` method to make sure the thread has completed before checking for the shared flag. It is also worth to note that `wait` is one of the method which guarantees that a thread result is valid.

Another use of the `wait` method can be made with a vector of thread descriptors when one wants to wait until all of them have completed.

```
# shared vector of threads descriptors
const thr-group (Vector)

# wait until all threads in the group are finished
const wait-all nil (for (thr) (thr-group) (thr:wait))
```

1.8.3 Condition variable

A *condition variable* is another mechanism to synchronize several threads. A condition variable is modeled with the **Condvar** object. At construction, the condition variable is initialized to `false`. A thread calling the `wait` method will block until the condition becomes `true`. The `mark` method can be used by a thread to change the state of a condition variable and eventually awake some threads which are blocked on it. The use of condition variable is particularly recommended when one need to make sure a particular thread has been doing a particular task. A detailed description is given in the *thread* chapter.

Chapter 2

Numbers and Strings

This chapters covers in detail the builtin objects used to manipulate numbers and strings. First the integer, relatif and real numbers are described. **Aleph** offers a broad range of methods for these three objects to support numerical computation. As a second step, string and character objects are described. Many examples show the various operations which can be used as automatic conversion between one type and another. Finally, the boolean object is described. These objects belongs to the class of *literal* objects.

2.1 Integer

The fundamental number representation is the **Integer**. The aleph integer is a 64 bits signed 2's complement number. The objects described in this chapter are called literal since they always have a string representation.

2.1.1 Integer format

The default literal format for an integer is the decimal notation. The minus sign (without blank) indicates a negative number. Hexadecimal and binary notations can also be used with prefix `0x` and `0b`. The underscore character _ can be used to make the notation more readable.

```
const a  123         → 123
trans b -255         → 255
const h  0xff        → 255
const b  0b1111_1111 → 255
```

Integer number are constructed from the literal notation or by using an explicit integer instance. The Integer class offers standard constructors. The default constructor creates an integer object and initialize it to 0. The other constructors take either an integer, a real number, a character or a string.

```
const a (Integer)      → 0
const b (Integer 2000) → 2000
const c (Integer "23") → 23
```

When the hexadecimal or binary notation is used, care should be taken to avoid a negative integer. For example, `0x_8000_0000_0000_0000` is the smallest negative number. This number will never be positive.

2.1.2 Integer arithmetic

Standard arithmetic operators are available as builtin operators. The usual addition '+', multiplication '*' and division '/' operate with two arguments. The substraction '-' operates with one or two arguments.

```
(+ 3 4) →  7
(- 3 4) → -1
(- 3)   → -3
(* 3 4) → 12
(/ 4 2) →  2
```

As a builtin object, the `Integer` object offers various methods for builtin arithmetic which directly operates on the object. The following example illustrates these methods.

```
trans  i 0 → 0
(i:++)     → 1
(i:--)     → 0
(i:+ 4)    → 4
(i:= 4)    → 4
(i:- 1)    → 3
(i:* 2)    → 8
(i:/ 2)    → 2
(i:+= 1)   → 5
(i:-= 1)   → 4
(i:*= 2)   → 8
(i:/= 2)   → 4
```

As a side effect, these methods allows a const symbol to be modified. Since the methods operates on an object, they do not modify the state of the symbol. Such methods are called *mutable* methods.

```
const i 0 → 0
(i:= 1)   → 1
```

2.1.3 Integer comparison

The comparison operators works the same. The only difference is that they always return a Boolean result. The comparison operators are namely equal '==', not equal '!=', less than '<', less equal '<=', greater '>' and greater equal '>='. These operators take two arguments.

```
(== 0 1) → false
(!= 0 1) → true
(>  4 3) → true
(>= 4 3) → true
(<  4 3) → false
(<= 4 3) → false
```

Like the arithmetic methods, the comparison operators are supported as object methods. These methods return a boolean object.

```
(i:=  1) → 1
(i:== 1) → true
(i:!= 0) → true
(i:>  0) → true
(i:>= 0) → true
(i:<  2) → true
(i:<= 2) → true
```

2.1.4 Integer calculus

Armed with all these functions, it is possible to develop a battery of functions operating with numbers. As another example, we revisit the *Fibonacci* sequence as demonstrated in the previous chapter. Such example was terribly slow, because of the double recursion. Another method suggested by Springer and Friedman in uses two functions to complete the same job.

```
const fib-it (gamma (n acc1 acc2) (
    if (== n 1) acc2 (fib-it (- n 1) acc2 (+ acc1 acc2))))

const fiboi (gamma (n) (
    if (== n 0) 0 (fib-it n 0 1)))
```

This later example is by far much faster, since it uses only one recursion. Although, it is no the fastest way to write it, but nobody is going to question the elegant aspect of recursion.

2.1.5 Other Integer methods

The `Integer` class offers other convenient methods. The `odd-p` and `even-p` are predicates. The `mod` take one argument and returns the modulo between the calling integer and the argument. The `to-string` method returns a string representation of the integer. The `abs` methods returns the absolute value of the calling integer.

```
(i:= 2)         → 2
(i:even-p)      → true
(i:= 3)         → 3
(i:odd-p)       → true
(i:mod 2)       → 1
(i:= -1)        → -1
(i:abs)         → 1
(i:to-string)  → "-1"
```

2.2 Relatif Number

A *relatif or big-num* is an integer with infinite precision. The `Relatif` class is similar to the **Integer** class except that it works with infinitely long number. The relatif notation uses a `r` or `R` to express a relatif number versus an integer one.

```
const a  123R          → 123R
trans b -255R          → 255R
```

```
const c  0xffR          → 255R
const d  0b1111_1111R   → 255R
const e (Relatif)       → 0R
const f (Relatif 2000)  → 2000R
const g (Relatif "23")  → 23R
```

2.2.1 Relatif operations

Most of the integer operations are supported by the **Relatif** object. The only difference is that there is no limitation on the number size. This naturally comes with a computational price. An amazing example is to compute the biggest know *prime Mersenne number*. The world record exponent is **6972593**. The number is therefore.

```
const i 1R
const m (- (i:shl 6972593) 1)
```

This number has **2098960** digits. You can use the `println` method if you wish, but you have been warned...

2.3 Real Number

The **Real** type is another fundamental number representation for floating point number. The internal representation is machine dependent, and generally follows the double representation with 64 bits as specified by the IEEE 754-1985 standard for binary floating point arithmetic. All integer operations are supported for real numbers.

2.3.1 Real format

The Aleph reader supports two types of literal representation for real number. The first representation is the *dotted decimal* notation. The second notation is the *scientific notation*.

```
const a  123.0 # a positive real
const b -255.5 # a negative real
const c  2.0e3 # year 2000.0
```

Real number are constructed from the literal notation or by using an explicit real instance. The `Real` class offers standard constructors. The default constructor creates a real number object and initialize it to 0.0. The other constructors takes either an integer, a real number, a character or a string.

2.3.2 Real arithmetic

The real arithmetic is similar to the integer one. When an integer is added to a real number, that number is automatically converted to a real and vice versa. Ultimately, a pure integer operation might generate a real result. The example below is extracted from one of the Aleph test suite.

```
(+ 1999.0 1)   → 2000.0
(+ 1999.0 1.0) → 2000.0
(- 2000.0 1)   → 1999.0
(- 2000.0 1.0) → 1999.0
(* 1000 2.0)   → 2000.0
```

```
(* 1000.0 2.0) → 2000.0
(/ 2000.0 2)   → 1000.0
(/ 2000.0 2.0) → 1000.0
```

Like the `Integer` object, the `Real` object has arithmetic builtin methods.

```
trans  r 0.0 → 0.0
(r:++)       → 1.0
(r:--)       → 0.0
(r:+ 4.0)    → 4.0
(r:= 4.0)    → 4.0
(r:- 1.0)    → 3.0
(r:* 2.0)    → 8.0
(r:/ 2.0)    → 2.0
(r:+= 1.0)   → 5.0
(r:-= 1.0)   → 4.0
(r:*= 2.0)   → 8.0
(r:/= 2.0)   → 4.0
```

2.3.3 Real comparison

The comparison operators works as the integer one. As for the other operators, an implicit conversion between an integer to a real is done automatically.

```
(== 2000   2000)   → true
(!= 2000   1999)   → true
(<  1999.0 2000.0) → true
(<= 1999   2000)   → true
(<= 2000.0 2000)   → true
(>  2000   1999.0) → true
(>= 2000.0 2000.0) → true
```

Comparison methods are also available for the `Real` object. These methods take either an integer or a real as argument.

```
(r:=  1.0) → 1.0
(r:== 1.0) → true
(r:!= 0.0) → true
(r:>  0.0) → true
(r:>= 0.0) → true
(r:<  2.0) → true
(r:<= 2.0) → true
```

2.3.4 A complex example

One of the most interesting point with functional programming language is the ability to create complex computation function. For example, let's assume we wish to compute the value at a point `x` of the *Legendre polynom of order n*. One of the solution is to encode the function given its order. Another solution is to compute the function and then compute the value. Here is the implementation taken from the aleph test suite. Note that the recursive definition of a *Legendre polynom* is:

$P_0(x) = 1$
$P_1(x) = x$
$nP_n(x) = (2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x)$

```
# legendre polynom order 0 and 1
const lp-0 (gamma (x) 1)
const lp-1 (gamma (x) x)

# legendre polynom of order n
const lp-n (gamma (n) (
    if (> n 1) {
      const lp-n-1 (lp-n (- n 1))
      const lp-n-2 (lp-n (- n 2))
      gamma (x) (n lp-n-1 lp-n-2) (/ (- (* (* (- (* 2 n) 1) x) (lp-n-1 x))
                                       (* (- n 1) (lp-n-2 x))) n)
     } (if (== n 1) lp-1 lp-0)
))

# generate order 2 polynom
const lp-2 (lp-n 2)

# print lp-2 (2)
println "lp2 (2) = " (lp-2 2)
```

Note that the computation can be done either with integer or real numbers. With integers, you might get some strange results anyway, but it will work. Note also how the closed variable mechanism is used. The recursion capture each level of the polynom until it is constructed. As an exercise, try to use a lambda expression instead of a gamma one, and compare the execution result with large number of `n`. Note also that we have here a double recursion. try to rewrite the example by using the same technique demonstrated with the Fibonacci sequence and compare the execution time.

2.3.5 Other real methods

The real numbers are delivered with a battery of functions. These include the trigonometric functions, the logarithm and couple others. Hyperbolic functions like `sinh`, `cosh`, `tanh`, `asinh`, `acosh` and `atanh` are also supported. The `sqrt` method return the square root of the calling real. The `floor` and `ceiling` returns respectively the floor and the ceiling of the calling real.

```
const r0 0.0        → 0.0
const r1 1.0        → 1.0
const r2 2.0        → 2.0
const rn -2.0       → -2.0
const rq (r2:sqrt)  → 1.414213
const pi 3.1415926  → 3.141592

(rq:floor)          → 1.0
(rq:ceiling)        → 2.0
(rn:abs)            → 2.0
(r1:log)            → 0.0
(r0:exp)            → 1.0
```

```
(r0:sin)          → 0.0
(r0:cos)          → 1.0
(r0:tan)          → 0.0
(r0:asin)         → 0.0
(pi:floor)        → 3.0
(pi:ceiling)      → 4.0
```

2.3.6 Accuracy and formating

Real numbers are not necessarily accurate, nor precise. The accuracy and precision are highly dependent on the hardware as well as the nature of the operation being performed. In any case, never assume that a real value is an exact one. Most of the time, a real comparison will fail, even if the numbers are very close together. When comparing real numbers, it is preferable to use the `?=` operator. Such operator result is bounded by the internal precision representation and will generally return the desired value. The real precision is an interpreter value which is set with the `set-real-precision` method. The actual precision can be retrieved with the `get-real-precision` method. By default, the precision is set to 0.00001;

```
interp:set-real-precision 0.0001
const r 2.0
const s (r:sqrt) → 1.4142135
(s:?= 1.4142)    → true
```

Real number formating is another story. The `format` method takes a *precision argument* which indicates the number of digits to print for the decimal part. Note that the format command might round the result as indicated in the example below.

```
const pi 3.1415926535
pi:format 3  → 3.142
```

2.4 Character

The **Character** object is another builtin object of the aleph engine. A character is internally represented by a byte and has a literal representation.

2.4.1 Character format

The standard quote notation is used to represent a character. In that respect, **Aleph** differs substantially from other functional language where the quote protect a form (hence the name `protect` in **Aleph**).

```
const LA01 'a' # the character a
const ND10 '0' # the digit 0
```

All characters from the *iso-8859-1* standard are supported in a string. For a lexical name, the character set is restricted to a smaller set.

```
a b c d e f g h j i k l m n o p q r s t u v w x y z
A B C D E F G H J I K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9 . + - * / = > < ! ?
```

Characters are constructed from the literal notation or by using an explicit character instance. The `Character` class offers standard constructors. The default constructor creates a null character. The other constructors take either an integer, a character or a string. The string can be either a single character or the literal notation.

```
const nilc (Character)        → null character
const a    (Character 'a')    → 'a'
const 0    (Character 48)     → 0
const mul  (Character "*")    → '*'
const div  (Character "'/'") → '/'
```

2.4.2 Character arithmetic

A character is like an integer, except that it operates in the range 0 to 255. The character arithmetic is simpler compared to the integer one and no overflow or underflow checking is done. Note that the arithmetic operations take an integer as an argument.

```
(+ 'a' 1)     → 'b'
(- '9' 1)     → '8'
```

Character object methods are also provided for arithmetic operations. Like the standard `+'` and – operators, these methods take an integer as an argument.

```
trans  c 'a'  → 'a'
(c:++)        → 'b'
trans  c '9'  → '9'
(c:--)        → '8'
(c:+ 1)       → '9'
(c:- 9)       → '0'
```

2.4.3 Character comparison

Comparison operators are also working with the `Character` object. The standard operators are namely equal '==', not equal '!=', less than '<', less equal '<=', greater '>' and greater equal '>='. These operators take two arguments.

```
(== 'a' 'b') → false
(!= '0' '1') → true
(>  'b' 'a') → true
(>= 'A' 'a') → true
(<  '4' '3') → false
(<= '9' '0') → false
```

The comparison operators are supported as object methods. These methods return a boolean object and take a character as an argument.

```
(c:=  'A') → 'A'
(c:== 'A') → true
(c:!= 'a') → true
(c:>  'a') → true
(c:>= 'a') → true
(c:<  'Z') → true
(c:<= 'Z') → true
```

2.4.4 Other character methods

The `Character` object comes with additional methods. These are mostly conversion methods and predicates. The `to-string` method returns a string representation of the calling character. The `to-integer` method returns an integer representation the calling character. The predicates are `alpha-p`, `digit-p`, `blank-p`, `eol-p`, `eof-p` and `nil-p`.

```
const LA01 'a'     → 'a'
const ND10 '0'     → '0'
(LA01:to-string)   → "a"
(LA01:to-integer)  → 97
(LA01:alpha-p)     → true
(ND10:digit-p)     → true
```

2.5 String

The **String** object is one of the most important builtin object in the aleph engine. A string can be seen as a vector of characters. However, the internal representation of string is slightly different.

2.5.1 String format

The standard double quote notation is used to represent literally a string. Standard escape sequences are also accepted to construct a string.

```
const hello "hello" → true
```

Any literal object can be used to construct a string. This means that `Integer`, `Real`, `Boolean` and `Character` objects are all valid to construct strings. The default constructor creates a null string. The string constructor can also takes a string.

```
const nils (String)       → ""
const one  (String 1)     → "1"
const a    (String 'a')   → "a"
const b    (String true)  → "true"
```

2.5.2 String operations

With strings, numerous methods can be provided. We illustrate here the most common one. Volume 2 of this manual series contains a complete description of the `String` object. The global operators `+`, `==` and `!=` are supported for strings.

```
const h "hello"
(h:length)        → 5
(h:get 0)         → 'h'
(h:== "world")    → false
(h:!= "world")    → true
(h:+= " world"    → "hello world"
```

The `sub-left` and `sub-right` methods return a sub-string, given the position index. For `sub-left`, the index is the terminating index, while `sub-right` is the starting index, counting from 0.

```
const msg "hello world"
(msg:sub-left  5) → "hello"
(msg:sub-right 6) → "world"
```

The `strip`, `strip-left` and `strip-right` are methods used to strip blanks and tabs. `strip` combines both `strip-left` and `strip-right`.

```
const h "   a lot of blanks   "
(h:strip) → "a lot of blanks"
```

The `split` method returns a vector of strings by splitting the string according to a break sequence. By default, the break sequence is the blank, tab and newline characters. The break sequence can be one or more characters passed as one single argument to the method.

```
const str "hello:world"
const vec str:split ":"
println (vec:length) → 2
```

The `fill-left` and `fill-right` methods can be used to fill a string with a character up to a certain length. If the string is longer than the length, nothing happens.

```
const pi  3.1415926535  → 3.1415926535
const val (pi:format 4) → 3.1416
(val:fill-left '0' 9)   → 0003.1416
```

2.5.3 String hash value

Computing the hash value of a string is an interesting problem. The algorithm used by the aleph engine is shown as an example below. Note that the `hashid` method is builtin in the `String` object. The program shows both internal and computed values.

```
# compute string hashid
const hashid (s) {
  const len (s:length)
  trans cnt 0
  trans val 0
  trans sht 17
  do {
    # compute the hash value
    trans i (Integer (s:get cnt))
    val:= (val:xor (i:shl sht))
    # adjust shift index
    if (< (sht:-= 7) 0) (sht:+= 24)
  } (< (cnt:++) len)
  eval val
}
```

When run, *example 0203.als*, the following result is obtained.

```
# test our favorite string
const hello "hello world"
(hello:hashid) → 1054055120
(hashid hello) → 1054055120
```

As a side note, it is recommended to print the shift amount in the program. One may notice, that the value remains bounded by 24. Since we are xoring the final value, it does illustrate that the algorithm is design for a 32 bits machine. It will work as well with a 64 bits machine (and give the same result), but the full integer range is not used. As an exercise, try to rewrite it for a 64 bits machine and compare the result. Try also to compute the modulo with a prime number of your choice.

Chapter 3

Container Objects

This chapter covers the builtin container objects and more specifically, the *iterable* objects such like `Cons`, `List` and `Vector`. We start this chapter with the `Cons` class and move to the `List` and **Vector** objects. Special objects like **Queue** and **Bitset** are mentioned at the end.

3.1 Cons builtin object

Originally, a `Cons` object or *cons cell* have been the fundamental object of the Lisp or Scheme machine. The cons cell is the building block for list and is of great importance in **Aleph** as well. A `Cons` object or simply a *cons cell* is a simple element used to build linked list. The cons cell holds an object and a pointer to the next cons cell. The cons cell object is called `car` and the next cons cell is called the `cdr`. This notation, found in Lisp in maintained here for the sake of tradition.

3.1.1 Cons cell constructors

The default constructor creates a cons cell those car is initialized to the nil object. The constructor can also take one or several objects.

```
const nil-cons (Cons)
const lst-cons (Cons 1 'a' "hello")
```

The constructor can take any kind of objects. When all objects have the same type, the result list is said to be *homogeneous*. If all objects do not have the same type, the result list is said to be *heterogeneous*. Cons list can also be constructed directly from the Aleph reader. Since all internal forms are built with cons cell, the construction can be achieved by simply *protecting* the form from being interpreted.

```
const blist (protect ((1) ((2) ((3)))))
```

3.1.2 Cons cell methods

A `Cons` object provides several methods to access the `car` and the `cdr` of a cons cell. Other methods allows access to a cons list by index.

```
const c (Cons "hello" "world")
(c:length)    → 2
(c:get-car)   → "hello"
(c:get-cadr)  → "world"
(c:get 0)     → "hello"
(c:get 1)     → "world"
```

The `set-car` method set the car of the cons cell. The `append` method appends a new cons cell at the end of the cons list and set the car with the specified object.

3.2 List builtin object

The `List` builtin object provides the facility of a double-link list. The `List` object is another example of *iterable* object. The `List` object provides support for forward and backward iteration.

3.2.1 List construction

A list is constructed like a cons cell with zero or more arguments. Unlike the cons cell, the `List` can have a null size.

```
const nil-list (List)
const dbl-list (List 1 'a' "hello")
```

3.2.2 List methods

`List` methods are similar the `Cons` object. The `append` method appends an object at the end of the list. The `insert` method inserts an object at the beginning of the list.

```
const list (List "hello" "world")
(list:length)          → 2
(list:get 0)           → "hello"
(list:get 1)           → "world"
(list:append "folks") → "hello" "world" "folks"
```

3.3 Vector builtin object

The `Vector` builtin object provides the facility of an index array of objects. The `Vector` object is another example of *iterable* object. The `Vector` object provides support for forward and backward iteration.

3.3.1 Vector construction

A vector is constructed like a cons cell or a list. The default constructor creates a vector with 0 objects.

```
const nil-vector (Vector)
const obj-vector (Vector 1 'a' "hello")
```

3.3.2 Vector methods

`Vector` methods are similar to the `List` object. The `append` method appends an object at the end of the vector. The `set` method set a vector position by index.

```
const vec (Vector "hello" "world")
(vec:length)          → 2
(vec:get 0)           → "hello"
(vec:get 1)           → "world"
(vec:append "folks")  → "hello" "world" "folks"
(vec:set 0 "bonjour") → "bonjour" "world" "folks"
```

3.4 Iteration

When an object is *iterable*, it can be used with the reserved keyword **for**. **for** iterates on one or several objects and binds associated symbols during each step of the iteration process. All iterable objects provides also the method `get-iterator` which returns an iterator for a given object. The use of iterator is justify during backward iteration, since **for** only perform forward iteration.

3.4.1 Function mapping

Given a function `func`, it is relatively easy to apply this function to all objects of an iterable object. The result is a list of successive calls with the function. Such function is called a mapping function and is generally called `map`.

```
const map (obj func) {
  trans result (Cons)
  for (car) (obj) (result:link (func car))
  eval result
}
```

The `link` method differs from the `append` method in the sense that the object to append is set to the cons cell car if the car and cdr is nil.

3.4.2 Multiple iteration

Multiple iteration can be done with one call to **for**. The computation of a scalar product is a simple but illustrative example.

```
# compute the scalar product of two vectors
const scalar-product (u v) {
  trans result 0
  for (x y) (u v) (result:+= (* x y))
  eval result
}
```

Note that the function `scalar-product` does not make any assumption about the object to iterate. One could compute the scalar product between a vector a list for example.

```
const u (Vector 1 2 3)
const v (List   2 3 4)
(scalar-product u v) → 28
```

3.4.3 Conversion of iterable objects

The use of an iterator is suitable for direct conversion between one object and another. The conversion to a vector can be simply defined as indicted below.

```
#convert an iterable object to a vector
const to-vector (obj) {
  trans result (Vector)
  for (i) (obj) (result:append i)
  eval result
}
```

3.4.4 Explicit iterator

An explicit iterator is constructed with the `get-iterator` method. At construction, the iterator is reset to the beginning position. The `get-object` method returns the object at the current iterator position. The `next` advances the iterator to its next position. The `valid-p` method returns `true` if the iterator is in a valid position. When the iterator supports backward operations, the `prev` method move the iterator to the previous position. Note that `Cons` objects do not support backward iteration. The `begin` method reset the iterator to the beginning. The `end` method moves the iterator the last position. This method is available only with backward iterator.

```
# reverse a list
const reverse-list (obj) {
  trans result (List)
  trans itlist (obj:get-iterator)
  itlist:end
  while (itlist:valid-p) {
    result:append (itlist:get-object))
    itlist:prev
  }
  eval result
}
```

3.5 Special Object

Aleph provides several builtin container objects which have proven to be useful. Such objects are **Queue** and **Bitset**.

3.5.1 Queue object

A *queue* is a special object which acts as container with a *fifo* policy. When an object is placed in the queue, it remains there until it has been dequeued.

```
# create a queue with objects
const q (Queue "hello" "world")
q:empty-p → false
q:length  → 2

# dequeue some object
q:dequeue → hello
```

```
q:dequeue → world
q:empty-p → true
```

3.5.2 Bitset object

A bitset is a special container for bit. A bitset can be constructed with a specific size. When the bitset is constructed, each bit can be marked and tested by index.

```
# create a bitset
const bs (BitSet)
bitset-p bs → true

# check, mark and clear
assert false (bs:get 0)
bs:mark 0
assert true  (bs:get 0)
bs:clear 0
assert false (bs:get 0)
```

Chapter 4

Class

This chapter covers the Aleph class model and its associated operations. The Aleph class model is slightly different compared to traditional one. Because Aleph has dynamic symbol bindings, it is not necessary to declare the class data members. A class is an Aleph object which can be manipulated by itself. Such class is said to belongs to a group of *meta class* as described later in this chapter. Once the class concept has been detailed, the chapter moves to the concept of instance of that class and shows how instance data members and functions can be used. The chapter terminates with a description of dynamic class programming.

4.1 The Class object

A *class* in the Aleph terminology is simply a nameset which can be replicated via a construction mechanism. A class is created with the reserved keyword **class**. The result is an object of type `Class` which supports various symbol binding operations.

4.1.1 Class declaration and binding

A new class is an object created with the reserved keyword **class**. Such class is an object which can be bound to a symbol.

```
const Color (class)
```

A list of initial instance data members can be specified as an argument to the **class** reserved keyword.

```
const Complex (class (re im))
```

Because a class acts like a nameset, it is possible to bind directly symbols with the *qualified name* notation.

```
const Color (class)             → <Class object>
const Color:RED-FACTOR    0.75 → 0.75
const Color:BLUE-FACTOR   0.75 → 0.75
const Color:GREEEN-FACTOR 0.75 → 0.75
```

When a data is defined in the class nameset, it is common to refer it as a *static data member*. A static data member is invariant over the instance of that class. When the data member is declared with the **const** reserved keyword, the symbol binding is `const` in the class nameset. It is also possible to use the **trans** reserved keyword.

4.1.2 Class closure binding

A lambda or gamma expression can be define for a class. If the class do not reference an instance of that class, the resulting closure is called a *static method* of that class. Static methods are invariant among the class instances. The standard declaration syntax for a lambda or gamma expression is still valid with a class.

```
const Color:get-primary-from-string (color value) {
  trans val "0x"
  val:+= (switch color (
           ("red"   (value:substr 1 3))
           ("green" (value:substr 3 5))
           ("blue"  (value:substr 5 7))
         )
  )
  Integer val
}
```

The invocation of a static method is done with the standard *qualified name* notation.

```
(Color:get-primary-from-string "red"   "#23c4e5") → 0x23
(Color:get-primary-from-string "green" "#23c4e5") → 0xc4
(Color:get-primary-from-string "blue"  "#23c4e5") → 0xe5
```

4.1.3 Class symbol access

A class acts as a nameset and therefore provides the mechanism to evaluate any symbol with the *qualified name* notation.

```
const Color:RED-VALUE "#ff0000" → "#ff0000"
const Color:print-primary-colors (color) {
  println "red   color " (Color:get-primary-color "red"   color)
  println "green color " (Color:get-primary-color "green" color)
  println "blue  color " (Color:get-primary-color "blue"  color)
}

# print the color components for the red color
Color:print-primary-colors Color:RED-VALUE
```

4.2 Instance

An *instance* of a class is an Aleph object which is constructed by a special class method called a *constructor*. If an instance constructor does not exist, the instance is said to have a default construction. An instance acts also as a nameset. The only difference with a class, is that a symbol resolution is done first in the instance nameset and then in the instance class. As a consequence, creating an instance is equivalent to define a default nameset hierarchy.

4.2.1 Instance construction

By default, a instance of the class is an object which defines an instance nameset. The simplest way to define an anonymous instance is to create it directly.

```
const i     ((class))
const Color (class)
const red   (Color)
```

The example above define an instance of an anonymous class. If a class object is bound to a symbol, such symbol can be used to create an instance of that class. When an instance is created, the special symbol named **this** is defined in the instance nameset. This symbol is bounded to the instance object and can be used to reference in an anonymous way the instance itself.

4.2.2 Instance initialization

When an instance is created, the Aleph engine looks for a special lambda expression called `initialize`. This lambda expression, if it exists, is executed after the default instance has been constructed. Such lambda expression is a method since it can refer to the **this** symbol and bind some instance symbols. The arguments which are passed during the instance construction are passed to the **initialize** method.

```
const Color (class)
trans Color:initialize (red green blue) {
  const this:red   (Integer red)
  const this:green (Integer green)
  const this:blue  (Integer blue)
}
# create some default colors
const Color:RED   (Color 255   0   0)
const Color:GREEN (Color   0 255   0)
const Color:BLUE  (Color   0   0 255)
const Color:BLACK (Color   0   0   0)
const Color:WHITE (Color 255 255 255)
```

In the example above, each time a color is created, a new instance object is created. The constructor is invoked with the **this** symbol bound to the newly created instance. Note that the qualified name `this:red` defines a new symbol in the instance nameset. Such symbol is sometimes referred as an *instance data member*. Note as well that there is no ambiguity in resolving the symbol `red`. Once the symbol is created, it shadows the one defined as a constructor argument.

4.2.3 Initialization with data member list

If the class was defined with a list of data members, the instance is created with these data members initialized to nil. Each symbol is defined as a transient symbol since they are supposed to be modified later. As a consequence, it is possible to use the reserved keyword **trans** inside the `initialize` method.

```
const Complex (class (re im))
trans Complex:initialize (re im)
  trans this:re (Real re)
  trans this:im (Real im)
```

The use of a class data member list is primarily dictated by the existence of a *copy constructor* for that class. If a method try to construct an object, an evaluation of an unbound data member with **trans** might trigger an inner instance data member to be set instead of the real one. This behavior exists only with **trans**. When **const** is used, the implementation guarantee that the symbol binding will be local to that instance.

4.2.4 Instance symbol access

An instance acts as a nameset. It is therefore possible to bind locally to an instance a symbol. When a symbol needs to be evaluated, the instance nameset is searched first. If the symbol is not found, the class nameset is searched. When an instance symbol and a class symbol have the same name, the instance symbol is said to shadow the class symbol. The simple example below illustrates this property.

```
const c   (class)
const c:a 1      → 1
const i   (c)
const j   (c)
const i:a 2      → 2

# class symbol access
println   c:a    → 1
# shadow symbol access
println   i:a    → 2
# non shadow access
println   j:a    → 1
```

When the instance is created, the special symbol **meta** is bound in the instance nameset with the instance class object. This symbol can therefore be used to access a shadow symbol

```
const c   (class)
const i   (c)
const c:a 1        → 1
const i:a 2        → 2
println   i:a      → 2
println   i:meta:a → 1
```

The symbol **meta** must be used carefully, especially inside constructor since it might create an infinite recursion as shown below.

```
const c (class)
trans c:initialize nil (const i (this:meta))
const i (c)
```

4.2.5 Instance method

When lambda expression is defined within the class or the instance nameset, that lambda expression is callable from the instance itself. If the lambda expression uses the **this** symbol, that lambda is called an instance method since the symbol **this** is defined in the instance nameset. If the instance method is defined in the class nameset, the instance method is said to be *global* (i.e. callable by any instance of that class). If the method is defined in the instance nameset, that method is said to be *local* and is callable by the instance only. Due to the nature of the nameset parent binding, only lambda expression can be used. Gamma expressions will not work since the gamma nameset has always the top level nameset as its parent one.

```
const Color (class)
# class constructor
trans Color:initialize (red green blue) {
  const this:red   (Integer red)
  const this:green (Integer green)
  const this:blue  (Integer blue)
}
const Color:RF 0.75
const Color:GF 0.75
const Color:BF 0.75
# this method returns a darker color
trans Color:darker nil {
  trans lr (Integer (max (this:red:*   Color:RF) 0))
  trans lg (Integer (max (this:green:* Color:GF) 0))
  trans lb (Integer (max (this:blue:*  Color:BF) 0))
  Color lr lg lb
}
# get a darker color than yellow
const yellow     (Color 255 255 0)
const dark-yellow (yellow:darker)
```

4.2.6 Instance operators

Any operator can be defined at the class or the instance level. Operators like **==** or **!=** generally requires the ability to assert if the argument is of the same type of the instance. The global operator **==** will return true if two classes are the same. With the use of the **meta**, it is possible to assert such equality.

```
# this method checks that two colors are equals
trans Color:== (color)
  if (== Color color:meta)
    if (!= this:red   color:red)   (return false)
    if (!= this:green color:green) (return false)
    if (!= this:blue  color:blue)  (return false)
    eval true
   false

# create a new yellow color
const  yellow (Color 255 255 0)
(yellow:== (Color 255 255 0))    → true
```

The global operator **==** returns `true` if both arguments are the same, even for classes. Method operators are left open to the user.

4.2.7 Complex number example

As a final example, a class simulating the behavior of a complex number is given hereafter. The interesting point to note is the use of the operators.

```
# class declaration
const Complex (class (re im))
trans Complex:initialize (re im) {
  trans this:re (Real re)
  trans this:im (Real im)
}

# class mutators
trans Complex:set-re (x) (trans this:re re)
trans Complex:set-im (x) (trans this:im im)

# class accessors
trans Complex:get-re nil (Real this:re)
trans Complex:get-im nil (Real this:im)
trans Complex:module nil {
  trans result (Real (+ (* this:re this:re) (* this:im this:im)))
  result:sqrt
}
trans Complex:format nil {
  trans result (String this:re)
  result:+= "+i"
  result:+= (String this:im)
}

# complex predicate
const complex-p (c) (
  if (instance-p c) (== Complex c:meta) false)

# operators
trans Complex:== (c) (
  if (complex-p c) (and (this:re:== c:re) (this:im:== c:im)) (
    if (number-p c) (and (this:re:== c) (this:im:zero-p)) false))

trans Complex:= (c) {
  if (complex-p c) {
    this:re:= (Real c:re)
    this:im:= (Real c:im)
    return this
  }
  this:re:= (Real c)
  this:im:= 0.0
  return this
}

trans Complex:+ (c) {
  trans result (Complex this:re this:im)
  if (complex-p c) {
    result:re:+= c:re
    result:im:+= c:im
    return result
  }
```

```
  result:re:+= (Real c)
  eval result
}

# create some complex number
const c1 (Complex 0.0 0.0)
const c2 (c1:+ 1)
println "complex [0.0:0.0] = " (c1:format)
println "complex [1.0:0.0] = " (c2:format)
```

4.3 Inheritance

Inheritance is the mechanism by which a class or an instance inherits methods and data member access from a parent object. The Aleph class model is based on a single inheritance model. When an instance object defines a parent object, such object is called a *super instance.* The instance which has a super instance is called a *derived instance.* The main utilization of inheritance is the ability to reuse methods for that super instance.

4.3.1 Derivation construction

A derived object is generally defined within the `initialize` method of that instance by setting the `super` data member. `super` is set to nil at the instance construction. The good news is that any object can be defined as a super instance, including builtin object.

```
const c (class)
const c:initialize nil {
  trans this:super 0
}
```

In the example above, an instance of class `c` is constructed. The super instance is with an integer object. As a consequence, the instance is derived from the `Integer` instance.

4.3.2 Derived symbol access

When an instance is derived from another one, any symbol which belongs to the super instance can be access with the use of the `super` data member. If the super class can evaluate a symbol, that symbol is resolved automatically by the derived instance.

```
const c       (class)
const i       (c)
trans i:a     1
const j       (c)
trans j:super i
println j:a   → 1
```

When a symbol is evaluated, a set of search rules is applied. Aleph gives the priority to the class nameset vs the super instance. As a consequence, a static data member might shadow a super instance data member. The rule associated with a symbol evaluation can be summarized as follow.

- Look in the instance nameset.

- Look in the class nameset.
- Look in the super instance if it exists.
- Look in the base object.

Chapter 5

Advanced Concepts

This chapter covers advanced concepts of the Aleph programming language. The first subject is the exception model. The second subject covers some properties of the namesets. Finally, the interpreter object is described in details.

5.1 Exception

An *exception* is an unexpected change in the execution flow. The Aleph model for exception is based on a mechanism which throws the exception to be caught by a handler. The mechanism is also designed to be compatible with the native "C++" implementation.

5.1.1 Throwing an exception

An exception is thrown with the reserved keyword **throw**. When an exception is thrown, the normal flow of execution is interrupted and an object used to carry the exception information is created. Such exception object is propagated backward in the call stack until an exception handler catch it.

```
if (not (number-p n)) (throw "type-error" "invalid object found" n)
```

The example above is the general form to throw an exception. The first argument is the *the exception id*. The second argument is the *exception reason*. The third argument is the *exception object*. The exception id and reason are always a string. The exception object can be any object which is carried by the exception. The reserved keyword **throw** accepts 0 or more arguments.

```
throw
throw "type-error"
throw "type-error" "invalid argument"
```

With 0 argument, the exception is thrown with the exception id set to "user-exception". With one argument, the argument is the exception id. With 2 arguments, the exception id and reason are set.

5.1.2 Exception handler

The reserved keyword **try** executes a form and catch an exception if one has been thrown. With one argument, the form is executed and the result is the result of the form execution unless an exception is caught. If an exception is caught, the result is the exception object. If the exception is a native one, the result is nil.

```
try (+ 1 2)                          → 3
try (throw)                          → nil
try (throw "hello")                  → nil
try (throw "hello" "world")          → nil
try (throw "hello" "world" "folks") → "folks"
```

In its second form, the **try** reserved keyword can accept a second form which is executed when an exception is caught. When an exception is caught, a new nameset is created and the special symbol **what** is bounded with the exception object. In such environment, the exception can be evaluated. The `what:eid` qualified name is the exception id. The `what:reason` qualified name is the exception reason and `what:object` is the exception object.

```
try (throw "hello")              (eval what:eid)    → "hello"
try (throw "hello" "world")      (eval what:reason) → "world"
try (throw "hello" "world" 2000) (eval what:object) → 2000
```

Exceptions are useful to notify abruptly that something went wrong. With an untyped language like Aleph, it is also a convenient mechanism to abort an expression call if some arguments do not match the expected types.

```
# protected factorial
const fact (n) {
  if (not (integer-p n)) (throw "number-error" "invalid argument in fact")
  if (== n 0) 1 (* n (fact (- n 1)))
}
(try (fact 5) 0)       → 120
(try (fact "hello") 0) → 0
```

5.2 Nameset

A nameset is created with the reserved keyword **nameset**. Without argument, the **nameset** reserved keyword creates a nameset without setting its parent. With one argument, a nameset is created and the parent set with the argument.

```
const nset       (nameset)
const nset       (nameset ...)
```

5.2.1 Default namesets

When a nameset is created, the symbol `.` is automatically created and bound to the newly created nameset. If a parent nameset exists, the symbol `..` is also automatically created. The use of the current nameset is a useful notation to resolve a particular name given a hierarchy of namesets.

```
trans a 1           → 1
block {
  trans   a (+ a 1) → 2
  println ..:a 1    → 1
}
println a           → 1
```

5.2.2 Nameset and inheritance

When a nameset is set as the super object of an instance, some interesting results are obtained. Because symbols are resolved in the nameset hierarchy, there is no limitation to use a nameset to simulate a kind of multiple inheritance. The following example illustrates this point.

```
const   cls (class)
const   ins (cls)
const   ins:super (nameset)
const   ins:super:value 2000
const   ins:super:hello "hello world "
println ins:hello ins:value → hello world 2000
```

5.3 Delayed Evaluation

Aleph provides a mechanism called *delayed evaluation*. Such mechanism permits the encapsulation of a form to be evaluated inside an object called a *promise*.

5.3.1 Creating a promise

The reserved keyword **delay** creates a *promise*. When the *promise* is created, the associated object is not evaluated. This means that the promise evaluates to itself.

```
const a (delay (+ 1 2))
promise-p a → true
```

The previous example creates a *promise* and store the argument form. The form is not yet evaluated. As a consequence, the symbol `a` evaluates to the *promise* object.

5.3.2 Forcing a promise

The reserved keyword **force** the evaluation of a *promise*. Once the *promise* has been forced, any further call will produce the same result. Note also that at this stage, the *promise* evaluates to the evaluated form.

```
trans   y 3
const   l ((lambda (x) (+ x y)) 1)
assert  4 (force l)
trans   y 0
assert  4 (force l)
```

5.4 Interpreter

The Aleph interpreter is by itself a special object with specialized methods which do not have equivalent using the standard aleph notation. The interpreter is always referred with the special symbol `interp`. The following table is a summary of the symbols and methods bound to the interpreter.

Symbol	Description
argv	command arguments vector
os-name	operating system name
os-type	operating system type
version	full aleph version
program-name	interpreter program name
major-version	aleph major version number
minor-version	aleph minor version number
patch-version	aleph patch version number
aleph-url	aleph official url name
load	load a file and execute it
clone	clone the interpreter
launch	launch a normal thread
daemon	launch a daemon thread
library	load and initialize a library
set-real-precision	set real number precision
get-real-precision	set real number precision

Table 5.1: Interpreter symbols

5.4.1 Arguments vector

The `interp:argv` qualified name evaluates to a vector of strings. Each argument is stored in the vector during the interpreter initialization.

```
zsh> aleph hello world
aleph> println (interp:argv:length) → 2
aleph> println (interp:argv:get 0)  → hello
```

5.4.2 Interpreter version and os

Several symbols can be used to track the interpreter version and the operating system. The full version is bound to the `interp:version` qualified name. The full version is composed of the `major`, `minor` and `patch` number. The operating system name is bound to the qualified name `interp:os-name`. The operating system type (like `unix`) is bound to `interp:os-type`.

```
println "major version number   : " interp:major-version
println "minor version number   : " interp:minor-version
println "patch version number   : " interp:patch-version
println "interpreter version    : " interp:version
println "operating system name  : " interp:os-name
println "operating system type  : " interp:os-type
println "aleph official url     : " interp:aleph-url
```

5.4.3 File loading

The `interp:load` method loads and execute a file. The interpreter interactive command session is suspended during the execution of the file. In case of error or if an exception is raised, the file execution is terminated. The process used to load a file is governed by the *file resolver*. Without extension, a compiled file is searched first and if not found a source file is searched.

5.4.4 Library loading

The `interp:library` method loads and initializes a library. The interpreter maintains a list of opened library. Multiple execution of this method for the same library does nothing. The method returns the library object.

```
interp:library "aleph-sys"
println "random number: " (aleph:sys:random)
```

Chapter 6

Threads Operations

This chapter covers the threads facilities builtin in the **Aleph** interpreter. The thread subsystem allows for the execution of concurrent forms with an automatic synchronization mechanism. Designing a good program with concurrent execution is a difficult task. It takes a while to get used with the various synchronization mechanisms which ensure a safe execution, that is no race condition or dead lock. Fortunately, Aleph provides some unique features that should ease such design.

6.1 Normal and Daemon threads

The interpreter supports two types of threads, called *normal* and *daemon* threads. A normal thread is started with the reserved keyword **launch**. A daemon thread is started with the reserved keyword **daemon**. The difference between a normal thread and a daemon thread is only in the termination of the interpreter. An aleph program is completed when all normal threads have terminated. This means that the master thread (i.e the first thread) is suspended until all normal threads have been executed. With daemon threads, the master thread terminates even if some daemon threads are still running.

6.1.1 Starting a normal thread

A normal thread is started with the reserved keyword **launch**. The form to execute in a thread is the argument. The simplest thread to execute is the *nil* thread.

```
launch (nil)
```

Even the *nil* thread does nothing in term of computation, it does a lot of things internally by turning on the shared objects sub-system.

6.1.2 Thread object and result

When a thread terminate, the thread object holds the result of the last executed form. The thread object is returned by the **launch** or **daemon** command. The `thread-p` predicates returns `true` if the object is a thread descriptor. The thread type can be check with the `normal-p` or `daemon-p` predicates.

```
const thr (launch (nil))
println   (thread-p thr) → true
println   (thr:normal-p) → true
```

The member data `result` of the thread object holds the result of the thread. Although the result can be accessed at any time, the returned value will be nil until the thread as completed its execution.

```
const thr (launch (nil))
println   (thr:result)  → nilp
```

Although the Aleph engine will ensure that the result is `nil` until the thread has completed its execution, it does not mean that it is a reliable approach to test until the result is not `nil`. The engine provides various mechanisms to synchronize a thread and eventually wait for its completion.

6.2 Shared Objects

The whole purpose of using a multi-threaded environment is to provide a concurrent execution with some shared variables. Although, several threads can execute concurrently without sharing data, the most common situation is that one or more global variable are accessed (and even changed) by one or more threads. Various scenarios are possible. For example, a variable is changed by one thread, the other thread just read its value. Another scenario is one read, multiple write, or even more complicated, multiple read and multiple write. In any case, the interpreter subsystem must ensure that each objects are in a good state when such operation do occur.

The Aleph engine provides an automatic synchronization mechanism for global objects, where only one thread can modify an object, but several thread can read it. This mechanism known as *read-write locking* guarantee that there is only one writer, but eventually multiple reader. When a thread start to modify an object, no other thread are allowed to read or write this object until the transaction has been completed. On the opposite, no thread is allowed to change (i.e. write) an object, until all thread which access (i.e. read) the object value have completed the transaction. Because a context switch can occur at any time, the object read-write locking will ensure a safe protection during each concurrent access.

6.2.1 Various shared objects

Shared objects can be very complicated to detect. For example, if a vector is shared by various threads, the engine will make sure that all vector objects are also shared. A closed variable in a lambda or gamma expression is another example of potential shared object. Executing such lambda form in a thread will automatically mark the closed variables as shared objects. Additionally, when the thread system is started, all object in the global nameset are marked shared.

6.2.2 Shared object predicate

The object predicate method `shared-p` returns true if an object is shared. Since all global objects are marked shared as soon as the thread system is turned on, the following example shows how a nil thread marks a shared variable.

```
# create simple symbol
const  a     1
```

```
assert false (a:shared-p)

# turn on the thread system
launch (nil)
assert true (a:shared-p)

# check another symbol
trans b 1
assert true (b:shared-p)
```

When an object is marked shared, it will remain in this state for rest of the session. Note that when an object is copied (by copy construction), the shared state is not copied. The copied object will become shared depending on its surrounding context. Such context can be a nameset or any other type of container which is shared or not.

6.2.3 Shared protection access

We illustrate the previous discussion with an interesting example and some variations around it. Let's consider a form which increase an integer object and another form which decrease the same integer object. If the integer is initialized to 0, and the two forms run in two separate threads, we might expect to see the value bounded by the time allocated for each thread. In other word, this simple example is a very good illustration of your machine scheduler.

```
# shared variable access
const var 0

const incr nil (while true (println "increase: " (var:= (+ var 1))))
const decr nil (while true (println "decrease: " (var:= (- var 1))))

# start both threads
launch (decr)
launch (incr)
```

In the previous example, `var` is initialized to 0. The `incr` thread increments `var` while the `decr` thread decrements `var`. Depending on the operating system, the result stays bounded within a certain range (generally -5000 to 5000). The previous example can be changed by using the main thread or a third thread to print the variable value. The end result is the same, except that there is more threads competing for the shared variable.

```
# shared variable access
const var 0

# incrementer, decrementer and printer
const incr nil (while true (var:= (+ var 1)))
const decr nil (while true (var:= (- var 1)))
const prtv nil (while true (println "value = " var)

# start all threads
launch (decr)
launch (incr)
launch (prtv)
```

6.3 Synchronization

Although, Aleph provides an automatic synchronization mechanism for reading or writing an object, it is sometimes necessary to control the execution flow. There are basically two techniques to do so. First, protect a form from being executed by several threads. Second, wait for one or several threads to complete their task before going to the next execution step.

6.3.1 Form synchronization

The reserved keyword **sync** can be used to synchronize a form. When a form, is synchronized, the Aleph engine guarantees that only one thread will execute this form.

```
const print-message (code mesg) (
  sync {
     errorln "error  : " code
     errorln "message: " mesg
  }
)
```

The previous example create a gamma expression which make sure that both the error code and error message are printed in one group, when several threads call it.

6.3.2 Thread completion

The other piece of synchronization is the thread completion indicator. The thread descriptor contains a method called `wait` which suspend the calling thread until the thread attached to the descriptor has been completed. If the thread is already completed, the method returns immediately.

```
# simple flag
const flag false

# simple shared tester
const ftest (val) (flag) (assert val (flag:shared-p))

# no thread mean not shared
ftest false

# in a thread it is shared
const thr (launch (ftest true))
thr:wait
assert true (flag:shared-p)
```

This example is taken from the test suites. It checks that a closed variable becomes shared when started in a thread. Note the use of the `wait` method to make sure the thread has completed before checking for the shared flag. It is also worth to note that `wait` is one of the method which guarantees that a thread result is valid.

Another use of the `wait` method can be made with a vector of thread descriptors when one wants to wait until all of them have completed.

```
# shared vector of threads descriptors
const thr-group (Vector)
```

```
# wait until all threads in the group are finished
const wait-all nil (for (thr) (thr-group) (thr:wait))
```

6.3.3 Complete example

We illustrate the previous discussion with a complete example. The idea is to perform a matrix multiplication. A thread is launched when when multiplying one line with one column. The result is stored in the thread descriptor. A vector of thread descriptor is used to store the result.

```
# initialize the shared library
interp:library "aleph-sys"

# shared vector of threads descriptors
const thr-group (Vector)

# this procedure waits until all threads in the group are finished
const wait-all nil (for (thr) (thr-group) (thr:wait))

# this procedure initialize a matrix with random numbers
# the matrix is a square one with its size as an argument
const init-matrix (n) {
  trans i (Integer 0)
  const m (Vector)
  do {
    trans v (m:append (Vector))
    trans j (Integer)
    do {
      v:append (aleph:sys:random)
    } (< (j:++) n)
  } (< (i:++) n)
  eval m
}

# this procedure multiply one line with one column
const mult-line-column (u v) {
  assert (u:length) (v:length)
  trans result 0
  for (x y) (u v) (result:+= (* x y))
  eval result
}

# this procedure multiply two vectors assuming one is a line and one is
# a column coming from the matrix
const mult-matrix (mx my) {
  for (lv) (mx) {
    assert true (vector-p lv)
    for (cv) (my) {
      assert true (vector-p cv)
      thr-group:append (launch (mult-line-column lv cv))
    }
```

```
  }
}

# check for some arguments
if (== 0 (interp:argv:length)) {
  errorln "usage: aleph 0607.als size"
  aleph:sys:exit 1
}

# get the integer and multiply
const n (Integer (interp:argv:get 0))
mult-matrix (init-matrix n) (init-matrix n)

# wait for all threads to complete
wait-all

# make sure we have the right number
assert (* n n) (thr-group:length)
```

6.3.4 Condition variable

A *condition variable* is another mechanism to synchronize several threads. A condition variable is modeled with the **Condvar** object. At construction, the condition variable is initialized to `false`. A thread calling the `wait` method will block until the condition becomes `true`. The `mark` method can be used by a thread to change the state of a condition variable and eventually awake some threads which are blocked on it. The following example shows how the main thread blocks until another change the state of the condition.

```
# create a condition variable
const cv (Condvar)

# this function runs in a thread - does some computation
# and mark the condition variable
const do-something nil {
  # do some computation
  ....
  # mark the condition
  cv:mark
}

# start some computation in a thread
launch (do-something)

# block until the condition is changed
cv:wait-unlock

# continue here
...
```

In this example, the condition variable is created at the beginning. The thread is started and the main thread blocks until the thread change the state of the condition variable. It is important to note the use of the `wait-unlock` method. When the main thread is re-started (after the condition variable has been marked), the main thread owns the lock associated with the condition variable. The `wait-unlock` method unlocks that lock when the main thread is restarted. Note also that the `wait-unlock` method reset the condition variable. if the `wait` method was used instead of `wait-unlock` the lock would still be owned by the main thread. Any attempt by other thread to call the mark method would result in the calling thread to block until the lock is released.

The **Condvar** class has several methods which can be used to control the behavior of the condition variable. Most of them are related to lock control. The `reset` method reset the condition variable. The `lock` and `unlock` control the condition variable locking. The `mark`, `wait` and `wait-unlock` method controls the synchronization among several threads.

Chapter 7

Regular Expressions

This chapter covers the **Aleph** regular expressions (*regex*) syntax and programming use. The **Aleph** *regex* is an original implementation with its own syntax and execution model.

7.1 Regular expression syntax

Aleph implements a regular expression engine via a special **Regex** object. A regular expression can be built implicitly or explicitly with the use of the `Regex` object. The *regex* syntax uses the '[' and ']' characters as block delimiters. When used in a source file, the lexical analyzer automatically recognizes a *regex* and built the object accordingly. In other word, the *regex* system is builtin in the **Aleph** language. The following example shows two equivalent way to define the same *regex* expression.

```
# syntax builtin regex
(== [$d+] 2000)          → true
# explicit builtin regex
(== (Regex "$d+") 2000) → true
```

In its first form, the '[' and ']' characters are used as syntax delimiters. The lexical analyzer automatically recognizes this token as a *regex* and built the equivalent `Regex` object. The second form is the explicit construction of the `Regex` object. Note also that the '[' and ']' characters are also used as *regex* block delimiters.

7.1.1 Regex characters and meta-characters

Any character, except the one used as operators can be used in a *regex*. The '$' character is used as a meta-character (or control character) to represent a particular set of characters. For example, `[hello world]` is a *regex* which match only the `"hello world"` string. The `[$d+]` *regex* matches one or more digits. The following meta characters are builtin in the *regex* engine.

- **$a** matches any letter or digit.
- **$b** matches any blank characters.
- **$d** matches any digit.

- **$l** matches any lower case letter.
- **$n** matches new line characters.
- **$s** matches any letter.
- **$u** matches any upper case letter.
- **$w** matches any aleph word constituent.
- **$x** matches any hexadecimal characters.

The uppercase version is the complement of the corresponding lowercase character set.

- **$A** matches any character except letter or digit.
- **$B** matches any character except blanks.
- **$D** matches any character except digit.
- **$L** matches any character except lower case letters.
- **$N** matches any character except new line.
- **$S** matches any character except letters.
- **$U** matches any character except upper case letters.
- **$W** matches any character except aleph word constituents.
- **$X** matches any character except hexadecimal characters.

A character which follows a $ character and that is not a meta character is treated as a normal character. For example `$[` is the '[' character. A quoted string can be used to define character matching which could otherwise be interpreted as control characters or operator. A quoted string also interprets standard *escaped* sequences but not meta characters.

```
(== [$d+]   2000) → true
(== ["$d+"] 2000) → false
```

7.1.2 Regex character set

A character set is defined with the '<' and '>' characters. Any enclosed character defines a character set. Note that meta characters are also interpreted inside a character set. For example, <`$d+-`> represents any digit or a plus or minus. If the first character is the ^ character in the character set, the character set is complemented with regards to its definition.

7.1.3 Regex blocks and operators

The '[' and ']' characters are the *regex* sub-expressions delimiters. When used at the top level of a *regex* definition, they can identify an implicit object. Their use at the top level for explicit construction is optional. The following example is strictly equivalent.

```
# simple real number check
const real-1 (Regex "$d*.$d+")
# another way with [] characters
const real-2 (Regex "[$d*.$d+]")
```

Sub-expressions can be nested (that's their role) and combined with operators. There is no limit in the nesting level.

```
# pair of digit testing
(== [$d$d[$d$d]+] 2000)  → true
(== [$d$d[$d$d]+] 20000) → false
```

The following unary operators can be used with single character, control characters and sub-expressions.

- ***** match zero or more times
- **+** match one or more times
- **?** match zero or one time.
- | alternation

Alternation is an operator which work with a secondary expression. Care should be taken when writing the right sub-expression. For example the following *regex* `[$d|hello]` is equivalent to `[[$d|h]ello]`. In other word, the minimal first sub-expression is used when compiling the *regex*.

7.1.4 Grouping

Groups of sub-expressions are created with the '(' and ')' characters. When a group is matched, the resulting sub-string is placed on a stack and can be used later. In this respect, the *regex* engine can be used to extract sub-strings. The following example extracts the month, day and year from a particular date format: `[($d$d):($d$d):($d$d$d$d)]`. This *regex* assumes a date in the form `mm:dd:yyyy`.

```
if (== (const re [($d$d):($d$d)]) "12:31") {
  trans hr (re:get 0)
  trans mn (re:get 1)
}
```

Grouping is the mechanism to retrieve sub-strings when a match is successful. If the *regex* is bind to a symbol, the `get` method can be used to get the sub-string by index.

7.2 Regex Object

Although a *regex* can be built implicitly, the `Regex` object can also be used to build a new *regex*. The argument is a string which is compiled during the object construction.

7.2.1 Literal object

A `Regex` object is a literal object. This means that the `to-string` method is available and that a call to the `println` special form will work directly.

```
const re (Regex "$d+")
println re           → $d+
println re:to-string → [$d+]
```

7.2.2 Regex operators

The == and != operators are the primary operators to perform a *regex* match. The == operator returns true if the `regex` matches the string argument from the beginning to the end of string. Such operator implies the begin and end of string anchoring. The < operator returns true if the *regex* matches the string or a substring or the string argument.

7.2.3 Regex methods

The primary *regex* method is the `get` method which returns by index the sub-string when a group has been matched. The `length` method returns the number of group match.

```
if (== (const re [($d$d):($d$d)]) "12:31") {
  re:length → 2
  re:get 0  → 12
  re:get 1  → 31
}
```

The `match` method returns the first string which is matched by the *regex*.

```
const regex [$d+]
regex:match "Happy new year 2001" → 2001
```

The `replace` method any occurrence of the matching string with the string argument.

```
const regex [$d+]
regex:replace "Hello year 2000" "2001" → hello year 2001
```

7.2.4 Argument conversion

The use of the `Regex` operators implies that the arguments are evaluated as literal object. For this reason, an implicit string conversion is made during such operator call. For example, passing the integer `12` or the string `"12"` is strictly equivalent. Care should be taken when using this implicit conversion with real numbers.

Chapter 8

Functional Programming

This chapter covers the interesting aspects of **Aleph** with respect to the *functional programming paradigm*. Functional programming is often described as the ability to *create functions that creates functions*. As a matter of fact, it is a far bigger subject that finds its root in the *Lambda Calculus*. A language (like **Aleph**) that supports the functional programming paradigm is also sometimes called a *high order language*.

8.1 Function expression

A *lambda expression* or a *gamma expression* can be seen like a function object with no name. During the evaluation process, the expression object is evaluated as well as the arguments (from left to right) and a result is produced by applying those arguments to the function object. An expression can be built dynamically as part of the evaluation process.

```
aleph >println ((lambda (n) (+n 1)) 1)
2
```

The difference between a *lambda expression* and a *gamma expression* is only in the nameset binding during the evaluation process. The *lambda expression* nameset is linked with the calling one, while the *gamma expression* nameset is linked with the top level nameset. The use of *gamma expression* is particularly interesting with recursive functions as it can generate a significant execution speedup. The previous example will behaves the same with a gamma expression.

```
aleph >println ((gamma (n) (+n 1)) 1)
2
```

8.1.1 Self reference

When combining a function expression with recursion, the need for the function to call itself is becoming a problem since that function expression does not have a name. For this reason, **Aleph** provides the reserved keyword **self** that is a reference to the function expression. We illustrate this capability with the well-known factorial expression written in pure functional style.

```
aleph >println ((gamma (n) (if (<= n 1) 1 (* n (self (- n 1))))) 5)
120
```

The use of a *gamma expression* versus a *lambda expression* is a matter of speed. Since the *gamma expression* does not have *free variable*, the symbol resolution is not a concern here.

8.1.2 Closed variables

One of the **Aleph** characteristic is the treatment of *free variables*. A variable is said to be free if it is not bound in the expression environment or its children at the time of the symbol resolution. For example, the expression `((lambda (n) (+ n x)) 1)` computes the sum of the argument `n` with the free variable `x`. The evaluation will succeeds if x is defined in one of the parent environment. Actually this example can also illustrates the difference between a *lambda expression* and a *gamma expression*. Let's consider the following forms.

```
trans x 1

const do-print nil {
  trans x 2
  println ((lambda (n) (+ n x)) 1)
}
```

The function do-print (which is a *gamma expression* because of the **const** reserved keyword) will produce `3` since it sums the argument `n` bound to `1`, with the free variable `x` which is defined in the calling environment as `2`. Now if we rewrite the previous example with a *gamma expression* the result will be one, since the expression parent will be the top level environment that defines `x` as 1.

```
trans x 1

const do-print nil {
  trans x 2
  println ((gamma (n) (+ n x)) 1)
}
```

With this example, it is easy to see that there is a need to be able to determine a particular symbol value during the expression construction. Doing so is called *closing a variable*. Closing a variable is a mechanism that binds into the expression a particular symbol with a value and such symbol is called a *closed variable*, since its value is closed under the current environment evaluation. For example, the previous example can be rewritten to close the symbol `x`.

```
trans x 1

const do-print nil {
  trans x 2
  println ((gamma (n) (x) (+ n x)) 1)
}
```

Note that the list of closed variable immediately follow the argument list. In this particular case, the function `do-print` will print `3` since `x` has been closed with the value `2` has defined in the function `do-print`.

8.1.3 Dynamic binding

Because **Aleph** has a dynamic binding symbol resolution, it is possible to have under some circumstances a free or closed variable. This kind of situation can happen when a particular symbol is defined under a condition.

```
lambda (n) {
  if (<= n 1) (trans x 1)
  println (+ n x)
}
```

With this example, the symbol x is a free variable if the argument `n` is greater than 1. While this mechanism can be powerful, extreme caution should be made when using such feature. Note also that many other language do not allow this kind of behavior. That kind of restriction is primarily driven by the need to have a language with *static binding*. The bad news is that it is impossible to write a compiler with dynamic symbol binding.

8.2 Functional objects

Everything in **Aleph** is an object. As a consequence, an object can be manipulated, even if it is lexical element, a symbol or a closure.

8.2.1 Lexical and qualified names

The basic forms elements are the lexical and qualified names. Lexical and qualified names are constructed by the **Aleph** reader. Although the evaluation process make that lexical object transparent, it is possible to manipulate them directly.

```
aleph >const sym (protect lex)
aleph >println   (sym:repr)
Lexical
```

In this example, the **protect** reserved keyword is used to avoid the evaluation of the lexical object named `lex`. Therefore the symbol `sym` refers to a lexical object. Since a lexical (and a qualified) object is a also a literal object, the `println` reserved function will work and print the object name. In fact, a literal object provides the `to-string` method that returns the string representation of a literal object.

```
aleph >const sym (protect lex)
aleph >println   (sym:to-string)
lex
```

8.2.2 Symbol and argument access

Each nameset maintains a table of symbols. A symbol is a binding between a name and an object. Eventually, the symbol carries the `const` flag. During the lexical evaluation process, the lexical object tries to find an object in the nameset hierarchy. Such object can be either a symbol or an argument. Again, this process is transparent, but can be controlled manually. Both lexical and qualified named object have the `map` method that returns the first object associated in the nameset hierarchy.

```
aleph >const obj 0
aleph >const lex (protect obj)
aleph >const sym (lex:map)
aleph >println   (sym:repr)
Symbol
```

A symbol is also a literal object, so the `to-string` and `to-literal` methods will return the symbol name. Symbol methods are provided to access or modify the symbol values. It is also possible to change the `const` symbol flag with the `set-const` method.

```
aleph >println (sym:get-const)
true
aleph >println (sym:get-object)
0
aleph> sym:set-object true
aleph >println (sym:get-object)
true
```

A symbol name cannot be modified, since the name must be synchronized with the nameset association. On the other hand, a symbol can be explicitly constructed. As any object, the = operator can be used to assign a symbol value. The operator will behaves like the `set-object` method.

```
aleph >const sym (Symbol "symbol")
aleph >println sym
symbol
aleph >sym:= 0
aleph >println (eval sym)
0
```

8.2.3 Closure

As an object, the `Closure` can be manipulated outside the traditional declarative way. A closure is a special object that holds an argument list, a set of closed variables and a form to execute. The mechanic of a closure evaluation has been described earlier. What we are interested here is the ability to manipulate a closure as an object and eventually modify it. Note that by default a closure is constructed as a lambda expression. With a boolean argument set to true the same result is obtained. With false, a gamma expression is created.

```
aleph >const f (Closure)
aleph >println (closure-p f)
true
```

This example creates an empty closure. The default closure is equivalent to the `trans f nil nil`. The same can be obtained with `(const f (Closure true))`. For a gamma expression, the following forms are equivalent, `const f (Closure false)` and `const f nil nil`. Remember that it is `trans` and `const` that differentiate between a lambda and a gamma expression. Once the closure object is defined, the `set-form` method can be used to bind a form.

```
# the simple way
trans f nil (println "hello world")
# the complex way
const f    (Closure)
f:set-form (protect (println "hello world"))
```

There are numerous situations where it is desirable to mute dynamically a closure expression. The simplest one is the closure that mute itself based on some context. With the use of `self`, a new form can be set to the one that is executed. Another use is a mechanism call *advice*, where some new computation are inserted prior the closure execution. Note that appending to a closure can lead to some strange results if the existing closure expression uses `return` special forms. In a multi-threaded environment, the ability to change a closure expression is particularly handy. For example a special thread could be used to monitor some context. When a particular situation develops, that threads might trigger some closure expression changes. Note that changing a closure expression does not affect the one that is executed. If such change occurs during a recursive call, that change is seen only at the next call.

8.3 Combinators example

The remaining part of this chapter is an example of functional programming based on combinators abstraction. A combinator (in the **Aleph** terminology) is a single argument closure without *free variables*. At this stage, there is no difference between a lambda expression and a gamma expression. The difference shows up only in the execution context. The simplest combinator is `const I (x) (eval x)`, that is the *identity* combinator.

8.3.1 Curried expression

A multi-argument closure can be converted to a single argument closure by encapsulating the arguments into other closure. For example, the function $f(x, y) = x + y$ can be computed, either with `const f (x y) (+ x y)` or by writing `const g (x) (gamma (y) (x) (+ x y))`. In the first form, the expression is called with the arguments, while the second form requires two calls. No matter how the call is made, the result, remains the same.

```
# direct call
aleph >println (f 1 2)
3
# curried call
aleph >println ((g 1) 2)
3
```

Clearly, this mechanism can be extended to several arguments. This technique, called *currying*, is named after Haskell B. Curry, but was first introduced by Moses Schonfinkel. With **Aleph**, the form evaluation is a two step process (eval, apply) than runs from left to right. Each arguments are first evaluated, placed on the eval stack and the function is applied. With the curry approach, the evaluation is sequential. If the arguments are evaluated to *normal* objects (that is objects that do not have side effects), the result should be the same, but with *complex* object (for example a file descriptor), the result might be different. Nevertheless, we will assume that a regular **aleph** form $(fxyz)$ can be curried to produce the same result by writing $(((fx)y)z)$.

8.3.2 Base combinators

If we are given a function *f*, can we express the same function in the form of nested combinators $C_1C_2...C_n$? The answer to that question is actually quite complex and is one addressed by the *computability theory*. Since we don't want to do the math here, let's rather focus on some interesting examples to illustrate our point. The *identity* combinator has been show previously. There are two other interesting combinators called K and S. K is the cancellation combinator, which drops its second argument and return its first one. An evaluation like (Kxy) is expected to evaluate x and return it. The curried form will be $((Kx)y)$ which does the same. S is the distribution combinator, which distributes an argument to two functions. An evaluation like $(Sfgx)$, which is curried like $(((Sf)g)x)$ will be equivalent to $((fx)(gx))$. With **Aleph**, both K and S combinators can be defined as follow.

```
const K (x) (gamma (y) (x) (eval x))
const S (f) (gamma (g) (f) (gamma (x) (f g) ((f x) (g x))))
```

It is amazing to note how these two combinators are powerful. For example, the `SKK` combinator sequence is the *identity* combinator. This can be verified with example `0803.als`. In other words, we have $SKK = I$. Because the *identity* combinator is convenient, we will keep it 'as is'.

8.3.3 Form transformation

Forms can be converted to a combinatoric representation. That transformation process is rather simple and use only the `SKI` combinators. To convince ourself, let's take a simple example like $f(x) = x + x$. The form to convert is simply $(+xx)$

- *step 1* terms currying
  (+ x x) → ((+ x) x)
- *step 2* S term mapping
  ((+ x) x) → (S [(+ x)]) [x]
- *step 3* I term mapping
  (S [(+ x)]) [x] → (S [(+ x)]) I
- *step 4* S term mapping
  (S [(+ x)]) I → (S ((S [+]) [x])) I
- *step 5* I term mapping
  (S ((S [+]) [x])) I → (S ((S [+]) I)) I
- *step 6* K term mapping
  (S ((S [+]) I)) I → (S ((S (K +)) I)) I

Note that the step 1 transform the form into a combinatoric representation. At this stage, the notation for the '+' operator is also a transformation from the builtin operator to a curried version of it. If call `c+` the curried '+' operator, we have the final implementation.

```
# the S combinator
const S (f) (gamma (g) (f) (gamma (x) (f g) ((f x) (g x))))

# the K combinator
const K (x) (gamma (y) (x) (eval x))

# the I combinator
const I (x) (eval x)

# curried '+' operator
const c+ (x) (gamma (y) (x) (+ x y))

# testing the reduction (+ x x) => (S ((S (K +)) I)) I)
println "((+ x x) 512) = " (((S ((S (K c+)) I)) I) 512)
```

The good news about all of this is that the transformation process is rather mechanical. Once a form has been transformed, it can be represented by a graph which is subject to optimization. Such optimization is called a *combinator graph reduction*. What it means is that we have here a mechanism to optimize a form. Moreover, we have also a simple mechanism to perform a form compilation that could be interpreted with a virtual combinatorial machine. Such compilation process is the subject of a later discussion... In summary, given a lambda or gamma expression with zero or more argument, the combinatoric transformation involves several steps.

- *Step 1* Form currying
  The form is transformed recursively to produce an expression with one argument. The operators are curried. The special forms are subject to a special treatment.
- *Step 2* SKI transformation
  The curried expression is transformed into a combinatoric representation with the SKI combinators.

8.3.4 Recursive combinator

Combinators offer an elegant way to address recursive form. The idea is to create a combinator that can restart itself while evaluating some arguments. We illustrate this point with the *de facto* factorial function.

```
# factorial - the old fashion way
const fact (n) (if (== n 1) 1 (* n (fact (- n 1))))
```

The transformation creates a combinatoric representation of the factorial with a closed variable which is the form to restart.

```
# factorial - as a combinator
const c-fact (f) (gamma (n) (f) (if (== n 1) 1 (* n ((f f) (- n 1)))))
```

The `c-fact` gamma expression evaluates to a gamma expression. The gamma expression uses the closed variable `f` to restart itself. This is the recursion in its combinatoric form. Note that we need another combinator to start the first evaluation. Example `0805.als` demonstrates this example.

```
# the U combinator
const U (f) (f f)
# the generated factorial
const fact (U c-fact)
```

However, we have used a trick here when defining the `c-fact` gamma expression since the expression restart itself. What we would like to have is rather:

```
# factorial - non restarting
const c-fact (f) (gamma (n) (f) (if (== n 1) 1 (* n (f (- n 1)))))
```

What we need now is combinator that tales care of restarting the factorial. This combinator is known as the *Y combinator* and is defined as follow:

```
# the Y combinator
const Y (f) ((gamma (g) (f) (f (gamma (x) (g) ((g g) x))))
             (gamma (g) (f) (f (gamma (x) (g) ((g g) x)))))
# the generated factorial
const fact (Y c-fact)
```

Example *0806.als* demonstrates what has just been described here. Note that the Y combinator is complex because it needs , first to restart the function (like the U combinator does), but also needs to proceed with the evaluation of the function itself. Adding some information statement is quite revealing.

```
# factorial - as a combinator
const c-fact (f) {
  println "creating factorial gamma expression"
  gamma (n) (f) {
    println "evaluating the factorial gamma with n = " n
    if (== n 1) 1 (* n (f (- n 1)))
  }
}

zsh >aleph 0806.als
creating factorial gamma expression
evaluating the factorial gamma with n = 5
creating factorial gamma expression
evaluating the factorial gamma with n = 4
creating factorial gamma expression
evaluating the factorial gamma with n = 3
creating factorial gamma expression
evaluating the factorial gamma with n = 2
creating factorial gamma expression
evaluating the factorial gamma with n = 1
fact 5 = 120
```

The Y combinator has also the interesting property to act as a fixed point combinator. It is amazing to note that `(c-fact (Y c-fact))` is almost equivalent to `(Y c-fact)` (up to a closure). That is the property $YF = F(YF)$ holds and YF is a fixed point. Such property is the root of the recursion. In fact, any combinator like `c-fact` can be transformed into a recursive function with the help of the Y combinator. To convince yourself, look at example `0807.als` which computes a Fibonacci value with the Y combinator. It is also clear that the Y combinator is an elegant way to perform recursion *without name* like the reserved keyword **self** does. Note also that a fixed point combinator can be characterized (up to a closure) with the SKI combinators by $Y = ((SI)Y)$.

8.3.5 Other combinators

There are other combinators. All of them involve some sort of computation analog to the SKI combinators. For example the *B combinator* is the composition combinator $((((Bf)g)x) = f(g(x))$. The *C combinator* is an argument swap combinator $(((Cf)x)y) = ((fy)x)$. The *W combinator* is the argument doubling combinator $((Wf)x) = ((fx)x)$. It can be shown that the base {S K} is the smallest combinator base. However, other base can be used to do the same job. For example {I B C W K} is another base. The rule of game is to find the base that please you.

Chapter 9

Librarian and Resolver

This chapter covers the use of the **axl** librarian utility as well as the **Librarian** object. The file path resolver is also described as a mean to search for a particular file to execute in a program.

9.1 Librarian

A librarian file is a special file that acts as a containers for various files. A librarian file is created with the **axl** *Aleph Cross Librarian* utility. Once a librarian file is created, it can be added to the interpreter resolver. The file access is later performed automatically by name with the standard interpreter `load` method.

9.1.1 Creating a librarian

The **axl** utility is the preferred way to create a librarian. Given a set of files, **axl** combines them into a single one.

```
zsh > axl -h
usage: axl [options] [files]
           [-h]      print this help message
           [-v]      print version information
           [-c]      create a new librarian
           [-x]      extract from the librarian
           [-s]      get file names from the librarian
           [-t]      report librarian contents
           [-f] lib  set the librarian file name
```

The `-c` option creates a new librarian. The librarian file name is specified with the `-f` option.

```
zsh > axl -c -f librarian.axl file-1.als file-2.als
```

The previous command combines `file-1.als` and `file-2.als` into a single file called `librarian.axl`. Note that any file can be included in a librarian.

9.1.2 Using the librarian

Once a librarian is created, the interpreter `-i` option can be used to specify it. The `-i` option accepts either a directory name or a librarian file. Once the librarian has been opened, the interpreter `load` method can be used as usual.

```
zsh  > aleph -i librarian.axl
aleph> interp:load "file-1.als"
aleph> interp:load "file-2.als"
```

The librarian acts like a file *archive*. The interpreter file resolver takes care to extract the file from the librarian when the `load` method is invoked.

9.1.3 Librarian contents

The **axl** utility provides the `-t` and `-s` options to look at the librarian contents. The `-s` option returns all file name in the librarian. The `-t` option returns a one line description for each file in the librarian.

```
zsh > axl -t -f librarian.axl
--------       1234 file-1.als
--------       5678 file-2.als
```

The one line report contains the file flags `--------`, the file size `1234` and the file name `file-1.als`. The file flags are not used at this time. One possible use in the future is for example, an *auto-load* bit or any other useful things.

9.1.4 Librarian extraction

The `-x` option permits to extract file from the librarian. Without any file argument, all files are extracted. With some file arguments, only those specified files are extracted.

```
zsh > axl -x -f librarian.axl
zsh > axl -x -f librarian.axl file-1.als
```

9.2 Librarian object

The **Librarian** object can be used within an **Aleph** program as a convenient way to create a collection of files or to extract some of them.

9.2.1 Output librarian

The **Librarian** object is a standard **Aleph** object. Its predicate is `librarian-p`. Without argument, a librarian is created in *output mode*. With a string argument, the librarian is opened in *input mode*, with the file name argument. The *output mode* is used to create a new librarian by adding file into it. The *input mode* is created to read file from the librarian.

```
# create a new librarian
const lbr (Librarian)
# add a file into it
lbr:add "file-1.als"
# write it
lbr:write "librarian.axl"
```

The `add` method adds a new file into the librarian. The `write` method the full librarian as a single file those name is `write` method argument.

9.2.2 Input librarian

With an argument, the librarian object is created in input mode. Once created, file can be read or extracted. The `length` method (which also work with an output librarian) returns the number of files in the librarian. The `Exists-p` predicate returns true if the file name argument exists in the librarian.The `get-names` method returns a vector of file names in this librarian. The `extract` method returns an *input stream* object for the specific file name.

```
# open a librarian for reading
const lbr (Librarian "librarian.axl")
# get the number of files
println (lbr:length)
# extract the first file
const is (lbr:extract "file-1.als")
# is is an input stream - dump each line
while (is:valid-p) (println (is:readln))
```

Most of the time, the librarian object is used to extract file dynamically. Because a librarian is *mapped* into the memory at the right offset, there is no worry to use big librarian, even for a small file. Note that any type of file can be used, text or binaries.

9.3 Resolver

The **Aleph** resolver is a special object used by the interpreter to resolve file path based on the search path. The resolver uses a mixed list of directories and librarian files in its search path. When a file path needs to be resolved, the search path is scanned until a matched is found. Because the librarian resolution is integrated inside the resolver, there is no need to worry about file extraction. That process is done automatically. The resolver can also be used inside an **Aleph** program to perform any kind of file path resolution.

9.3.1 Resolver object

The resolver object is created without argument. The `add` method adds a directory path or a librarian file to the resolver. The `valid` method checks for the existence of a file. The `lookup` method returns an input stream object associated with the object.

```
# create a new resolver
const rslv (Resolver)
assert true (resolver-p rslv)

# add the local directory on the search path
rslv:add "."

# check if file test.als exists
# if this is ok - print its  contents
if (rslv:valid-p "test.als") {
  const is (rslv:lookup "test.als")
  while (is:valid-p) (println (is:readln))
```

```
}
```

Appendix A

Reserved keywords

This appendix contains a summary of the Aleph reserved keywords with their syntax.

assert [reserved]

Description

The **assert** reserved keyword check for equality between two operands. Both objects must be of the same type. If the equality test fails, the reserved keyword print a message and abort the execution. By default, the assertion checking is turned off. The interpreter option `-f assert` enables the assertion checking. When the interpreter is compiled in debug mode, the assertion checking is turned on by default.

Syntax

```
assert form-1 form-2
```

Example

```
assert true (== 1 1)
assert 3    (+  2 1)
```

block [reserved]

Description

The **block** reserved keyword defines a new nameset for sequential execution of regular form or implicit form. When the block form is evaluated, the block nameset is linked to its parent nameset. When all forms have been executed, the block nameset is destroyed and he result of the last evaluation in the block is considered to be the result of the block evaluation.

Syntax

```
block regular form
block block form
```

Example

```
trans a 1
block {
  assert    a 1
  trans     a (+ 1 1)
  assert    a 2
  assert ..:a 1
}
assert 1 a
```

class [reserved]

Description

The **class** reserved keyword creates a new class object. Without argument, an instance of that class is created without data members. With a list of arguments, the instance is created with a set of data member initialized to nil.

Syntax

```
class
class data member-list
```

Example

```
const Color (class)
trans Color:initialize (red green blue) {
  const this:red   red
  const this:green green
  const this:blue  blue
}

const red   (Color 255   0   0)
const green (Color   0 255   0)
const blue  (Color   0   0 255)
```

const [reserved]

Description

The **const** reserved keyword binds a symbol with an object and marks it as a constant symbol. When used with three or four argument, a gamma expression is automatically created. **const** can also be used to bind class or instance members.

Syntax

```
const symbol object
const symbol argument body
const symbol argument closed variables body
```

Example

```
const number 123
const max (x y) (if (> x y) x y)
```

daemon [reserved]

Description

The **daemon** reserved keyword creates a new `thread` by executing the form argument in a daemon thread. The created thread is executed by creating a clone of the interpreter and starting immediately the execution of the form with the cloned interpreter. The command returns the thread object in the calling thread. When the thread terminates, the thread object holds the result of the last executed form. The main thread does not wait for a daemon thread to terminate.

Syntax

```
daemon form
```

Example

```
daemon (println "hello world")
```

delay [reserved]

Description

The **delay** reserved keyword delays the evaluation of the form argument by creating a `Promise` object. The promise evaluate to itself until a call to force the evaluation has been made. When the promise has been forced, the evaluation result is stored. Further call to force will produce the same result.

Syntax

```
delay form
```

Example

```
trans  y 3
const  l ((lambda (x) (+ x y)) 1)
assert 4 (force l)
trans  y 0
assert 4 (force l)
```

do [reserved]

Description

The **do** reserved keyword is used to build loop with forward condition. The first argument is the loop body and the second argument is the loop condition which must evaluates to a boolean object.

Syntax

```
do body condition
```

Example

```
const number-of-digits (s) {
    const len (s:length)
    trans index 0
    trans count 0
    do {
      trans c (s:get index)
      if (c:digit-p) (count:++)
    } (< (index:++) len)
    eval count
}
```

errorln [reserved]

Description

The **errorln** reserved keyword prints on the interpreter error stream a set of arguments. Each arguments have to be a literal which are converted to a string. When all arguments have been printed a new line character is printed. The **error** reserved keyword behaves like **errorln** excepts that a new line character is not printed at the end of the arguments.

Syntax

```
errorln
errorln nil
errorln literal-argument-list
```

Example

```
errorln
errorln "hello milenium" ' ' 2000
```

eval [reserved]

Description

The **eval** reserved keyword simply evaluates the object argument. The form is useful when returning an argument from a lambda or gamma expression using an implicit form.

Syntax

```
eval object
```

Example

```
const ret (x) (eval x)
eval (protect (+ 1 2))
```

for [reserved]

Description

The **for** reserved keyword provides a facility to iterate on *iterable* objects. `Cons`, `List` and `Vector` are typical iterable objects. For each iterable objects, a symbol is set after each iteration. Each object symbol value can be used for further computation. The iteration stops when one of the objects iterator is at the end position.

Syntax

```
for symbol-list iterable-object-list body
```

Example

```
# compute the scalar product of two vectors
const scalar-product (u v) {
  trans result 0
  for (x y) (u v) (result:+= (* x y))
  eval result
}
```

force [reserved]

Description

The **force** reserved keyword forces the evaluation of its argument. If the argument evaluates to a `promise` object, the promise evaluation is forced. If the argument is not a promise, **force** behaves like **eval**. When a promise has been forced, further call to force will not change the evaluation result.

Syntax

```
force object
```

Example

```
trans  y 3
const  l ((lambda (x) (+ x y)) 1)
assert 4 (force l)
trans  y 0
assert 4 (force l)
```

if [reserved]

Description

The **if** reserved keyword executes a form based on the evaluation of a boolean expression. In its first representation, **if** executes a form if the condition is evaluated to `true`. An alternate form can be specified and is executed if the boolean expression evaluates to `false`. It is an error to use a conditional form which does not evaluate to a boolean object.

Syntax

```
if cond true-form
if cond true-form else-form
```

Example

```
const max (x y) (if (> x y) x y)
```

lambda [reserved]

Description

The **lambda** reserved keyword creates a new `closure` object with eventually a set of arguments and a set of closed variables. In its first form, the closure is declared with a set of arguments or `nil` to indicate no argument. In its second form, the closure is declared with a set of arguments and a set of closed variables. The closed variables are evaluated at the construction of the closure and become part of the closure object. When the closure is called, a new nameset is created and linked with the parent nameset. The set of calling arguments are bounded in that nameset with the formal argument list to become the actual arguments. The set of closed variables is linked at runtime to the closure nameset. A lambda or gamma expression can have its argument declared as *const* argument.

Syntax

```
lambda nil body
lambda argument-list body
lambda argument-list closed-variables-list body
```

Example

```
const no-args (lambda nil (+ 1 1))
const add     (lambda ((const x) (const y)) (+ x y))
const closed  (lambda (x) (y) (+ x y))
```

launch [reserved]

Description

The **launch** reserved keyword creates a new `thread` by executing the form argument in a normal thread. The created thread is added in the normal thread list by creating a clone of the interpreter and starting immediately the execution of the form with the cloned interpreter. The command returns the thread object in the calling thread. When the thread terminates, the thread object holds the result of the last executed form. The main thread is suspended until all normal threads have completed their execution.

Syntax

```
launch form
```

Example

```
launch (println "hello world")
```

loop [reserved]

Description

The **loop** reserved keyword executes a loop based on an initial condition, an exit condition and a step form. The initial condition is only executed one time. The exit condition is tested at each loop iteration. The `loop` reserved keyword creates its own nameset since the initial condition generally binds symbol locally for the loop.

Syntax

```
loop init-form exit-form step form
```

Example

```
loop (trans i 0) (< i 10) (i:++) (println i)
```

nameset [reserved]

Description

The **nameset** reserved keyword creates a new nameset. With no argument, a new nameset is created and no parent is binded to this nameset. With one argument, the argument must evaluate to a nameset and that nameset is used as the parent one. If a nameset has to be created with the global nameset as the parent, the symbol `...` can be used to reference the top level nameset. The symbol `.` references the current nameset. The symbol `..` references the parent nameset of the current nameset.

Syntax

```
nameset
nameset parent-nameset
```

Example

```
const local-nameset-not-bound (nameset)
const local-nameset-bounded   (nameset ...)
const ...:global-nameset      (nameset)
```

println [reserved]

Description

The **println** reserved keyword prints on the interpreter output stream a set of arguments. Each arguments have to be a literal which is converted to a string. When all arguments have been printed a new line character is printed. The **print** reserved keyword behaves like **println** excepts that a new line character is not printed at the end of the arguments.

Syntax

```
println
println nil
println literal-argument-list
```

Example

```
println
println "hello milenium" ' ' 2000
```

protect [reserved]

Description

The **protect** reserved keyword take a single argument and returns it without evaluation. Protect is mainly use to get a symbol or form object.

Syntax

```
protect object
```

Example

```
const cons (protect (+ 1 2))
```

return [reserved]

Description

The **return** reserved keyword causes the current expression to stop its evaluation and returns the argument or nil. `return` is primarily used in lambda or gamma expressions. If used in a top level block, the block execution is stopped and the control is transfered to the top level.

Syntax

```
return object
```

Example

```
return (+ 1 2)
```

sync [reserved]

Description

The **sync** reserved keyword is a form synchronizer. Within a multi-threaded environment, the Aleph engine guarantees that only one thread will execute the form. The other threads are suspended until the form has been completed.

Syntax

```
sync form
```

Example

```
const print-message (code mesg) (
  sync {
     errorln "error  : " code
     errorln "message: " mesg
  }
)
```

switch [reserved]

Description

The **switch** reserved keyword is a form selector. The first argument is the object to switch. The second argument is a list of forms with an object matcher and an execution form. The `else` reserved keyword can be used as default matcher.

Syntax

```
switch selector list-of-condition
```

Example

```
const get-primary-color (color value) (
  switch color (
    ("red"   (return (value:substr 0 2))
    ("green" (return (value:substr 2 4))
    ("blue"  (return (value:substr 4 6))
  )
)
```

throw [reserved]

Description

The **throw** reserved keyword throws an exception. Without argument, an exception of type *user-exception* is thrown. with one argument, the *exception id* is set. With two arguments, the *exception id* and *exception reason* are set. With three arguments, *exception id*, *exception reason* and the *exception object* are set.

Syntax

```
throw
throw exception id
throw exception id exception reason
throw exception id exception reason exception object
```

Example

```
throw
throw "type-error"
throw "type-error" "invalid argument"
```

trans [reserved]

Description

The **trans** reserved keyword creates or sets a symbol with an object. **trans** searches in the current nameset only. If a symbol is found, it is set with the object. If the symbol is not found, it is created in the current nameset. **trans** can also be used with *qualified* names. With 3 or 4 arguments, **trans** creates automatically a lambda expression.

Syntax

```
trans symbol object
trans symbol argument body
trans symbol argument closed variables body
```

Example

```
trans a 1
trans fact (n) (if (< n 1) 1 (* n (fact (- n 1))))
```

try [reserved]

Description

The **try** reserved keyword catch an exception in the current execution nameset. The first argument is a form to execute. The optional second argument is the *exception handler* to be called in case of exception. If there is no exception handler, all exceptions are caught. The result of execution is either the result of the form execution, or the exception object in case of exception, or nil if the exception is a native one. If there is an exception handler, the handler is executed with a new nameset and the special symbol `what` is bound to the exception. If the exception is nil, the symbol *what* is undefined.

Syntax

```
try form
try form exception-handler
```

Example

```
try (+ 1 2)                              → 3
try (throw)                              → nil
try (throw "hello")                      → nil
try (throw "hello" "world")              → nil
try (throw "hello" "world" "folks")      → "folks"
```

while [reserved]

Description

The **while** reserved keyword is used to build loop with forward condition. The first argument is the loop condition and the second argument is the loop body.

Syntax

```
while cond body
```

Example

```
const gcd (u v) {
    while (!= v 0) {
      trans r (u:mod v)
      u:= v
      v:= r
    }
    eval u
}
```

Appendix B

Literal Objects

This chapter is a reference of the Aleph reserved objects with their respective builtin methods. The Aleph reserved objects are those objects defined in the global interpreter nameset and bind as reserved names.

Object	Description
Real	double floating point number
Regex	regular expression object
String	string reserved object
Boolean	boolean reserved object
Integer	64 bits signed integer
Relatif	infinite precision signed integer
Character	8 bits iso-8859-1 character

Table B.1: Aleph reserved objects

For each reserved object, Aleph provides a *predicate* which can be used to test for the object type. The base type for each reserved object is the **Literal** type. The predicate `literal-p` always returns `true` for these objects. The table below is a resume of the reserved predicates.

Object	Predicate
Real	real-p
Regex	regex-p
String	string-p
Boolean	boolean-p
Integer	integer-p
Relatif	relatif-p
Literal	literal-p
Character	character-p

Table B.2: Aleph reserved object predicates

All literal have a string representation. The `to-string` method is always available for these reserved objects. A literal object has a default constructor. Generally, it can also be constructed by a same type object or by a string object.

Literal [reserved]

Description

The **Literal** object is a base object for all literal object. The sole purpose of a literal object is to provide to methods named `to-string` and `to-literal` that return a string representation of the literal object.

Derivation summary

Derived from	Description
Serial	the base serial object

Methods Summary

Method	Description
to-string	returns a string representation
to-literal	returns a literal representation

Literal:to-string

- return: `String`
- arguments: `none`

The **`to-string`** method returns a string representation of the literal. The string is expected to represent at best the literal.

Literal:to-literal

- return: `String`
- arguments: `none`

The **`to-literal`** method returns a string representation of the literal. The string differs from the `to-string` method in the sense that the string is a literal representation. For example the literal representation of a string is the *quoted* string.

Boolean [reserved]

Description

The **Boolean** reserved object implements the behavior of a native boolean type. Two builtin symbols, namely **true** and **false** are used to represent the value of a boolean instance. The Boolean type is primarily used for test expression.

Derivation summary

Derived from	Description
Literal	the base literal object

Constructors Summary

Constructor	Description
Boolean	default boolean to false
Boolean *Boolean*	boolean from boolean value
Boolean *String*	boolean from string value

Operators Summary

Operator	Description
==	return true if both boolean and equal
!=	return true if both boolean and not equal

Integer [reserved]

Description

The **Integer** reserved object implements the behavior of a native 64 bits signed integer type. Standard decimal notation is used to construct integer object from a literal. The integer object can also be constructed from a string. Standard operators are provided for this class.

Derivation summary

Derived from	Description
Literal	the base literal object

Constructors Summary

Constructor	Description
Integer	default integer to 0
Integer *Real*	integer from real value
Integer *String*	integer from string value
Integer *Integer*	integer from integer value
Integer *Character*	integer from character value

Operators Summary

Operator	Description
==	return true if integer or real and equal
!=	return true if integer or real and not equal
+	return the sum with an integer or real
-	return the negation or substraction with an integer or real
*	return the multiplication with an integer or real
/	return the inverse or division with an integer or real
<	return true if less than an integer or real
<=	return true if less equal than an integer or real
>	return true if greater than an integer or real
>=	return true if greater equal than an integer or real
++	return this integer incremented by one
–	return this integer decremented by one
+=	return this integer summed with the argument
-=	return this integer substracted with the argument
*=	return this integer multiplied with the argument
/=	return this integer divided with the argument

Methods Summary

Method	Description
or	binary or with the argument
abs	return the absolute value
not	return the binary negation
shl	shift left by a certain amount
shr	shift right by a certain amount
and	binary and with the argument
xor	binary xor with the argument
mod	return the modulo with the argument
odd-p	return true with an odd number
even-p	return true with an even number
zero-p	return true if the integer is null

Integer:or

- return: `Integer`
- arguments: `Integer`

The **`or`** method returns the binary or between the integer and the integer argument.

Integer:abs

- return: `Integer`
- arguments: `none`

The **`abs`** method returns the absolute value of the calling integer instance.

Integer:not

- return: `Integer`
- arguments: `none`

The **`not`** method returns the binary negation of the calling integer instance.

Integer:shl

- return: `Integer`
- arguments: `Integer`

The **`shl`** method returns a new integer corresponding to the calling integer instance shifted left by the integer argument.

Integer:shr

- return: `Integer`
- arguments: `Integer`

The **shr** method returns a new integer corresponding to the calling integer instance shifted right by the integer argument.

Integer:and

- return: `Integer`
- arguments: `Integer`

The **and** method returns a new integer corresponding to the binary and between the calling integer instance and the integer argument.

Integer:xor

- return: `Integer`
- arguments: `Integer`

The **xor** method returns a new integer corresponding to the binary xor between the calling integer instance and the integer argument.

Integer:mod

- return: `Integer`
- arguments: `Integer`

The **mod** method returns the modulo between the integer instance and the integer argument. A `type-error` exception is raised if the argument is not an argument.

Integer:odd-p

- return: `Boolean`
- arguments: `none`

The **odd-p** method returns `true` if the integer instance is odd, false otherwise.

Integer:even-p

- return: `Boolean`
- arguments: `none`

The **even-p** method returns `true` if the integer instance is even, false otherwise.

Integer:zero-p

- return: `Boolean`
- arguments: `none`

The **zero-p** method returns `true` if the integer instance is null, false otherwise.

Relatif [reserved]

Description

The **Relatif** reserved object implements the behavior of an unlimited signed integer type. Standard decimal notation followed by the 'r' or 'R' character is used to construct relatif object from a literal. The relatif object can also be constructed from a string. This class is similar to the **Integer** class.

Derivation summary

Derived from	Description
Literal	the base literal object

Constructors Summary

Constructor	Description
Relatif	default relatif to 0
Relatif *Real*	relatif from real value
Relatif *String*	relatif from string value
Relatif *Integer*	relatif from integer value
Relatif *Relatif*	relatif from relatif value
Relatif *Character*	relatif from character value

Operators Summary

Methods Summary

Relatif:or

- return: `Relatif`
- arguments: `Relatif`

The **`or`** method returns the binary or between the relatif and the relatif argument.

Relatif:abs

- return: `Relatif`
- arguments: `none`

The **`abs`** method returns the absolute value of the calling relatif instance.

Relatif:not

Operator	Description
==	return true if relatif or integer equal
!=	return true if relatif or integer and not equal
+	return the sum with an relatif or integer
-	return the negation or substraction with an relatif or integer
*	return the multiplication with an relatif or integer
/	return the inverse or division with an relatif or integer
<	return true if less than an relatif or integer
<=	return true if less equal than an relatif or integer
>	return true if greater than an relatif or integer
>=	return true if greater equal than an relatif or integer
++	return this relatif incremented by one
–	return this relatif decremented by one
+=	return this relatif summed with the argument
-=	return this relatif substracted with the argument
*=	return this relatif multiplied with the argument
/=	return this relatif divided with the argument

Method	Description
or	binary or with the argument
abs	return the absolute value
not	return the binary negation
shl	shift left by a certain amount
shr	shift right by a certain amount
and	binary and with the argument
xor	binary xor with the argument
mod	return the modulo with the argument
odd-p	return true with an odd number
even-p	return true with an even number
zero-p	return true if the relatif is null

- return: `Relatif`
- arguments: `none`

The **`not`** method returns the binary negation of the calling relatif instance.

Relatif:shl

- return: `Relatif`
- arguments: `Relatif`

The **`shl`** method returns a new relatif corresponding to the calling relatif instance shifted left by the relatif argument.

Relatif:shr

- return: `Relatif`
- arguments: `Relatif`

The **shr** method returns a new relatif corresponding to the calling relatif instance shifted right by the relatif argument.

Relatif:and

- return: `Relatif`
- arguments: `Relatif`

The **and** method returns a new relatif corresponding to the binary and between the calling relatif instance and the relatif argument.

Relatif:xor

- return: `Relatif`
- arguments: `Relatif`

The **xor** method returns a new relatif corresponding to the binary xor between the calling relatif instance and the relatif argument.

Relatif:mod

- return: `Relatif`
- arguments: `Integer`

The **mod** method returns the modulo between the relatif instance and the relatif argument. A `type-error` exception is raised if the argument is not an argument.

Relatif:odd-p

- return: `Boolean`
- arguments: `none`

The **odd-p** method returns `true` if the relatif instance is odd, false otherwise.

Relatif:even-p

- return: `Boolean`
- arguments: `none`

The **even-p** method returns `true` if the relatif instance is even, false otherwise.

Relatif:zero-p

- return: `Boolean`
- arguments: `none`

The **zero-p** method returns `true` if the relatif instance is null, false otherwise.

Real [reserved]

Description

The **Real** reserved object implements the behavior of a double floating point number type. Standard decimal dot notation or scientific notation is used to construct real object from a literal. The real object can also be constructed from an integer, a character or a string.

Derivation summary

Derived from	Description
Literal	the base literal object

Constructors Summary

Constructor	Description
Real	default real 0.0
Real *Real*	real from real value
Real *Integer*	real from integer value
Real *String*	real from string value
Real *Character*	real from character value

Operators Summary

Operator	Description
==	return true if integer or real and equal
?=	return true if integer or real and equal at the precision
!=	return true if integer or real and not equal
+	return the sum with an integer or real
-	return the negation or substraction with an integer or real
*	return the multiplication with an integer or real
/	return the inverse or division with an integer or real
<	return true if less than an integer or real
<=	return true if less equal than an integer or real
>	return true if greater than an integer or real
>=	return true if greater equal than an integer or real
++	return this real incremented by one
–	return this real decremented by one
+=	return this real summed with the argument
-=	return this real substracted with the argument
*=	return this real multiplied with the argument
/=	return this real divided with the argument

Methods Summary

Method	Description
abs	return the absolute value
sqrt	return the square root
log	return the natural logarithm
exp	return the exponential
sin	return the sine
cos	return the cosine
tan	return the tangent
asin	return the arc sine
acos	return the arc cosine
atan	return the arc tangent
sinh	return the hyperbolic sine
cosh	return the hyperbolic cosine
tanh	return the hyperbolic tangent
nan-p	return true if nan
floor	return the floor
asinh	return the hyperbolic arc sine
acosh	return the hyperbolic arc cosine
atanh	return the hyperbolic arc tangent
zero-p	return true if null
format	return a formatted string
ceiling	return the ceiling

Real:nan-p

- return: `Boolean`
- arguments: `none`

The **`nan-p`** method returns `true` is the calling real number instance is not-a-number (nan).

Real:ceiling

- return: `Real`
- arguments: `none`

The **`ceiling`** method returns the ceiling of the calling real number instance.

Real:floor

- return: `Real`
- arguments: `none`

The **`floor`** method returns the floor of the calling real number instance.

Real:abs

- return: `Real`

- arguments: `none`

The **`abs`** method returns the absolute value of the calling real number instance.

Real:sqrt

- return: `Real`
- arguments: `none`

The **`sqrt`** method returns the square root of the calling real number instance.

Real:log

- return: `Real`
- arguments: `none`

The **`log`** method returns the natural logarithm of the calling real number instance.

Real:exp

- return: `Real`
- arguments: `none`

The **`exp`** method returns the exponential of the calling real number instance.

Real:sin

- return: `Real`
- arguments: `none`

The **`sin`** method returns the sine of the calling floating point instance. The angle is expressed in radian.

Real:cos

- return: `Real`
- arguments: `none`

The **`cos`** method returns the cosine of the calling floating point instance. The angle is expressed in radian.

Real:tan

- return: `Real`
- arguments: `none`

The **`tan`** method returns the tangent of the calling floating point instance. The angle is expressed in radian.

Real:asin

- return: `Real`
- arguments: `none`

The **`asin`** method returns the arc sine of the calling floating point instance. The result is in radian.

Real:acos

- return: `Real`
- arguments: `none`

The **`acos`** method returns the arc cosine of the calling floating point instance. The result is in radian.

Real:atan

- return: `Real`
- arguments: `none`

The **`atan`** method returns the arc tangent of the calling floating point instance. The result is in radian.

Real:sinh

- return: `Real`
- arguments: `none`

The **`sinh`** method returns the hyperbolic sine of the calling real number instance.

Real:cosh

- return: `Real`
- arguments: `none`

The **`cosh`** method returns the hyperbolic cosine of the calling real number instance.

Real:tanh

- return: `Real`
- arguments: `none`

The **`atan`** method returns the hyperbolic tangent of the calling real number instance.

Real:asinh

- return: `Real`
- arguments: `none`

The **`asinh`** method returns the hyperbolic arc sine of the calling real number instance.

Real:acosh

- return: `Real`
- arguments: `none`

The **`acosh`** method returns the hyperbolic arc cosine of the calling real number instance.

Real:atanh

- return: `Real`
- arguments: `none`

The **`atanh`** method returns the hyperbolic arc tangent of the calling real number instance.

Real:zero-p

- return: `Boolean`
- arguments: `none`

The **`zero-p`** method returns true if the calling real instance is null, false otherwise.

Real:format

- return: `String`
- arguments: `Integer`

The **`format`** method format the calling real instance with `n` digits after the decimal point. The number of digits is the format argument.

Character [reserved]

Description

The **Character** reserved object implements the behavior of an 8 bit character type. A character can be constructed from a literal quoted notation or with a string. Various methods are provided to compare or convert characters.

Derivation summary

Derived from	Description
Literal	the base literal object

Constructors Summary

Constructor	Description
Character	default null character
Character *Character*	character from character value
Character *Integer*	character from integer value
Character *String*	character from string value

Operators Summary

Operator	Description
==	return true if character and equal
!=	return true if character and not equal
<	return true if less than a character
<=	return true if less equal than a character
>	return true if greater than a character
>=	return true if greater equal than a character
++	return this character incremented by one
–	return this character decremented by one
+=	return this character summed with the argument
-=	return this character substracted with the argument

Methods Summary

Character:incr

- return: `Character`
- arguments: `none`

Method	Description
alpha-p	return true if the character is alphabetic
digit-p	return true if the character is a digit
blank-p	return true if the character is a blank or tab
eol-p	return true if the character is an end of line
eof-p	return true if the character is an end of file
nil-p	return true if the character is nil
to-integer	return an integer representation

The **`incr`** method increments the current character value by one.

Character:decr

- return: `Character`
- arguments: `none`

The **`decr`** method decrements the current character value by one.

Character:alpha-p

- return: `Boolean`
- arguments: `none`

The **`alpha-p`** method returns `true` if character is an alphabetic character, `false` otherwise.

Character:digit-p

- return: `Boolean`
- arguments: `none`

The **`digit-p`** method returns `true` if character is a digit character, `false` otherwise.

Character:blank-p

- return: `Boolean`
- arguments: `none`

The **`blank-p`** method returns `true` if character is a blank or tab character, `false` otherwise.

Character:eol-p

- return: `Boolean`
- arguments: `none`

The **`eol-p`** method returns `true` if character is an end of line character, `false` otherwise.

Character:eof-p

- return: `Boolean`

- arguments: `none`

The **`eof-p`** method returns `true` if character is an end of file character, `false` otherwise.

Character:nil-p

- return: `Boolean`
- arguments: `none`

The **`nil-p`** method returns `true` if character is the nil character, `false` otherwise.

String [reserved]

Description

The **String** reserved object implements the behavior of an internal character array. The double quote notation is the literal notation for a string. A string can also be constructed from the standard Aleph objects. Strings can be compared, transformed or extracted with the help of the methods listed below.

Derivation summary

Derived from	Description
Literal	the base literal object

Constructor Summary

Constructor	Description
String	default null string
String *String*	string from string value
String *Real*	string from real value
String *Integer*	string from integer value
String *Boolean*	string from boolean value

Operators Summary

Operator	Description
==	return true if string and equal
!=	return true if string and not equal
+	return the sum with a literal
+=	return this string summed with a literal

Methods Summary

String:length

- return: `Integer`
- arguments: `none`

The **`length`** method returns the length of the string.

String:strip-left

Method	Description
length	return the length of this string
strip-left	remove leading blanks and tabs
strip-right	remove trailing blanks and tabs
strip	remove leading and trailing blanks
split	split a string into a vector
extract	extract strings from a string
to-upper	convert to upper case
to-lower	convert to lower case
get	return a character by index
sub-left	return a left sub string
sub-right	return a right sub string
fill-left	return a string filled on the left
fill-right	return a string filled on the right
substr	return a sub string by index

- return: `String`
- arguments: `none`

The **`strip-left`** method removes the leading blanks and tabs and returns a new string.

String:strip-right

- return: `String`
- arguments: `none`

The **`strip-right`** method removes the trailing blanks and tabs and returns a new string.

String:strip

- return: `String`
- arguments: `none`

The **`strip`** method removes the leading, trailing blanks and tabs and returns a new string.

String:split

- return: `Vector`
- arguments: `none|String`

The **`split`** method split the string into one or more string according to break sequence. If no argument is passed to the call, the break sequence is assumed to be a blank, tab and eol characters.

String:extract

- return: `Vector`
- arguments: `Character`

The **`extract`** method extracts one or more string which are enclosed by a control character passed as an argument. The method returns a vector of strings.

String:to-upper

- return: `String`
- arguments: `none`

The **`to-upper`** converts all string characters to upper case and returns a new string.

String:to-lower

- return: `String`
- arguments: `none`

The **`to-lower`** method converts all string characters to lower case and returns a new string.

String:get

- return: `Character`
- arguments: `Integer`

The **`get`** method returns a the string character at the position given by the argument. If the index is invalid, an exception is raised.

String:sub-left

- return: `String`
- arguments: `Integer`

The **`sub-left`** method returns the left sub string of the calling string up-to the argument index. If the index is out of range, the string is returned.

String:sub-right

- return: `String`
- arguments: `Integer`

The **`sub-right`** method returns the right sub string of the calling string starting at the argument index. If the index is out of range, the string is returned.

String:fill-left

- return: `String`
- arguments: `Character Integer`

The **`fill-left`** method returns a string filled on the left with the character argument. The second argument is the desired length of the resulting string. If the calling is too long, the string is returned.

String:fill-right

- return: `String`
- arguments: `Character Integer`

The **`fill-left`** method returns a string filled on the right with the character argument. The second argument is the desired length of the resulting string. If the calling is too long, the string is returned.

String:substr

- return: `String`
- arguments: `Integer Integer`

The **`substr`** method returns a string starting at the first argument index and ending at the second argument index. If the indexes are out of range, an exception is raised.

regex [reserved]

Description

The `Regex` object is a special object which is automatically instantiated by the interpreter when using the delimiter character `[` and `]`. The *regex* syntax involves the use of standard characters, meta characters and control characters. Additionnaly, a string can be use to specify a series of characters.

In its first form, the '[' and ']' characters are used as syntax delimiters. The lexical analyzer automatically recognizes this token as a *regex* and built the equivalent `Regex` object. The second form is the explicit construction of the `Regex` object. Note also that the '[' and ']' characters are also used as *regex* block delimiters.

Any character, except the one used as operators can be used in a *regex*. The '$' character is used as a meta-character (or control character) to represent a particular set of characters. For example, `[hello world]` is a *regex* which match only the `"hello world"` string. The `[$d+]` *regex* matches one or more digits. The following control characters are builtin in the *regex* engine.

- **$a** matches any letter or digit.
- **$b** matches any blank characters.
- **$d** matches any digit.
- **$l** matches any lower case letter.
- **$n** matches new line characters.
- **$s** matches any letter.
- **$u** matches any upper case letter.
- **$w** matches any aleph word constituent.
- **$x** matches any hexadecimal characters.

The uppercase version is the complement of the corresponding lowercase character set. A character which follows a $ character and that is not a meta character is treated as a normal character. For example `$[` is the '[' character. A quoted string can be used to define character matching which could otherwise be interpreted as control characters or operator. A quoted string also interprets standard *escaped* sequences but not meta characters.

- **$A** matches any character except letter or digit.
- **$B** matches any character except blanks.
- **$D** matches any character except digit.
- **$L** matches any character except lower case letters.
- **$N** matches any character except new line.
- **$S** matches any character except letters.

- **$U** matches any character except upper case letters.
- **$W** matches any character except aleph word constituents.
- **$X** matches any character except hexadecimal characters.

A character set is defined with the '<' and '>' characters. Any enclosed character defines a character set. Note that meta characters are also interpreted inside a character set. For example, `<$d+->` represents any digit or a plus or minus. If the first character is the ˆ character in the character set, the character set is complemented with regards to its definition.

The following unary operators can be used with single character, control characters and sub-expressions.

- ***** match zero or more times
- **+** match one or more times
- **?** match zero or one time.
- | alternation

Alternation is an operator which work with a secondary expression. Care should be taken when writing the right sub-expression. Groups of sub-expressions are created with the '(' and ')' characters. When a group is matched, the resulting sub-string is placed on stack and can be later used. In this respect, the *regex* engine can be used to extract sub-strings.

Derivation summary

Derived from	**Description**
Literal	the base literal object

Constructors Summary

Constructor	**Description**
Regex	default regex object
Regex *String*	regex with string specification

Operators Summary

Operator	**Description**
==	return true if the string is matched
!=	return true if the string does not match
<	return true if the string partially match

Methods Summary

Regex:length

Method	Description
get	returns a group sub-string by index
length	returns the length of the group vector
match	returns the first matching string
replace	replace all matching strings with the argument

- return: `Integer`
- arguments: `none`

The **`length`** method returns the length of the group vector when a *regex* match has been successful.

Regex:get

- return: `String`
- arguments: `Integer`

The **`get`** method returns by index the group sub-string when a *regex* match has been successful.

Regex:match

- return: `String`
- arguments: `String`

The **`match`** method returns the first matching string of the argument string.

Regex:replace

- return: `String`
- arguments: `String String`

The **`replace`** method returns a string constructed by replacing all matching sub-string (from the first argument) with the second argument string.

Appendix C

Container Objects

This chapter is a reference of the Aleph reserved container objects with their respective builtin methods. Some of these container objects are *iterable* objects.

Object	Description
Cons	cons cell and single linked list
List	double linked list
Node	graph node object
Edge	graph edge object
Graph	general graph
Queue	queue object
Vector	array index vector
Bitset	bit set object
Buffer	buffer object

Table C.1: Aleph container objects

Object	Description
Cons	cons cell and single linked list
List	double linked list
Vector	array index vector

Table C.2: Aleph iterable containers

For each container object, Aleph provides a *predicate* which can be used to test for the object type.

When an object is iterable, an iterator constructor constructor is provided. The *iterable-p* predicate returns true if the container is an iterable object. The `get-iterator` method can be used to construct an object iterator. For a given iterator, the predicates `end-p` and `valid-p` can be used to check for the end or a valid iterator position. The `next` method move the iterator to its next position. The `prev` method move the iterator (if possible) to its previous position. The `get-object` method returns the object at the current iterator position.

Object	**Predicate**
Cons	cons-p
List	list-p
Node	node-p
Edge	edge-p
Graph	graph-p
Queue	queue-p
Vector	vector-p
Bitset	bitset-p
Buffer	buffer-p

Table C.3: Aleph container object predicates

Cons [reserved]

Description

A `Cons` instance or simply a *cons cell* is a simple element used to build linked list. The cons cell holds an object and a pointer to the next cons cell. The cons cell object is called `car` and the next cons cell is called the `cdr`. Historically, `car` means *Current Address Register* and `cdr` means *Current Data Register*. We retain in Aleph this notation for the sake of tradition.

Constructors Summary

Constructor	Description
Cons	default cons cell with nil car
Cons *object-list*	linked cons cell with arguments

Methods Summary

Method	Description
get-car	returns the car of the cons cell
get-cdr	returns the cdr of the cons cell
get-cadr	returns the car of the cdr or nil
get-caddr	returns the car of the cdr of the cdr or nil
get-cadddr	returns the car of the cdr of the cdr of the cdr or nil
length	returns the length of the cons cell
nil-p	returns true if the car is nil
block-p	returns true if the cons cell is a block
get-iterator	returns a forward iterator
set-car	set the car of the cons cell
set-cdr	set the cdr of the cons cell
append	appends an object at the end of the cons cell
link	appends an object or set the car if nil
get	returns an object at a certain position

Cons:get-car

- return: `Object`
- arguments: `none`

The **`get-car`** method returns the car of the calling cons cell.

Cons:get-cdr

- return: `Cons`
- arguments: `none`

The **`get-cdr`** method returns the cdr of the calling cons cell.

Cons:get-cadr

- return: `Object`
- arguments: `none`

The **`get-cadr`** method returns the car of the cdr of the calling cons cell or nil if the cdr is nil.

Cons:get-caddr

- return: `Object`
- arguments: `none`

The **`get-caddr`** method returns the car of the cdr of the cdr of the calling cons cell or nil if the cdr is nil.

Cons:get-cadddr

- return: `Object`
- arguments: `none`

The **`get-cadddr`** method returns the car of the cdr of the cdr of the cdr of the calling cons cell or nil if the cdr is nil.

Cons:length

- return: `Integer`
- arguments: `none`

The **`length`** method returns the length of the cons cell. The minimum length returned is always 1.

Cons:nil-p

- return: `Boolean`
- arguments: `none`

The **`nil-p`** predicate returns `true` if the car of the calling cons cell is nil, `false` otherwise.

Cons:block-p

- return: `Boolean`
- arguments: `none`

The **`block-p`** predicate returns `true` if the cons cell is of type block, `false` otherwise.

Cons:get-iterator

- return: `Iterator`

- arguments: `none`

The **`get-iterator`** returns a forward iterator for this cons cell. No backward methods are supported for this object.

Cons:set-car

- return: `Object`
- arguments: `Object`

The **`set-car`** set the car of the calling cons cell. The object argument is returned by the method.

Cons:set-cdr

- return: `Cons`
- arguments: `Cons`

The **`set-cdr`** set the cdr of the calling cons cell. The cons cell argument is returned by the method.

Cons:append

- return: `Object`
- arguments: `Object`

The **`append`** method appends an object at the end of the cons cell chain by creating a new cons cell and linking it with the last cdr. The object argument is returned by this method.

Cons:link

- return: `Object`
- arguments: `Object`

The **`append`** method is similar to `append` except that a new cons cell is not created if the car is nil. Instead the car is set with the calling object. The object argument is returned by this method.

Cons:get

- return: `Object`
- arguments: `Integer`

The **`get`** method returns the car of the cons cell chain at a certain position specified by the integer index argument.

List [reserved]

Description

The `List` builtin object provides the facility of a double-link list. The `List` object is another example of *iterable* object. The `List` object provides support for forward and backward iteration.

Constructors Summary

Constructor	Description
List	empty double linked list
List *object-list*	double linked list with arguments

Methods Summary

Method	Description
length	returns the length of the cons cell
get-iterator	returns a forward iterator
append	appends an object at the end of the cons cell
insert	inserts an object or set the car if nil
get	returns an object at a certain position

List:length

- return: `Integer`
- arguments: `none`

The **`length`** method returns the length of the list. The minimum length is 0 for an empty list.

List:get-iterator

- return: `Iterator`
- arguments: `none`

The **`get-iterator`** returns a forward/backward iterator for this list.

List:append

- return: `Object`
- arguments: `Object`

The **`append`** method appends an object at the end of the list. The object argument is returned by this method.

List:insert

- return: `Object`
- arguments: `Object`

The **`insert`** method inserts an object at the beginning of the list. The object argument is returned by this method.

List:get

- return: `Object`
- arguments: `Integer`

The **`get`** method returns the object in the list at a certain position specified by the integer index argument.

Vector [reserved]

Description

The `Vector` builtin object provides the facility of an index array of objects. The `Vector` object is another example of *iterable* object. The `Vector` object provides support for forward and backward iteration.

Constructors Summary

Constructor	Description
Vector	empty vector
Vector *object-list*	vector with arguments

Methods Summary

Method	Description
length	returns the length of the cons cell
get-iterator	returns a forward iterator
append	appends an object at the end of the cons cell
insert	inserts an object or set the car if nil
get	returns an object at a certain position
set	set an object at a certain position
exists	return true if the object argument exists

Vector:length

- return: `Integer`
- arguments: `none`

The **`length`** method returns the length of the vector. The minimum length is 0 for an empty vector.

Vector:get-iterator

- return: `Iterator`
- arguments: `none`

The **`get-iterator`** returns a forward/backward iterator for this vector.

Vector:append

- return: `Object`
- arguments: `Object`

The **`append`** method appends an object at the end of the vector. The object argument is returned by this method.

Vector:get

- return: `Object`
- arguments: `Integer`

The **`get`** method returns the object in the vector at a certain position specified by the integer index argument.

Vector:set

- return: `Object`
- arguments: `Integer Object`

The **`set`** method set a vector position with an object. The first argument is the vector index. The second argument is the object to set. The method returns the object to set.

Vector:exists

- return: `Boolean`
- arguments: `Object`

The **`exists`** method returns true if the object argument exists in the vector. This method is useful to make sure that only one occurrence of an object is added to a vector.

Node [reserved]

Description

The `Node` builtin object is part of the graph facility of the Aleph standard objects. A node (or vertex) is a graph components which is linked with edges (object Edge). A node can hold a client object. Once a node has been constructed, it is possible to link it with edges and add it to a graph.

Constructors Summary

Constructor	Description
Node	default node
Node *object*	node with client object

Methods Summary

Method	Description
get-client	returns the node client object
set-client	set the node client object
degree	returns the node degree
input-degree	returns the node input degree
output-degree	returns the node input degree
add-input-edge	link this node with an input edge
add-output-edge	link this node with an output edge
get-input-edge	link this node with an input edge
get-output-edge	link this node with an output edge

Node:get-client

- return: `Object`
- arguments: `none`

The **`get-client`** method returns the node client object. If the client object is not set, nil is returned.

Node:set-client

- return: `Object`
- arguments: `Object`

The **`set-client`** method sets the node client object. The object is returned by this method.

Node:degree

- return: `Integer`
- arguments: `none`

The **`degree`** method returns the node degree, that is the sum of the input degree and the output degree.

Node:input-degree

- return: `Integer`
- arguments: `none`

The **`input-degree`** method returns the node input degree, that is the number of input edges to this node.

Node:output-degree

- return: `Integer`
- arguments: `none`

The **`output-degree`** method returns the node output degree, that is the number of output edges for this node.

Node:add-input-edge

- return: `Edge`
- arguments: `Edge`

The **`add-input-edge`** method adds the edge object argument to the node input edge list. This method also sets the edge target node and increase the node input degree. The edge argument is returned by this method.

Node:add-output-edge

- return: `Edge`
- arguments: `Edge`

The **`add-output-edge`** method adds the edge object argument to the node output edge list. This method also sets the edge source node and increase the node output degree. The edge argument is returned by this method.

Node:get-input-edge

- return: `Edge`
- arguments: `Integer`

The **`get-input-edge`** method returns the node input edge by index. If the index is negative or bigger that the node input degree, an exception is raised.

Node:get-output-edge

- return: `Edge`
- arguments: `Integer`

The **`get-output-edge`** method returns the node output edge by index. If the index is negative or bigger that the node output degree, an exception is raised.

Edge [reserved]

Description

The `Edge` builtin object is part of the graph facility of the Aleph standard objects. An edge is a graph component which connects two nodes called respectively the source and target nodes. An edge can be constructed alone, with two edges or with a client object. The edge can also be added later to the graph.

Constructors Summary

Constructor	Description
Edge	default edge
Edge *object*	edge with client object
Edge *node node*	edge with source and target nodes

Methods Summary

Method	Description
get-source	returns the edge source node
get-target	returns the edge target node
get-client	returns the node client object
set-source	set the edge source node
set-target	set the edge target node
set-client	set the node client object

Edge:get-source

- return: `Node`
- arguments: `none`

The **`get-source`** method returns the edge source node. If the client object is not set, nil is returned.

Edge:get-target

- return: `Node`
- arguments: `none`

The **`get-target`** method returns the edge target node. If the client object is not set, nil is returned.

Edge:get-client

- return: `Object`
- arguments: `none`

The **`get-client`** method returns the node client object. If the client object is not set, nil is returned.

Edge:set-source

- return: `Node`
- arguments: `Node`

The **`set-source`** method sets the edge source node. The node is returned by this method.

Edge:set-target

- return: `Node`
- arguments: `Node`

The **`set-target`** method sets the edge target node. The node is returned by this method.

Edge:set-client

- return: `Object`
- arguments: `Object`

The **`set-client`** method sets the node client object. The object is returned by this method.

Graph [reserved]

Description

The `Graph` builtin object is a general graph class that manages a set of nodes and edges. The graph is built by adding node and edges to the graph. Additional edges can be added by connecting nodes.

Constructors Summary

Constructor	Description
Graph	default graph

Methods Summary

Method	Description
add	add a node or edge
exists	checks if a node or edge exists
get-edge	return an edge by index
get-node	return a node by index
number-of-nodes	return the number of nodes
number-of-edges	return the number of edges

Graph:add

- return: `Object`
- arguments: `Node` | `Edge`

The **`add`** method adds a node or an edge to the graph. When adding an edge, the methods check that the source and target nodes are also part of the graph.

Graph:exists

- return: `Boolean`
- arguments: `Node` | `Edge`

The **`exists`** method returns true if the node or edge argument exists in the graph.

Graph:get-edge

- return: `Edge`
- arguments: `Integer`

The **`get-edge`** method returns an edge by index. If the index is out of range, an exception is raised.

Graph:get-node

- return: `Node`
- arguments: `Integer`

The **`get-node`** method returns a node by index. If the index is out of range, an exception is raised.

Graph:number-of-nodes

- return: `Integer`
- arguments: `none`

The **`number-of-nodes`** methods returns the number of nodes in the graph.

Graph:number-of-edges

- return: `Integer`
- arguments: `none`

The **`number-of-edges`** methods returns the number of edges in the graph.

Queue [reserved]

Description

The `Queue` builtin object is a container used to queue and dequeue objects. The order of entry in the queue defines the order of exit from the queue. The queue is constructed either empty or with a set of objects.

Constructors Summary

Constructor	Description
Queue	default queue
Queue `objects...`	queue with objects

Methods Summary

Method	Description
flush	flush the queue
enqueue	enqueue an object
dequeue	dequeue an object
length	returns the numbers of queued objects
empty-p	returns true if the queue is empty

Queue:enqueue

- return: `Object`
- arguments: `Object`

The **`enqueue`** adds an object in the queue and returns the queued object.

Queue:dequeue

- return: `Object`
- arguments: `none`

The **`dequeue`** dequeue an object in the order it was queued.

Queue:length

- return: `Object`
- arguments: `none`

The **`length`** returns the number of queued objects.

Queue:empty-p

- return: `Object`
- arguments: `none`

The **`empty-p`** method returns true if the queue is empty.

Queue:flush

- return: `none`
- arguments: `none`

The **`flush`** method flushes the queue so that it is empty.

Bitset [reserved]

Description

The `Bitset` builtin object is a container for multi bit storage. The size of the bitset is determined at construction. With the use of an index, a particular bit can be set, cleared and tested.

Constructors Summary

Constructor	Description
Bitset	default bitset
Bitset `size`	bitset with size

Methods Summary

Method	Description
get	get a bit by index
set	set a bit by index
mark	mark a bit by index
clear	clear a bit by index
length	returns the bitset length

Bitset:get

- return: `Boolean`
- arguments: `Integer`

The **`get`** method returns the bit value by the index argument.

Bitset:set

- return: `none`
- arguments: `Integer Boolean`

The **`set`** method set the bit value by the index argument with the boolean second argument.

Bitset:mark

- return: `none`
- arguments: `Integer`

The **`mark`** method marks a bit by the index argument.

Bitset:clear

- return: `none`
- arguments: `Integer`

The **`clear`** method clears a bit by the index argument.

Bitset:length

- return: `Integer`
- arguments: `none`

The **`length`** method returns the length of the bitset.

Buffer [reserved]

Description

The `Buffer` builtin object is a character buffer that is widely used with i/o operations. The buffer can be constructed with or without literal arguments. The standard methods to add or pushback characters are available. One attractive method is the write method which can write a complete buffer to an output stream specified as an argument.

Constructors Summary

Constructor	Description
Buffer	default buffer
Buffer [`literal`]	buffer with literals

Methods Summary

Method	Description
add	add a literal or buffer
get	get a character
read	read a character
reset	reset this buffer
length	return the buffer length
write	write the buffer to an output stream
to-string	return a string representation
get-word	get a word from a network byte order
get-quad	get a quad from a network byte order
get-octa	get a octa from a network byte order

Buffer:add

- return: `none`
- arguments: `Literal|Buffer`

The **`add`** method add a literal object or a buffer to the buffer. The literal object is automatically converted to a sequence of characters. For a buffer, the entire content is copied into the buffer.

Buffer:get

- return: `Character`
- arguments: `none`

The **`get`** method returns the next available character in the buffer but do not remove it.

Buffer:read

- return: `Character`
- arguments: `none`

The **`read`** method returns the next available character and remove it from the buffer.

Buffer:reset

- return: `none`
- arguments: `none`

The **`reset`** method reset the entire buffer and destroy its contents.

Buffer:length

- return: `Integer`
- arguments: `none`

The **`length`** method returns the length of the buffer.

Buffer:write

- return: `none`
- arguments: `Output`

The **`write`** method writes the buffer contents to the output stream argument.

Buffer:to-string

- return: `String`
- arguments: `none`

The **`to-string`** method returns a string representation of the buffer.

Buffer:pushback

- return: `none`
- arguments: `Literal`

The **`add`** method push back a literal object in the buffer. The literal object is automatically converted to a sequence of characters.

Buffer:get-word

- return: `Integer`
- arguments: `none`

The **`get-word`** method reads a word from the buffer and convert it to an integer. The word is assumed to be in network byte order and is converted to the host format before becoming an integer.

Buffer:get-quad

- return: `Integer`
- arguments: `none`

The **`get-quad`** method reads a quad from the buffer and convert it to an integer. The quad is assumed to be in network byte order and is converted to the host format before becoming an integer.

Buffer:get-octa

- return: `Integer`
- arguments: `none`

The **`get-quad`** method reads an octa from the buffer and convert it to an integer. The octa is assumed to be in network byte order and is converted to the host format before becoming an integer.

Appendix D

Special Objects

This chapter is a reference of the Aleph reserved special objects with their respective builtin methods. Special objects are those objects which interact with the interpreter.

Object	Description
Object	base object
Interp	current interpreter
Thread	thread descriptor object
Condvar	condition variable object
Symbol	symbol name object
Closure	closure object
Lexical	lexical name object
Resolver	file path resolver object
Qualified	qualified name object
Librarian	librarian collector object

Table D.1: Aleph special objects

For each special objects (except `Object`) , Aleph provides a *predicate* which can be used to test for the object type.

Object	Predicate
Interp	interp-p
Thread	thread-p
Condvar	condvar-p
Closure	closure-p
Lexical	lexical-p
Symbol	symbol-p
Resolver	resolver-p
Qualified	qualified-p
Librarian	librarian-p

Table D.2: Aleph special object predicates

Object [reserved]

Description

The base object `Object` provides several methods which are common to all objects.

Methods Summary

Method	Description
repr	object representation string
rdlock	object read lock
wrlock	object write lock
unlock	object unlock
shared-p	shared object predicate

Object:repr

- return: `String`
- arguments: `none`

The **`repr`** method returns the object name in the form of a string. The result string is called the *representation* string.

Object:rdlock

- return: `none`
- arguments: `none`

The **`rdlock`** method try to acquire the object in read-lock mode. If the object is currently locked in write mode by another thread, the calling thread is suspended until the lock is released.

Object:wrlock

- return: `none`
- arguments: `none`

The **`wrlock`** method try to acquire the object in write-lock mode. If the object is currently locked by another thread, the calling thread is suspended until the lock is released.

Object:unlock

- return: `none`
- arguments: `none`

The **`unlock`** method try to unlock an object. An object will be unlocked if and only if the calling thread is the one who acquired the lock.

Object:shared-p

- return: `Boolean`
- arguments: `none`

The **`shared-p`** method returns `true` if the object is shared.

Interp [reserved]

Description

The interpreter object is automatically bounded for each executing. There is no constructor for this object. The current interpreter is bounded to the **interp** reserved symbol.

Data member Summary

Member	Description
argv	interpreter argument vector
os-name	operating system name
os-type	operating system type
version	full aleph version
program-name	interpreter program name
major-version	aleph major version number
minor-version	aleph minor version number
patch-version	aleph patch version number
aleph-url	aleph official url name

Methods Summary

Method	Description
load	load a file
clone	clone the interpreter
library	open a shared library
set-real-precision	set real precision
get-real-precision	get real precision

Interp:load

- return: `Boolean`
- arguments: `String`

The **`loop`** method opens a file those name is the method argument and execute each form in the file by doing a read-eval loop. When all forms have been executed, the file is closed and the method return `true`. In case of exception, the file is closed and the method returns `false`.

Interp:clone

- return: `Interp`
- arguments: `none`

The **clone** method returns a clone of the calling interpreter.

Interp:library

- return: `Library`
- arguments: `String`

The **library** method opens a shared library and a returns a shared library object.

Interp:launch

- return: `Thread`
- arguments: `form`

The **launch** method executes the form argument in a normal thread. The normal thread is created by cloning the current interpreter.

Interp:daemon

- return: `Thread`
- arguments: `form`

The **daemon** method executes the form argument in a daemon thread. The normal thread is created by cloning the current interpreter.

Interp:set-real-precision

- return: `none`
- arguments: `Real`

The **set-real-precision** method sets the interpreter real precision. The *real-precision* is used by the `?=` operator to compare real values.

Interp:get-real-precision

- return: `Real`
- arguments: `none`

The **get-real-precision** method returns the interpreter real precision. The *real-precision* is used by the `?=` operator to compare real values.

Thread [reserved]

Description

The `Thread` object is a special object which acts as a thread descriptor. Such object is created with the `launch` or `daemon` reserved keywords. Note that the thread object does not have a constructor.

Data member Summary

Member	Description
result	thread completion result

Methods Summary

Method	Description
wait	wait for a thread to complete
normal-p	normal thread predicate
daemon-p	daemon thread predicate

Thread:wait

- return: `none`
- arguments: `none`

The **`wait`** method suspends the calling thread until the thread argument as completed. The `wait` method is the primary mechanism to detect a thread completion.

Thread:normal-p

- return: `Boolean`
- arguments: `none`

The **`normal-p`** method returns `true` if the thread argument is a normal thread.

Thread:daemon-p

- return: `Boolean`
- arguments: `none`

The **`daemon-p`** method returns `true` if the thread argument is a normal thread.

Condvar [reserved]

Description

The condition variable `Condvar` object is a special object which provides a mean of synchronization between one and several threads. The condition is said to be false unless it have been marked. When a condition is marked, all threads waiting for that condition to become true are notified and one thread is run with that condition.

Methods Summary

Method	**Description**
lock	lock the condition variable mutex
mark	mark the condition variable and notify
wait	wait for a marking
reset	reset the condition variable
unlock	unlock the condition variable mutex
wait-unlock	wait for a marking, then reset and unlock

Condvar:lock

- return: `none`
- arguments: `none`

The **`lock`** method locks the condition variable mutex. If the mutex is already locked, the calling thread is suspended until the lock is released. the method returns, the resumed thread owns the condition variable lock. It is the thread responsibility to reset the condition variable and unlock it.

Condvar:mark

- return: `none`
- arguments: `none`

The **`mark`** method mark the condition variable and notify all pending threads of such change. The mark method is the basic notification mechanism.

Condvar:wait

- return: `none`
- arguments: `none`

The **`wait`** method wait for a condition variable to be marked. When such condition occurs, the suspended thread is run. When the method returns, the resumed thread owns the condition variable lock. It is the thread responsibility to reset the condition variable and unlock it.

Condvar:reset

- return: `none`
- arguments: `none`

The **`reset`** method acquire the condition variable mutex, reset the mark, and unlock it. If the lock has been taken, the calling thread is suspended.

Condvar:unlock

- return: `none`
- arguments: `none`

The **`unlock`** method unlock the condition variable mutex. This method should be used after a call to `lock` or `wait`.

Condvar:wait-unlock

- return: `none`
- arguments: `none`

The **`wait-unlock`** method wait until a condition variable is marked. When such condition occurs, the suspended thread is run. Before the method returns, the condition variable is reset and the mutex unlocked. With two threads to synchronize, this is the preferred method compared to `wait`.

Lexical [reserved]

Description

The `Lexical` object is a special object built by the **Aleph** reader. A lexical name is also a literal object. Although the best way to create a lexical name is with a form, the lexical object can also be constructed with a string name. A lexical name can be mapped to a symbol by using the `map` method.

Derivation summary

Derived from	Description
Literal	the literal object class

Constructors Summary

Constructor	Description
Lexical	create a nil lexical
Lexical *name*	create a lexical by name

Methods Summary

Method	Description
map	get the object mapped by the lexical

Lexical:map

- return: `Object`
- arguments: `none`

The **`map`** method returns the object that is mapped by the lexical name. Most of the time, a symbol object is returned since it is the kind of object stored in a nameset. Eventually the mapping might returns an argument object if used inside a closure.

Qualified [reserved]

Description

The `Qualified` object is a special object built by the **Aleph** reader. A qualified object is similar to a lexical object. It is also a literal object. Like a lexical name, a qualified name can be created with a form or by direct construction with a name. Like a lexical name, the `map` method can be used to retrieve the symbol associated with that name.

Derivation summary

Derived from	Description
Literal	the literal object class

Constructors Summary

Constructor	Description
Qualified	create a nil qualified
Qualified *name*	create a qualified by name

Methods Summary

Method	Description
map	get the object mapped by the qualified

Qualified:map

- return: `Object`
- arguments: `none`

The **`map`** method returns the object that is mapped by the qualified name. Most of the time, a symbol object is returned since it is the kind of object stored in a nameset. Eventually the mapping might returns an argument object if used inside a closure.

Symbol [reserved]

Description

The `Symbol` object is a special object used by nameset to map a name with an object. Generally a symbol is obtained by mapping a lexical or qualified name. As an object, the symbol holds a name, an object and a *const* flag. The symbol name cannot be changed since it might introduce inconsistencies in the containing nameset. On the other hand, the *const* flag and the object can be changed. A symbol is a literal object. A symbol that is not binded to a nameset can be constructed dynamically. Such symbol is said to be *not interned*.

Derivation summary

Derived from	**Description**
Literal	the literal object class

Constructors Summary

Constructor	**Description**
Symbol *name*	create a symbol by name
Symbol *name object*	create a symbol by name and object

Methods Summary

Method	**Description**
get-const	get the symbol const flag
set-const	set the symbol const flag
get-object	get the symbol object
set-object	set the symbol object

Symbol:get-const

- return: `Boolean`
- arguments: `none`

The **`get-const`** method returns the symbol const flag. If the flag is true, the symbol object cannot be changed unless that flags is reset with the `set-const` method.

Symbol:set-const

- return: `none`
- arguments: `Boolean`

The **`set-const`** method set the symbol const flag. This method is useful to mark a symbol as const or to make a const symbol mutable.

Symbol:get-object

- return: `Object`
- arguments: `none`

The **`get-object`** method returns the symbol object.

Symbol:set-object

- return: `none`
- arguments: `Object`

The **`set-object`** method set the symbol object.

Closure [reserved]

Description

The `Closure` object is a special object that represents a lambda or gamma expression. A closure is represented by a set of arguments, a set of closed variables and a form to execute. A boolean flag determines the type of closure. The closure predicate `lambda-p` returns `true` if the closure is a lambda expression. Closed variables can be defines and evaluated with the use of the qualified name mechanism. Closure mutation is achieved with the `add-argument` and `set-form` method. An empty closure can be defined at construction as well.

Derivation summary

Derived from	**Description**
Object	the base object class

Constructors Summary

Constructor	**Description**
Closure	create a default closure
Closure *type*	create a lambda expression if true

Methods Summary

Method	**Description**
lambda-p	return true for a lambda closure
get-form	get the closure form
set-form	set the closure form
add-argument	add an argument to the closure

Closure:lambda-p

- return: `Boolean`
- arguments: `none`

The **`lambda-p`** predicate returns true if the closure is a lambda expression. The predicate returns false for a gamma expression.

Closure:get-form

- return: `Object`
- arguments: `none`

The **`get-form`** method returns the closure form object.

Closure:set-form

- return: `none`
- arguments: `Object`

The **`set-form`** method sets the closure form object.

Closure:add-argument

- return: `none`
- arguments: `String|Lexical|form`

The **`add-argument`** method adds an argument to the closure. The argument object can be either a string, a lexical object of a simple form that defines a *const* lexical name.

Librarian [reserved]

Description

The `Librarian` object is a special object that read or write a librarian. Without argument, a librarian is created for writing purpose. With one file name argument, the librarian is created for reading.

Derivation summary

Derived from	Description
Object	the base object class

Constructors Summary

Constructor	Description
Librarian	create a librarian for writing
Librarian *name*	create a librarian for reading

Methods Summary

Method	Description
add	add a new file to the librarian
write	write a librarian
length	return the librarian length
extract	extract a file from the librarian
exists-p	check if a file exists in the librarian
get-names	returns a vector of file in the librarian

Librarian:add

- return: `none`
- arguments: `String`

The **`add`** method adds a file into the librarian. The librarian must have been opened in write mode.

Librarian:write

- return: `none`
- arguments: `String`

The **`write`** method writes a librarian to a file those name is the argument.

Librarian:length

- return: `Integer`
- arguments: `none`

The **`length`** method returns the number of file in the librarian. This method work, no matter how the librarian has been opened.

Librarian:exists-p

- return: `Boolean`
- arguments: `String`

The **`exists-p`** predicate returns true if the file argument exists in the librarian.

Librarian:extract

- return: `InputMapped`
- arguments: `String`

The **`extract`** method returns an input stream mapped to the file name argument.

Resolver [reserved]

Description

The `Resolver` object is a special object that gives the ability to open a file based on a file path resolver. The resolver maintains a list of valid path and returns an input stream for a file on demand.

Derivation summary

Derived from	Description
Object	the base object class

Constructors Summary

Constructor	Description
Resolver	create a default resolver

Methods Summary

Method	Description
add	add a new path to the resolver
lookup	find a file by resolving its name
valid-p	check for a valid file

Resolver:add

- return: `none`
- arguments: `String`

The **`add`** method adds a path into the resolver. The path can points either to a directory or a librarian.

Resolver:lookup

- return: `Input`
- arguments: `String`

The **`lookup`** method resolves the file name argument and returns an input stream for that file.

Resolver:valid-p

- return: `Boolean`
- arguments: `String`

The **`valid-p`** predicate returns true if the file name argument can be resolved. If the file name can be resolved, the `lookup` method can be called to get an input stream.

Bibliography

[1] Revised Report on the Algorithmic Language Scheme. Technical report, November 1991.

[2] *C++ Language Reference Manual*, 1996.

[3] Guy L. Steele Jr. *Common Lisp, The Language.* 1990.

[4] Donald E. Knuth. *The Art of Computer Programming, Volume 1.* 1997.

[5] Donald E. Knuth. *The Art of Computer Programming, Volume 2.* 1997.

[6] Donald E. Knuth. *The Art of Computer Programming, Volume 3.* 1997.

[7] George Springer and Daniel P. Friedman. *Scheme and the Art of Programming.* 1997.

[8] Bjarne Stroustrup. *The C++ Programming Language.* 2000.

Index