

Volume 1 / Writer's Reference

AFNIX Writing System

Revision 4.0

Amaury C. Darsch

Visit <http://www.afnix.org>

Contents

Chapter 1. Getting started	1
1. First contact	1
2. Writing structure	4
3. Nameset and bindings	7
4. Special forms	9
5. Built-in objects	12
6. Class and instance	13
7. Miscellaneous features	14
8. Threads	16
9. The interpreter object	18
Chapter 2. Literals	19
1. Integer number	19
2. Relatif number	21
3. Real number	22
4. Complex number	23
5. Character	24
6. String	26
7. Regular expression	27
Chapter 3. Container objects	31
1. Cons object	31
2. List object	32
3. Vector object	32
4. Set object	32
5. Iteration	33
6. Special Objects	34
Chapter 4. Classes	35
1. Class object	35
2. Instance	36
3. Inheritance	39
Chapter 5. Advanced concepts	43
1. Exception	43
2. Nameset	44
3. Delayed Evaluation	45
4. Enumeration	45
5. Logger	46
6. Interpreter	46
7. Librarian object	48
8. Librarian object	49
9. File resolver	50

10. Thread operations	50
11. Shared objects	51
12. Synchronization	52
13. Function expression	55
Appendix A. Installation Guide	59
1. Software distribution	59
2. Installation procedure	59
3. Running AFNIX	62
4. Special features	62
Appendix B. Maintainer notes	65
1. The distribution tree	65
2. Configuration and setup	65
3. Compilation	66
4. Building the package	67
5. Specific makefile rules	67
Appendix C. Release notes	69
1. Release 4.0	69
2. Release 3.9	69
3. Release 3.8	69
4. Release 3.7	69
5. Release 3.6	70
6. Release 3.5	70
7. Release 3.4	71
8. Release 3.3	71
9. Release 3.2	72
10. Release 3.1	72
11. Release 2.9	73
12. Release 2.8	73
13. Release 2.7	74
14. Release 2.6	74
15. Release 2.5	75
16. Release 2.4	75
17. Release 2.3	76
18. Release 2.2	77
19. Release 2.1	77
20. Release 2.0	78
21. Release 1.9	79
22. Release 1.8	80
23. Release 1.7	80
24. Release 1.6	81
25. Release 1.5	81
26. Release 1.4	82
27. Release 1.3	83
28. Release 1.2	84
29. Release 1.1	84
30. Release 1.0	85
Index	87

Getting started

AFNIX is a multi-threaded functional engine with dynamic symbol bindings that supports the object oriented paradigm. The system features a state of the art runtime engine that runs on both 32 and 64 bits platforms. The system specification offers a rich syntax that makes the functional programming a pleasant activity. When the interpreter is used interactively, *text* is 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 or operates with an input stream.

AFNIX is a comprehensive set of application clients, modules and services. The original distribution contains the core interpreter with additional clients like the compiler, the librarian and the debugger. The distribution contains also a rich set of modules that are dedicated to a particular domain. The basic modules are the standard i/o module, the system module and the networking module. Above modules are services. A service is another extension of the engine that provides extra functionalities with help of several modules. This hierarchy is strictly enforced in the system design and provides a clear functional separation between application domain. When looking for a particular feature, it is always a good idea to think in term of module or service functionality.

AFNIX operates with a set of keywords and predicates. The engine has a native Unicode database. The set of standard objects provides support for integers, real numbers, strings, characters and boolean. Various containers like list, vector, hash table, bitset, and graphs are also available in the core distribution. The syntax incorporates the concept of lambda expression with explicit closure. Symbol scope limitation within a lambda expression is a feature called gamma expression. Form like notation with an easy block declaration is also another extension with respect to other system. The object model provides a single inheritance mechanism with dynamic symbol resolution. Special features include instance parenting, class binding instance inference and deference. Native class derivation and method override is also part of the object model with fixed class objects and forms. The engine incorporates an original regular expression engine with group matching, exact or partial match and substitution. An advanced exception engine is also provided with native run-time compatibility.

AFNIX implements a true multi-threaded engine with an automatic object protection mechanism against concurrent access. A read and write locking system which operates with the thread engine is also built in the core system. The object memory management is automatic inside the core interpreter. Finally, the 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 programming interface has been designed to ease the integration of foreign libraries.

1. First contact

The fundamental syntactic object is a *form*. A form is parsed and immediately executed by the interpreter. A form is generally constructed with a function name and a set of arguments. The process of executing a form is called the *evaluation*. The next example

illustrates one of the simplest form which is supported by the engine. This form simply displays the message *hello world* .

1.1. Hello world. At the interpreter prompt, a form is constructed with the special object *println* . The unique argument is a string which is placed between double quotes.

```
1 (axi) println "Hello World"
2 Hello World
```

The interpreter can be invoked to enter one or several forms interactively. The form can also be placed in a text file and the interpreter called to execute it. The `als` is the referred extension for a text file, but it can be anything. A simple session which executes the above file – assuming the original file is called `hello.als` – is shown below.

```
1 zsh> axi hello.als
2 Hello World
```

In interactive mode, the interpreter waits for a form. When a form is successfully constructed, it is then immediately executed by the engine. Upon completion, the interpreter prompt is displayed and the interpreter is ready to accept a new form. A session is terminated by typing `ctrl-d` . Another way to use the engine is to call the compiler client called *axc* , and then invoke the interpreter with the compiled file. The interpreter assumes the `.axc` extension for compiled file and will automatically figure out which file to execute when a file name is given without an extension.

```
1 zsh> axc hello.als
2 zsh> axi hello.axc
3 Hello World
4 zsh> axi hello
5 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.

1.2. Interpreter command. The interpreter can be invoked with several options, a file to execute and some program arguments. The `[h]` option prints the various interpreter options.

```
1 zsh> axi -h
2 usage: axi [options] [file] [arguments]
3 [h] print this help message
4 [v] print system version
5 [m] enable the start module
6 [i path] add a resolver path
7 [e mode] force the encoding mode
8 [f assert] enable assertion checks
9 [f nopath] do not set initial path
10 [f noseed] do not seed random engine
11 [f seed] seed random engine
```

The `[v]` option prints the interpreter version and operating system. The `[f]` option turns on or off some additional options like the assertion checking. The use of program arguments is illustrated later in this chapter. The `[i]` option adds a path to the interpreter resolver. Several `[i]` options can be specified. The order of search is determined by the option order. As mentioned earlier, the use of the resolver combined with the *librarian* is described in a specific chapter. If the initial file name to execute contains a directory path, such path is added automatically to the interpreter resolver path unless the `[nopath]` option is specified.

Binding	Description
backspace	Erase the previous character
delete	Erase at the cursor position
insert	Toggle insert with in-place
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

Binding	Description
left	Move the cursor to the left
right	Move the cursor to the right
up	Move up in the history list
down	Move down in the history list

1.3. Interactive line editing. Line editing capabilities is provided when the interpreter is used interactively. Error messages are displayed in red if the terminal supports colors. Various accelerators are bound to the terminal as indicated in the table below.

The arrow are also bound to their usual functions. Note that when using the history, a multi-line command editing access is provided by the interpreter.

1.4. Command line arguments. The interpreter command line arguments are stored in a vector called *argv* which is part of the *interp* object. A complete discussion about object and class is covered in the *class object chapter* . At this time, it is just necessary to note that a method is invoked by a name separated from the object symbol name with a semicolon. The example below illustrates the use of the vector argument.

```

1 # argv.als
2 # print the argument length and the first one
3 println "argument length: " (interp:argv:length)
4 println "first argument : " (interp:argv:get 0)
5 zsh> axi argv.als hello world
6 2
7 hello

```

1.5. Loading a source file. The interpreter object provides also the *load* method to load a 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 reads one form after another and executes them sequentially.

```

1 # load the source file demo.als
2 (axi) interp:load "demo.als"
3 # load the compiled file demo.axc
4 (axi) interp:load "demo.axc"
5 # load whatever is found
6 (axi) interp:load "demo"

```

The *load* method operates with the help of the interpreter resolver. By default the source file extension is *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. Without extension, the compiled file is searched first. If it is not found the source file is searched and loaded.

1.6. The compiler. The client `axc` is the *cross compiler* . It generates a binary file that can be run across several platforms. The `[h]` option prints the compiler options.

```

1 usage: axc [options] [files]
2 [h] print this help message
3 [v] print version information
4 [i] path add a path to the resolver
5 [e mode] force the encoding mode

```

One or several files can be specified on the command line. The source file is searched with the help of the resolver. The resolver `[i]` option can be used to add a path to the resolver.

2. Writing structure

The structure of file is a succession of valid syntactic objects separated by blank lines or comments. During the compilation or the execution process, each syntactic object is processed one after another in a single pass. Reserved keywords are an integral part of the writing systems. The association of symbols and literal constitutes a *form* . A form is the basic execution block in the writing system. When the form uses reserved keyword, it is customary to refer to it as a *special form* .

2.1. Character set and comments. The writing system operates with the standard Unicode character set. Comments starts with the character `#` . All characters are consumed until the end of line. Comments can be placed anywhere in the source file. Comments entered during an interactive session are discarded.

2.2. Native objects. The writing system operates mostly with objects. An object is created upon request or automatically by the engine when a native representation is required. To perform this task, several native objects, namely *Boolean* for boolean objects, *Integer* , *Relatif* for integer numbers, *Real* for floating-point number, *Complex* for complex number, *Byte* , *Character* and *String* for character or string manipulation are built inside the engine. Most of the time, a native object is built implicitly from its lexical representation, but an explicit representation can also be used.

```

1 const boolean true
2 const integer 1999
3 const relatif 1234567890R
4 const complex 1.0+1.0i
5 const real 2000.0
6 const string "afnix"
7 const char 'a'
8 trans symbol "hello world"
9 trans symbol 2000

```

The *const* and *trans* reserve 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. The *const* reserved keyword creates a *constant 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 transient symbol. When a symbol is marked transient, the object bound to the symbol can be changed while this is not possible with a constant symbol. Eventually, a symbol can be destroyed with the special form *unref* . It is worth to note that it is the symbol which is destroyed and not the object associated with it.

2.3. Stop and resume parsing. The parsing process is stopped in the presence of the ◀ character (Unicode U+25C0). The parsing operation is resumed with the ▶ character (Unicode U+25B6). Such mechanism is useful when dealing with multi line statements. This mechanism is also a good example of Unicode based control characters.

2.4. Forms. 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. 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 with complex expressions.

```
1 const three (+ 1 2)
```

This example defines a symbol which is initialized with the integer 3, that is the result of the computation (+ 1 2). The example shows also that a Polish notation is used for arithmetic. In fact, + is a built-in operator which causes the arguments to be summed (if possible). Evaluation can be nested as well as definition and assignment. 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, the result is discarded.

```
1 const b (trans a (+ 1 2))
2 assert a 3
3 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 which is the result of the previous evaluation. Finally, the form (const b 3) is evaluated, that is, the symbol 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.

2.5. Lambda expression. A *lambda expression* is another name for a function. The term comes 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 keyword. A lambda expression takes 0 or more arguments and returns 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. The function is then called and the object result is transmitted to the calling form. The use of *trans* vs *const* is explained later. To illustrate the use of a lambda expression, the computation of an integer factorial is described in the next example.

```
1 # declare the factorial function
2 trans fact (n) (
3   if (== n 1) 1 (* n (fact (- n 1))))
4 # compute factorial 5
5 println "factorial 5 = " (fact 5)
```

This example 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* special form and can be easily understood. The function is called in the next form when the *println* special form is executed. Note that here, the call to *fact* produces an integer object, which is converted automatically by the *println* keyword.

2.6. Block form. The notation used in the *fact* program is the standard form notation originating from Lisp and the Scheme dialect. There is also 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.

```

1 # declare the factorial procedure
2 trans fact (n) {
3   if (== n 1) 1 (* n (fact (- n 1)))
4 }
5 # compute factorial 5
6 println "factorial 5 = " (fact 5)

```

Another way to create a lambda expression is via the *lambda* special form. 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.

```

1 # declare the factorial procedure
2 const fact (lambda (n) (
3   if (== n 1) 1 (* n (fact (- n 1)))))
4 # compute factorial 5
5 println "factorial 5 = " (fact 5)

```

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

2.7. 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. There is also 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.

```

1 # declare the factorial procedure
2 const fact (n) (
3   if (== n 1) 1 (* n (fact (- n 1))))
4 # compute factorial 5
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 design 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 this book.

2.8. Lambda generation. A lambda expression can be used to generate another lambda expression. In other word, a function can generate a function, an that capability is an essential ingredient of the *functional programming* paradigm. The interesting part with lambda expression is the concept of closed variables. In the next example, 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 it is the object created by the lambda or the gamma call which is called a *closure* .

```

1 # a gamma which creates a lambda
2 const gen (n) (
3   lambda (x) (n) (+ x n))
4 # create a function which add 2 to its argument
5 const add-2 (gen 2)

```

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

In short, a lambda expression is a function with or without closed variables, which works with nested symbol sets also called *namesets* . A gamma expression is a function with or without closed variable which is bounded 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.

2.9. 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* .

```
1 const proc-nilp (args) {
2   trans result 0
3   for (i) (args) (result:+= i)
4   eval result
5 }
6 assert 3 (proc-nilp 1 2)
7 assert 7 (proc-nilp 1 2 4)
```

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

```
1 # check with arguments
2 const proc-args (a b args) {
3   trans result (+ a b)
4   for (i) (args) (result:+= i)
5   eval result
6 }
7 assert 3 (proc-args 1 2)
8 assert 7 (proc-args 1 2 4)
```

It is an error to specify formal arguments after *args* . Multiple *args* formal definition are not allowed. The symbol *args* can also be defined as a constant argument.

```
1 # check with arguments
2 const proc-args (a b (const args)) {
3   trans result (+ a b)
4   for (i) (args) (result:+= i)
5   eval result
6 }
7 assert 7 (proc-args 1 2 4)
```

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 most common one 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, there is 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 look-up is done by traversing all nameset from

the bottom to the top and stopping when one is found. In term of 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 `'...'`.

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*.

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 bounded to a symbol. Here is a nameset called *example* in the top level nameset.

```
1 # create a new nameset called example
2 const example (nameset .)
3 # bind a symbol in this nameset
4 const example:hello "hello"
5 println example:hello
```

3.3. Qualified name. In the previous example, a symbol is referenced in a given nameset by using a *qualified name* such like *example:hello*. A qualified name defines a path to access a symbol. The use of a 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.

3.4. Symbol binding. The *trans* reserved keyword has been shown in all previous example. The reserved keyword *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 bound in this nameset. If *example* is a class, *hello* is bounded as a class symbol. 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.

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

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 constant property is held by the symbol itself. When trying to set a constant symbol, an exception is raised. The reserved keyword *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 gamma expression. Like *trans* the rule are the same except that the symbol is marked constant.

```
1 # create automatically a gamma expression
2 const max (x y) (if (> x y) x y)
```

3.6. Symbol unreferencing. The *unref* reserved keyword removes a symbol reference in a given context. When the context is a nameset, the object associated with the symbol is detached from the symbol, eventually destroyed with the symbol removed from the nameset.

```
1 # create a symbol number
2 const x 1
3 # unreference it
4 unref x
```

3.7. 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 which 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.

```
1 # create a gamma expression with const argument
2 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.

```
1 const proc-args (a b) {
2   trans result (+ a b)
3   for (i) (args) (result:+= i)
4   eval result
5 }
6 assert 3 (proc-args 1 2)
7 assert 7 (proc-args 1 2 4)
```

4. Special forms

Special forms provides are reserved keywords which are most of the time imperative statement, as part of the writing system. Special forms are an integral part of the writing system and interact directly with the interpreter. In most cases, a special forms returns the last evaluated object. Most of the special forms are control flow statements.

4.1. If special form. 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.

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

4.2. While special form. The *while* reserved keyword takes 2 or 3 arguments. With 2 arguments, the loop is constructed with a condition and a form. With 3 arguments, the first argument is an initial condition that is executed only once. When an argument acts as a loop condition, the condition evaluate to a boolean. The loop 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.

```
1 const gcd (u v) {
2   while (!= v 0) {
3     trans r (u:mod v)
4     u:= v
5     v:= r
6   }
7   eval u
8 }
```

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. In the presence of 3 arguments, the first argument is an initialization condition that is executed only once. In this mode, it is important to note that the loop introduce its own nameset. The loop condition can be used to initialize a local condition variable.

```
1 while (trans valid (is:valid-p)) (valid) {
2   # do something
3   # adjust condition
4   valid:= (and (is:valid-p) (something-else))
5 }
```

4.3. Do special form. 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 loop can accept either 2 or 3 arguments. With 2 arguments, the first argument is the loop body and the second argument is the exit loop condition. With 3 arguments, the first argument is the initial condition that is executed only once.

```
1 # count the number of digits in a string
2 const number-of-digits (s) {
3   const len (s:length)
4   trans index 0
5   trans count 0
6   do {
7     trans c (s:get index)
8     if (c:digit-p) (count:++)
9   } (< (index:++) len)
10  eval count
11 }
```

4.4. Loop special form. The *loop* reserved keyword is another form of loop. It takes 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* special form creates its own namespace, since the initialize condition generally creates new symbols for the loop only.

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

A loop can also be designed with a *Counter* object. In this case, a counter is created with an initial and final count values. The counter *step-p* method can then be used to run the loop

```
1 # a counter from 0 to 10
2 trans cnt (Counter 10)
3 # a simple loop from 1 to 10
4 loop (cnt:step-p) (println cnt)
```

In this example, the counter prints from 1 to 10 since the counter is designed to operate from 0 to 9, and the *println* function is called after the *step-p* predicate.

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

```
1 # return the primary color in a rgb
2 const get-primary-color (color value) (
3   switch color (
4     ("red" (return (value:substr 0 2)))
5     ("green" (return (value:substr 2 4)))
6     ("blue" (return (value:substr 4 6)))
7   ))
```

4.6. Return special form. 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.

```
1 # initialize a vector with a value
2 const vector-init (length value) {
3   # treat nil vector first
4   if (<= length 0) return (Vector)
5   trans result (Vector)
6   do (result:add value) (> (length:-- 0)
7 }
```

4.7. Eval and protect. The *eval* reserved keyword forces the evaluation of the object argument. The reserved keyword *eval* is typically used in a function body to return a particular symbol value. It 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 object without evaluating it. Typically when used with a form, *protect* returns the form itself. It can also be used to prevent a symbol evaluation. When used with a symbol, the symbol object itself is returned.

```
1 const add (protect (+ 1 2))
2 (eval add)
```

Note that in the preceding example that the evaluation will return a lambda expression which is evaluated immediately and which returns the integer 3.

4.8. Assert special form. 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.

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

4.9. Block special form. 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.

```
1 trans a 1
2 block {
3   assert a 1
4   trans a (+ 1 1)
5   assert a 2
6   assert ...:a 1
7 }
8 assert 1 a
```

5. Built-in objects

Several built-in objects and built-in operators for arithmetic and logical operations are also integrated in the writing system. The *Integer* and *Real* classes are primarily used to manipulate numbers. The *Boolean* class is used to for boolean operations. Other built-in objects include *Character* and *String*. The exact usage of these objects is described in the next chapter.

5.1. Arithmetic operations. Support for the arithmetic operations is provided with the standard operator notation. Normally, these operators will tolerate various object type mixing and the returned value will generally be bound to an object that provides the minimum loss of information. Most of the operations are done with the $+$, $-$, $*$ and $/$ operators.

```
1 (+ 1 2)
2 (- 1)
3 (* 3 5.0)
4 (/ 4.0 2)
```

5.2. Logical operations. The *Boolean* class is used to represent the boolean value *true* and *false*. These last two symbols are built-in in the interpreter as constant symbols. There are also special forms like *not*, *and* and *or*. Their usage is self understandable.

```
1 not true
2 and true (== 1 0)
3 or (< -1 0) (> 1 0)
```

Predicate	Description
<code>nil-p</code>	check nil object
<code>eval-p</code>	check evaluation
<code>real-p</code>	check real object
<code>regex-p</code>	check regex object
<code>object-p</code>	check for non nil object
<code>string-p</code>	check string object
<code>number-p</code>	check number object
<code>method-p</code>	check method object
<code>boolean-p</code>	check boolean object
<code>integer-p</code>	check integer object
<code>character-p</code>	check character object

Predicate	Description
<code>class-p</code>	check class object
<code>thread-p</code>	check thread object
<code>promise-p</code>	check promise object
<code>lexical-p</code>	check lexical object
<code>literal-p</code>	check literal object
<code>closure-p</code>	check closure object
<code>nameset-p</code>	check nameset object
<code>instance-p</code>	check instance object
<code>qualified-p</code>	check qualified object

5.3. Predicates. A *predicate* is a function which returns a boolean object. There is always a built-in predicate associated with a built-in object. 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. The *object-p* predicate is the negation of the *nil-p* predicate.

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

```

1 # return true if the argument is a number
2 const number-p (n) (
3   or (integer-p n) (real-p n))

```

Special predicates for functional and symbolic programming are also built-in in the engine.

Finally, for each object, a predicate is also associated. For example, *cons-p* is the predicate for the *Cons* object and *vector-p* is the predicate for the *Vector* object. Another issue related to evaluation, is to decide whether or not an object can be evaluated. The predicate *eval-p* which is a special form is designed to answer this question. Furthermore, the *eval-p* predicate is useful to decide whether or not a symbol is defined or if a qualified name can be evaluated.

```

1 assert true (eval-p .)
2 assert false (eval-p an-unknown-symbol)

```

6. Class and instance

Classes and instances are the fundamental objects that provide support for the object oriented paradigm. A *class* is a nameset which can be bounded automatically when an *instance* of that class is created. The class model is sloppy. Compared to other systems, there is no need to declare the data members for a particular class. Data members are created

during the instance construction. An instance can also be created without any reference to a class. Methods can be bound to the class or the instance or both. An instance can also be muted during the execution process.

6.1. Class and members. A class is declared with the reserved keyword *class* . The resulting object acts like a nameset and it is possible to bind symbol to it.

```

1 # create a class object
2 const Circle (class)
3 const Circle:PI 3.1415926535
4 # access by qualified name
5 println Circle:PI

```

In the previous example, the symbol *Circle* is created as a class object. With the help of a qualified name, the symbol *PI* is created inside the class nameset. In this case, the symbol *PI* is invariant with respect to the instance object. A form can also be bound to the class nameset. In both cases, the symbol or the form is accessed with the help of a qualified name.

6.2. Instances. An instance of a class is created like any built-in object. If a method called *preset* is defined for that class, the method is used to initialize the instance.

```

1 # create a class
2 const Circle (class)
3 trans Circle:preset (r) {
4   const this:radius (r:clone)
5 }
6 # create a radius 1 circle
7 const c (Circle 1)

```

This example calls for several comments. First the *preset* lambda expression is bound to the class. Since *preset* is a reserved name for the class object, the form is automatically executed at the instance construction. Second, note that the instance data member *radius* is created by the lambda expression and another reserved keyword called *this* is used to reference the instance object as it is customary with other programming systems.

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. Therefore, the use of the reserved keyword *this* is not recommended in a gamma expression, although it is perfectly acceptable to create a symbol with such name.

```

1 # create a perimeter method
2 trans Circle:perimeter nil (
3   * (* 2.0 Circle:PI) this:radius)
4 # call the method with our circle
5 trans p (c:perimeter)

```

It must be clear that the *perimeter* symbol defines a method at the class level. It is perfectly acceptable to define a methods at the instance level. Such method is called a *specialized method* .

7. Miscellaneous features

7.1. Iteration. An iteration facility is provided for some objects known as *iterable objects* . The *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.

```

1 # compute the scalar product of two vectors
2 const scalar-product (u v) {
3   trans result 0
4   for (x y) (u v) (result:+= (* x y))
5   eval result
6 }

```

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.

```

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

```

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

```

1 # scalar product with explicit iterators
2 const scalar-product (u v) {
3   trans result 0
4   trans u-it (u:iterator)
5   trans v-it (v:iterator)
6   while (u:valid-p) {
7     trans x (u:get-object)
8     trans y (v:get-object)
9     result:+= (* x y)
10    u:next
11    v:next
12  }
13  eval result
14 }

```

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.

7.2. Exception. An *exception* is an unexpected change in the execution flow. The exception model 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 implementation. An exception is thrown with the special form *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 special form *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.

```

1 try (+ 1 2)
2 try (throw)
3 try (throw "hello")
4 try (throw "hello" "world")
5 try (throw "hello" "world" "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.

```

1 # protected factorial
2 const fact (n) {
3   if (not (integer-p n))
4     (throw "number-error" "invalid argument")

```

```

5   if (== n 0) 1 (* n (fact (- n 1)))
6   }
7   # exception handler
8   const handler nil {
9     error!n what:eid ',' what:reason
10  }
11  (try (fact 5) handler)
12  (try (fact "hello") handler)

```

The special symbol *what* stores the necessary information about the place that generated the exception. Most of the time, the qualified name *what:reason* or *what:about* is used. The only difference is that *what:about* contains the file name and line number associated with the reason that generated the exception.

7.3. Regular Expressions. A regular expression or *regex* is an object which is used to match certain text patterns. Regular expressions are built implicitly by the parser with the use of the `[` and `]` characters. Special class of characters are defined with the help of the `$` character. For example, `$d` is the class of character digits as defined by the Unicode consortium. Different regular expression can be grouped by region to be matched as indicated in the example below.

```

1  if (== (const re [($d$d):($d$d)]) "12:31") {
2    trans hr (re:get 0)
3    trans mn (re:get 1)
4  }

```

In the previous example, a regular expression object is bound to the symbol *re*. The *regex* contains two groups. 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.

7.4. Delayed evaluation. The special form *delay* creates a special object called a *promise* which records the form to be later evaluated. The special form *force* causes a promise to be evaluated. Subsequent call with *force* will produce the same result.

```

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

```

8. Threads

The interpreter provides a powerful mechanism which allows the concurrent execution of forms and the synchronization of shared objects. The engine provides supports the creation and the synchronization of threads with a native object locking mechanism. During the execution, the interpreter wait until all threads are completed. A threads is created with the reserved keyword *launch*. In the presence of several threads, the interpreter manages automatically the shared objects and protect them against concurrent access.

```

1  # shared variable access
2  const var 0
3  const decr nil (while true (var:= (- var 1)))
4  const incr nil (while true (var:= (+ var 1)))
5  const prtv nil (while true (println "value = " var))
6  # start 3 threads
7  launch (prtv)
8  launch (decr)
9  launch (incr)

```

8.1. Form synchronization. Although, the engine 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 engine guarantees that only one thread will execute this form.

```

1  const print-message (code mesg) (
2    sync {
3      errorln "error : " code
4      errorln "message: " mesg
5    }
6  )

```

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.

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.

```

1  # simple flag
2  const flag false
3  # simple shared tester
4  const ftest (bval) (flag:= bval)
5  # run the thread and wait
6  const thr (launch (ftest true))
7  thr:wait
8  assert true flag

```

This example is taken from the test suites. It checks that a boolean variable is set in a thread. Note the use of the *wait* method to make sure the thread has completed before checking for the flag value. 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.

```

1  # shared vector of threads descriptors
2  const thr-group (Vector)
3  # wait until all threads in the group are finished
4  const wait-all nil (for (thr) (thr-group) (thr:wait))

```

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.

8.4. Asynchronous evaluation. The special form *future* creates a special object called a *future* which is used to evaluate an object asynchronously. The evaluation starts with the help of the *force* special form. The *sync* special form can be used to synchronise with the future.

```

1  trans f (future 1)
2  force f

```

9. The interpreter object

The interpreter can also be seen as an object. As such, it provides several special symbols and forms. For example, the symbol *argv* is the argument vector. The symbol *library* is an interpreter method that loads a library. A complete description of the interpreter object is made in a special chapter of this book.

Literals

This chapters covers in detail the literals objects used to manipulate numbers and strings. First the integer, relatif, real and complex numbers are described. There is a broad range of methods for these three objects that 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* , which are objects that have a string representation. A special literal object known as *regular expression* or *regex* is also described at the end of this chapter.

1. Integer number

The fundamental number representation is the *Integer* . The integer is a 64 bits signed 2's complement number. Even when running with a 32 bits machine, the 64 bits representation is used. If a larger representation is needed, the *Relatif* object might be more appropriate. The *Integer* object is a literal object that belongs to the number class.

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.

```
1  const a 123
2  trans b -255
3  const h 0xff
4  const b 0b1111_1111
```

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.

```
1  const a (Integer)
2  const b (Integer 2000)
3  const c (Integer "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 exhibits also the strange property to be equal to its negation since with 2's complement, there is no positive representation.

1.2. Integer arithmetic. Standard arithmetic operators are available as built-in operators. The usual addition *+* , multiplication *** and division */* operate with two arguments. The subtraction *-* operates with one or two arguments.

```
1  + 3 4
2  - 3 4
3  - 3
4  * 3 4
5  / 4 2
```

As a built-in object, the *Integer* object offers various methods for built-in arithmetic which directly operates on the object. The following example illustrates these methods.

```

1 trans i 0
2 i:++
3 i:--
4 i:+ 4
5 i:= 4
6 i:- 1
7 i:* 2
8 i:/ 2
9 i:+= 1
10 i:-= 1
11 i:*= 2
12 i:/= 2

```

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* .

```

1 const i 0
2 i:= 1

```

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.

```

1 == 0 1
2 != 0 1

```

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

```

1 i:= 1
2 i:== 1
3 i:!= 0

```

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 introduction chapter. Such example was terribly slow, because of the double recursion. Another method suggested by Springer and Friedman uses two functions to perform the same job.

```

1 const fib-it (gamma (n acc1 acc2) (
2   if (== n 1) acc2 (fib-it (- n 1) acc2 (+ acc1 acc2))))
3 const fiboi (gamma (n) (
4   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, it is still an elegant way to write complex functions.

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 *abs* methods returns the absolute value of the calling integer.

```

1 i:even-p
2 i:odd-p
3 i:mod 2
4 i:=- 1
5 i:abs
6 i:to-string

```

The *Integer* object is a *literal object* and a *number object*. As a literal object, the *to-string* and *to-literal* methods are provided to obtain a string representation for the integer object. Although the *to-string* method returns a string representation of the calling integer, the *to-literal* method returns a parsable string. Strictly speaking for an integer, there is no difference between a string representation and a literal representation. However, this is not true for other objects.

```

1 (axi) const i 0x123
2 (axi) println (i:to-string)
3 291
4 (axi) println (i:to-literal)
5 291

```

As a number object, the integer number can also be represented in hexadecimal format. The *to-hexa* and *to-hexa-string* methods are designed to obtain such representation. In the first form, the *to-hexa* method return a literal hexadecimal string representation with the appropriate prefix while the second one does not.

```

1 (axi) const i 0x123
2 (axi) println (i:to-hexa)
3 0x123
4 (axi) println (i:to-hexa-string)
5 123

```

2. Relatif number

A *relatif* or big number 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* suffix to express a relatif number versus an integer one. The *Relatif* object is a literal object that belongs to the number class. The predicate associated with the *Relatif* object is *relatif-p*.

```

1 const a 123R
2 trans b -255R
3 const c 0xffR
4 const d 0b1111_1111R
5 const e (Relatif)
6 const f (Relatif 2000)
7 const g (Relatif "23")

```

2.1. Relatif operations. Most of the *Integer* class 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 known prime Mersenne number. The world record exponent is 6972593. The number is therefore:

```

1 const i 1R
2 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...

3. Real number

The *real* class implements the 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. The *Real* object is a literal object that belongs to the number class.

3.1. Real format. The parser supports two types of literal representation for real number. The first representation is the *dotted decimal* notation. The second notation is the *scientific notation* .

```
1 const a 123.0 # a positive real
2 const b -255.5 # a negative real
3 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.

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. Ultimately, a pure integer operation might generate a real result.

```
1 + 1999.0 1 # 2000.0
2 + 1999.0 1.0 # 2000.0
3 - 2000.0 1 # 1999.0
4 - 2000.0 1.0 # 1999.0
5 * 1000 2.0 # 2000.0
6 * 1000.0 2.0 # 2000.0
7 / 2000.0 2 # 1000.0
8 / 2000.0 2.0 # 1000.0
```

Like the *Integer* object, the *Real* object has arithmetic built-in methods.

```
1 trans r 0.0 # 0.0
2 r:++ # 1.0
3 r:-- # 0.0
4 r:+ 4.0 # 4.0
5 r:= 4.0 # 4.0
6 r:- 1.0 # 3.0
7 r:* 2.0 # 8.0
8 r:/ 2.0 # 2.0
9 r:+= 1.0 # 5.0
10 r:-= 1.0 # 4.0
11 r:*= 2.0 # 8.0
12 r:/= 2.0 # 4.0
```

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.

```
1 == 2000 2000 # true
2 != 2000 1999 # true
```

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

```
1 r:= 1.0 # 1.0
2 r== 1.0 # true
3 r!= 0.0 # true
```

4. Complex number

A *complex* number is another type of number used to solve complex operations that are not possible with real number. A complex number is made of a real part and an imaginary part. When the real part is omitted, one has a pure imaginary number. The standard notation is to write the imaginary part with the letter *i* giving a literal notation that resembles an arithmetic operation.

```

1 trans z 1.0+1.0i
2 trans z (Complex 1.0 1.0)
3 trans z "1.0i"

```

4.1. Complex operations. Most of the *real* class operations are supported by the *Complex* object including addition, subtraction and multiplication. A special case arises with the square root of negative number.

4.2. A complex example. One of the most interesting point with functional programming language is the ability to create complex function. For example let's assume we wish to compute the value at a point x of the *Legendre polynomial 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.

```

1 # legendre polynomial order 0 and 1
2 const lp-0 (gamma (x) 1)
3 const lp-1 (gamma (x) x)
4 # legendre polynomial of order n
5 const lp-n (gamma (n) (
6   if (> n 1) {
7     const lp-n-1 (lp-n (- n 1))
8     const lp-n-2 (lp-n (- n 2))
9     gamma (x) (n lp-n-1 lp-n-2)
10    (/ (- (* (* (- (* 2 n) 1) x)
11         (lp-n-1 x))
12        (* (- n 1) (lp-n-2 x)))) n)
13    } (if (== n 1) lp-1 lp-0)
14  ))
15 # generate order 2 polynomial
16 const lp-2 (lp-n 2)
17 # print lp-2 (2)
18 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 polynomial until it is constructed. Note also that we have here a double recursion.

4.3. 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 square root *sqr*t method return the square root of the calling real. The *floor* and *ceiling* returns respectively the floor and the ceiling of the calling real.

```

1 const r0 0.0 # 0.0
2 const r1 1.0 # 1.0
3 const r2 2.0 # 2.0
4 const rn -2.0 # -2.0
5 const rq (r2:sqrt) # 1.414213
6 const pi 3.1415926 # 3.141592
7 rq:floor # 1.0
8 rq:ceiling # 2.0
9 rn:abs # 2.0
10 r1:log # 0.0

```

```

11 r0:exp # 1.0
12 r0:sin # 0.0
13 r0:cos # 1.0
14 r0:tan # 0.0
15 r0:asin # 0.0
16 pi:floor # 3.0
17 pi:ceiling # 4.0

```

4.4. Accuracy and formatting. 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-absolute-precision* method while the *get-absolute-precision* returns the interpreter precision. There is also a *set-relative-precision* and *get-relative-precision* methods used for the definition of relative precision. By default, the absolute precision is set to 0.00001 and the relative precision is set to 1.0E-8.

```

1 interp:set-absolute-precision 0.0001
2 const r 2.0
3 const s (r:sqrt) # 1.4142135
4 (s:?= 1.4142) # true

```

Real number formatting 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.

```

1 const pi 3.1415926535
2 pi:format 3 # 3.142

```

If additional formatting is needed, the *String fill-left* and *fill-right* methods can be used.

```

1 const pi 3.1415926535 # 3.1415926535
2 const val (pi:format 4) # 3.1416
3 println (val:fill-left '0' 9) # 0003.1416

```

4.5. Number object. The *Integer*, *Relatif* and *Real* objects are all derived from the *Number* object which is a *Literal* object. As such, the predicate *number-p* is the right mechanism to test an object for a number. The class also provides the basic mechanism to format the number as a string. For integer and relatif, the hexadecimal representation can be obtained by the *to-hexa* and *to-hexa-string* methods. For integer and real numbers, the *format* method adjusts the final representation with the precision argument as indicated before. It is worth to note that a formatted integer gets automatically converted into a real representation.

5. Character

The *Character* object is another built-in object. A character is internally represented by a quad by using a 31 bit representation as specified by the Unicode standard and ISO 10646.

5.1. Character format. The standard quote notation is used to represent a character. In that respect, there is here a substantial difference with other functional language where the quote protect a form.

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

All characters from the *Unicode codeset* are supported by the **AFNIX** engine. The 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 quoted character or the literal notation based on the *U+* notation in hexadecimal. For example, *U+40* is the @ character while *U+3A3* is the sigma capital letter Σ .

```
1 const nilc (Character) # null character
2 const a (Character 'a') # a
3 const 0 (Character 48) # 0
4 const mul (Character "*") # *
5 const div (Character "U+40") # @
```

5.2. Character arithmetic. A character is like an integer, except that it operates in the range 0 to 0x7FFFFFFF. 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.

```
1 + 'a' 1 # 'b'
2 - '9' 1 # '8'
```

Several *Character* object methods are also provided for arithmetic operations in a way similar to the *Integer* class.

```
1 trans c 'a' # 'a'
2 c:++ # 'b'
3 trans c '9' # '9'
4 c:-- # '8'
5 c:+ 1 # '9'
6 c:- 9 # '0'
```

5.3. 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.

```
1 == 'a' 'b' # false
2 != '0' '1' # true
```

5.4. 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* , *eos-p* and *nil-p* .

```
1 const LA01 'a' # 'a'
2 const ND10 '0' # '0'
3 LA01:to-string # "a"
4 LA01:to-integer # 97
5 LA01:alpha-p # true
6 ND10:digit-p # true
```

6. String

The *String* object is one of the most important built-in object in the **AFNIX** engine. Internally, a string is a vector of *Unicode characters* . Because a string operates with Unicode characters, care should be taken when using composing characters.

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

```
1 const hello "hello"
```

Any literal object can be used to construct a string. This means that integer, real, boolean or character objects are all valid to construct strings. The default constructor creates a null string. The string constructor can also takes a string.

```
1 const nils (String) # ""
2 const one (String 1) # "1"
3 const a (String 'a') # "a"
4 const b (String true) # "true"
```

6.2. String operations. The *String* object provides numerous methods and operators. The most common ones are illustrated in the example below. The *length* methods returns the total number of characters in the string object. It is worth to note that this number is not necessarily the number of printed characters since some characters might be *combining characters* used, for example, as diacritics. The *non-combining-length* method might be more adapted to get the number of printable characters.

```
1 const h "hello"
2 h:length # 5
3 h:get 0 # 'h'
4 h:== "world" # false
5 h:!= "world" # true
6 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.

```
1 # example of sub-left method
2 const msg "hello world"
3 msg:sub-left 5 # "hello"
4 msg:sub-right 6 # "world"
```

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

```
1 # example of strip method
2 const str " hello world "
3 println (str:strip) # "hello world"
```

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.

```
1 # example of split method
2 const str "hello:world"
3 const vec (str:split ":") # "hello" "world"
4 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.

```

1 # example of fill-left method
2 const pi 3.1415926535 # 3.1415926535
3 const val (pi:format 4) # 3.1416
4 val:fill-left '0' 9 # 0003.1416

```

6.3. Conversion methods. The case conversion methods are the standard *to-upper* and *to-lower* methods. The method operates with the internal Unicode database. As a result, the conversion might change the string length. Other conversion methods related to the Unicode representation are also available. These are rather technical, but can be used to put the string in a normal form which might be suitable for comparison. Such conversion always uses the Unicode database normal form representation.

```

1 # example of case conversion
2 const str "hello world"
3 println (str:to-upper) # HELLO WORLD

```

6.4. String hash value. The *hashid* method is a method that computes the hash value of a string. The value depends on the target machine and will change between a 32 bits and a 64 bits machine. Example *example 0203.als* illustrates the computation of a hash value for our favorite test string.

```

1 # test our favorite string
2 const hello "hello world"
3 hello:hashid # 1054055120

```

The algorithm used by the engine is shown as an example below. 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. With a 64 bits machine the algorithm is slightly modified to use the extra space. This also means that the hashid value is not portable across platforms.

```

1 # compute string hashid
2 const hashid (s) {
3   const len (s:length)
4   trans cnt 0
5   trans val 0
6   trans sht 17
7   do {
8     # compute the hash value
9     trans i (Integer (s:get cnt))
10    val:= (val:xor (i:shl sht))
11    # adjust shift index
12    if (< (sht:-= 7) 0) (sht:+= 24)
13  } (< (cnt:++) len)
14  eval val
15 }

```

7. Regular expression

A regular expression or *regex* is a special literal object designed to describe a character string in a compact form with regular patterns. A regular expression provides a convenient way to perform pattern matching and field extraction within a character string.

Character	Description
\$a	matches any letter or digit
\$b	matches any blank characters
\$c	matches any combining alphanumeric
\$d	matches any digit
\$e	matches eol, cr and eos
\$l	matches any lower case letter
\$n	matches eol or cr
\$s	matches any letter
\$u	matches any upper case letter
\$v	matches any valid afnix constituent
\$w	matches any word constituent
\$x	matches any hexadecimal characters

7.1. Regex syntax. A regular expression is defined with 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 parser automatically recognizes a regex and built the object accordingly. The following example shows two equivalent methods for the same regex expression.

```

1 # syntax built-in regex
2 (== [d+] 2000) # true
3 # explicit built-in regex
4 (== (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.

7.2. 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 built-in in the regex engine.

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.

```

1 (== [d+] 2000) # true
2 (== ["d+"] 2000) # false

```

Combining alphanumerical characters can generate surprising result when used with Unicode string. Combining alphanumeric characters are alphanumeric characters and non spacing combining mark as defined by the Unicode consortium. In practice, the combining marks are the diacritics used with regular letter, such like the accents found in the western languages. Because the writing system uses a canonical decomposition for representing the Unicode string, it turns out that the printed string is generally represented with more bytes, making the string length longer than it appears.

7.3. 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.

Operator	Description
*	match 0 or more times
+	match 1 or more times
?	match 0 or 1 time
—	alternation

If the first character is the `^` character in the character set, the character set is complemented with regards to its definition.

7.4. 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.

```

1 # simple real number check
2 const real-1 (Regex "$d*.$d+")
3 # another way with [] characters
4 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.

```

1 # pair of digit testing
2 (== [d$d[d$d$]+] 2000) # true
3 (== [d$d[d$d$]+] 20000) # false

```

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

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.

7.5. 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):(ddd$d)]`. This regex assumes a date in the form `mm:dd:yyyy`.

```

1 if (== (const re [(d$d):(d$d)]) "12:31") {
2   trans hr (re:get 0)
3   trans mn (re:get 1)
4 }

```

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

7.6. 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. 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.

```

1 const re (Regex "$d+")
2 println re # $d+
3 println re:to-string # [d+]

```

7.7. 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 sub-string or the string argument.

7.8. 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.

```

1 if (== (const re [($d$d):($d$d)]) "12:31") {
2   re:length # 2
3   re:get 0 # 12
4   re:get 1 # 31
5 }
```

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

```

1 const regex [\$d+]
2 regex:match "Happy new year 2000" # 2000
```

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

```

1 const regex [\$d+]
2 regex:replace "Hello year 2000" "3000" # hello year 3000
```

7.9. 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.

Container objects

This chapter covers the standard container objects and more specifically, *iterable* objects such like *Cons* , *List* and *Vector* . Special objects like *Fifo* , *Queue* and *Bitset* are treated at the end of this chapter. Although the name container is sufficient enough to describe the object functionality, it is clear that a container is more than a simple object reservoir. In particular, the choice of a container object is often associated to the underlying algorithm used to store the object. For example, a vector is appropriate when storing by index is important. If the order of storage must be preserved, then a fifo object might be more appropriate. In any case, the choice of a container is always a question of compromise, so is the implementation.

1. Cons 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 similar in some respect to the *cons cell* found in traditional functional programming language. A *Cons* object 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 original Lisp notation is maintained here for the sake of tradition. Although a cons cell is the building block for single linked list, the cell itself is not a list object. When a list object is needed, the *List* double linked list object might be more appropriate.

1.1. Cons cell constructors. The default constructor creates a cons cell whose *car* is initialized to the nil object. The constructor can also take one or several objects.

```
1 const nil-cons (Cons)
2 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* . List can also be constructed directly by the parser. Since all internal forms are built with cons cell, the construction can be achieved by simply *protecting* the form from being interpreted.

```
1 const blist (protect ((1) ((2) ((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 list by index.

```
1 const c (Cons "hello" "world")
2 c:length # 2
3 c:get-car # "hello"
4 c:get-cadr # "world"
5 c:get 0 # "hello"
6 c:get 1 # "world"
```

The *set-car* method set the car of the cons cell. The *add* method adds a new cons cell at the end of the cons list and set the car with the specified object.

2. List object

The *List* 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.

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.

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

2.2. List methods. The *List* object methods are similar the *Cons* object. The *add* method adds an object at the end of the list. The *insert* method inserts an object at the beginning of the list.

```
1 const list (List "hello" "world")
2 list:length # 2
3 list:get 0 # "hello"
4 list:get 1 # "world"
5 list:add "folks" # "hello" "world" "folks"
```

3. Vector object

The *Vector* 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.

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

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

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

```
1 const vec (Vector "hello" "world")
2 vec:length # 2
3 vec:get 0 # "hello"
4 vec:get 1 # "world"
5 vec:add "folks" # "hello" "world" "folks"
6 vec:set 0 "bonjour" # "bonjour" "world" "folks"
```

4. Set object

The *Set* object provides the facility of an object container. The *Set* object is another example of *iterable object*. The *Set* object provides support for forward iteration. One of the property of a set is that there is only one object representation per set. Adding two times the same object results in one object only.

4.1. Set construction. A set is constructed like a vector. The default constructor creates a set with 0 objects.

```
1 const nil-set (Set)
2 const obj-set (Set 1 'a' "hello")
```

4.2. Set methods. The *Set* object methods are similar to the *Vector* object. The *add* method adds an object in the set. If the object is already in the set, the object is not added. The *length* method returns the number of elements in the set.

```

1  const set (Set "hello" "world")
2  set:get-size # 2
3  set:add "folks" # "hello" "world" "folks"

```

5. Iteration

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

5.1. 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* .

```

1  const map (obj func) {
2    trans result (Cons)
3    for (car) (obj) (result:link (func car))
4    eval result
5  }

```

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

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

```

1  # compute the scalar product of two vectors
2  const scalar-product (u v) {
3    trans result 0
4    for (x y) (u v) (result:+= (* x y))
5    eval result
6  }

```

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.

```

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

```

5.3. 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.

```

1  # convert an iterable object to a vector
2  const to-vector (obj) {
3    trans result (Vector)
4    for (i) (obj) (result:add i)
5    eval result
6  }

```

5.4. Explicit iterator. An explicit iterator is constructed with the *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.

```

1 # reverse a list
2 const reverse-list (obj) {
3   trans result (List)
4   trans itlist (obj:iterator)
5   itlist:end
6   while (itlist:valid-p) {
7     result:add (itlist:get-object))
8   itlist:prev
9 }
10 eval result
11 }
```

6. Special Objects

The engine incorporates other container objects. To name a few, such objects are the *Queue* , *Bitset* or *Fifo* objects.

6.1. 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. The *Fifo* and *Queue* objects are somehow similar, with the fundamental difference that the queue is a blocking.

```

1 # create a queue with objects
2 const q (Queue 2)
3 q:push "hello"
4 q:push "world"
5 q:empty-p # false
6 q:length # 2
7 # dequeue some object
8 q:pop # hello
9 q:pop # world
10 q:empty-p # true
```

6.2. Bitset object. A *bit set* is a special container for bit. A bit set can be constructed with a specific size. When the bit set is constructed, each bit can be marked and tested by index. Initially, the bitset size is null.

```

1 # create a bit set by size
2 const bs (Bitset 8)
3 bitset-p bs # true
4 # check, mark and clear
5 assert false (bs:marked-p 0)
6 bs:mark 0
7 assert true (bs:marked-p 0)
8 bs:clear 0
9 assert false (bs:marked-p 0)
```

CHAPTER 4

Classes

This chapter covers the class model and its associated operations. The class model is slightly different compared to traditional one because dynamic symbol bindings do not enforce to declare the class data members. A class is an 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.

1. Class object

A *class object* is simply a nameset which can be replicated via a construction mechanism. A class is created with the special form *class* . The result is an object of type *Class* which supports various symbol binding operations.

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

```
1 const Color (class)
```

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

```
1 const Color (class)
2 const Color:RED-FACTOR 0.75
3 const Color:BLUE-FACTOR 0.75
4 const Color:GREEN-FACTOR 0.75
```

When a data is defined in the class nameset, it is common to refer it as a *class data member* . A class 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 in the class nameset.

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 *class method* of that class. Class methods are invariant among the class instances. The standard declaration syntax for a lambda or gamma expression is still valid with a class.

```
1 const Color:get-primary-by-string (color value) {
2   trans val "0x"
3   val:+= (switch color (
4     ("red" (value:substr 1 3))
5     ("green" (value:substr 3 5))
6     ("blue" (value:substr 5 7))
7   ))
8   Integer val
9 }
```

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

```

1 Color:get-primary-by-string "red" "#23c4e5"
2 Color:get-primary-by-string "green" "#23c4e5"
3 Color:get-primary-by-string "blue" "#23c4e5"

```

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.

```

1 const Color:RED-VALUE "#ff0000"
2 const Color:print-primary-colors (color) {
3   println "red color " (
4     Color:get-primary-color "red" color)
5   println "green color " (
6     Color:get-primary-color "green" color)
7   println "blue color " (
8     Color:get-primary-color "blue" color)
9 }
10 # print the color components for the red color
11 Color:print-primary-colors Color:RED-VALUE

```

2. Instance

An *instance* of a class is an 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.

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.

```

1 const i ((class))
2 const Color (class)
3 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.

2.2. Instance initialization. When an instance is created, the engine looks for a special lambda expression called *preset*. 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 *preset* method.

```

1 const Color (class)
2 trans Color:preset (red green blue) {
3   const this:red (Integer red)
4   const this:green (Integer green)
5   const this:blue (Integer blue)
6 }
7 # create some default colors
8 const Color:RED (Color 255 0 0)
9 const Color:GREEN (Color 0 255 0)
10 const Color:BLUE (Color 0 0 255)
11 const Color:BLACK (Color 0 0 0)
12 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.

2.3. 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.

```

1  const c (class)
2  const c:a 1
3  const i (c)
4  const j (c)
5  const i:a 2
6  # class symbol access
7  println c:a
8  # shadow symbol access
9  println i:a
10 # non shadow access
11 println j:a

```

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.

```

1  const c (class)
2  const i (c)
3  const c:a 1
4  const i:a 2
5  println i:a
6  println i:meta:a

```

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

```

1  const c (class)
2  trans c:preset nil (const i (this:meta))
3  const i (c)

```

2.4. 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*, that is, 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.

```

1  const Color (class)
2  # class constructor
3  trans Color:preset (red green blue) {
4    const this:red (Integer red)
5    const this:green (Integer green)
6    const this:blue (Integer blue)
7  }
8  const Color:RF 0.75

```

```

9  const Color:GF 0.75
10 const Color:BF 0.75
11 # this method returns a darker color
12 trans Color:darker nil {
13   trans lr (Integer (max (this:red:* Color:RF) 0))
14   trans lg (Integer (max (this:green:* Color:GF) 0))
15   trans lb (Integer (max (this:blue:* Color:BF) 0))
16   Color lr lg lb
17 }
18 # get a darker color than yellow
19 const yellow (Color 255 255 0)
20 const dark-yellow (yellow:darker)

```

2.5. 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* symbol, it is possible to assert such equality.

```

1  # this method checks that two colors are equals
2  trans Color:== (color) {
3    if (== Color color:meta) {
4      if (!= this:red color:red) (return false)
5      if (!= this:green color:green) (return false)
6      if (!= this:blue color:blue) (return false)
7      eval true
8    } false
9  }
10 # create a new yellow color
11 const yellow (Color 255 255 0)
12 (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.

2.6. 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. As illustrated before, the class uses a default method `method` to initialize the data members.

```

1  # class declaration
2  const Complex (class)
3  # constructor
4  trans Complex:preset (re im) {
5    trans this:re (Real re)
6    trans this:im (Real im)
7  }

```

The constructor creates a complex object with the help of the real part and the imaginary part. Any object type which can be bound to a *Real* object is acceptable.

```

1  # class mutators
2  trans Complex:set-re (x) (trans this:re (Real re))
3  trans Complex:set-im (x) (trans this:im (Real im))
4  # class accessors
5  trans Complex:get-re nil (Real this:re)
6  trans Complex:get-im nil (Real this:im)

```

The accessors and the mutators simply provides the interface to the complex number components and perform a cloning of the calling or returned objects.

```

1  # complex number module
2  trans Complex:module nil {
3    trans result (Real (+ (* this:re this:re)
4      (* this:im this:im)))
5    result:sqrt

```

```

6 }
7 # complex number formatting
8 trans Complex:format nil {
9   trans result (String this:re)
10  result:+= "+i"
11  result:+= (String this:im)
12 }

```

The *module* and *format* are simple methods. Note the the complex number formatting is arbitrary here.

```

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

```

The *complex-p* predicate is the perfect illustration of the use of the *meta* reserved symbol. However, it shall be noted that the meta-comparison is done if and only if the calling argument is an instance.

```

1 # operators
2 trans Complex:== (c) (
3   if (complex-p c) (and (this:re== c:re)
4     (this:im== c:im)) (
5     if (number-p c) (and (this:re== c)
6       (this:im:zero-p)) false))
7 trans Complex:= (c) {
8   if (complex-p c) {
9     this:re:= (Real c:re)
10    this:im:= (Real c:im)
11    return this
12  }
13  this:re:= (Real c)
14  this:im:= 0.0
15  return this
16 }
17 trans Complex:+ (c) {
18   trans result (Complex this:re this:im)
19   if (complex-p c) {
20     result:re:+= c:re
21     result:im:+= c:im
22     return result
23   }
24   result:re:+= (Real c)
25   eval result
26 }

```

The operators are a little tedious to write. The comparison can be done with a complex number or a built-in number object. The assignation operator creates a copy for both the real and imaginary part. The summation operator is given here for illustration purpose.

3. Inheritance

Inheritance is the mechanism by which a class or an instance inherits methods and data member access from a parent object. The 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.

3.1. Derivation construction. A derived object is generally defined within the *preset* method of that instance by setting the *super* data member. The *super* reserved keyword is set to nil at the instance construction. The good news is that any object can be defined as a super instance, including built-in object.

```

1  const c (class)
2  const c:preset nil {
3    trans this:super 0
4  }

```

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. Another consequence of this scheme is that derived instance do not have to be built from the same base class.

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.

```

1  const c (class)
2  const i (c)
3  trans i:a 1
4  const j (c)
5  trans j:super i
6  println j:a

```

When a symbol is evaluated, a set of search rules is applied. The engine gives the priority to the class nameset vs the super instance. As a consequence, a class 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.

3.3. Instance re-parenting. The ability to set dynamically the parent instance make the object model an ideal candidate to support *instance re-parenting* . In this model, a change in the parent instance is automatically reflected at the instance method call.

```

1  const c (class)
2  const i (c)
3  trans i:super 0
4  println (i:to-string) # 0
5  trans i:super "hello world"
6  println (i:to-string) # hello world

```

In this example, the instance is originally set with an *Integer* instance parent. Then the instance is *re-parented* with a *String* instance parent. The call to the *to-string* method illustrates this behavior.

3.4. Instance re-binding. The ability to set dynamically the instance class is another powerful feature of the class model. In this approach, the instance meta class can be changed dynamically with the *mute* method. Furthermore, it is also possible to create initially an instance without any class binding, which is later muted.

```

1  # create a point class
2  const point (class)
3  # point class
4  trans point:preset (x y) {
5    trans this:x x
6    trans this:y y
7  }
8  # create an empty instance
9  const p (Instance)
10 # bind the point class

```

```
11 p:mute point 1 2
```

In this example, when the instance is muted, the *preset* method is called automatically with the extra arguments.

3.5. Instance inference. The ability to instantiate dynamically inferred instance is offered by the instance model. An instance *b* is said to be inferred by the instance *a* when the instance *a* is the super instance of the instance *b*. The instance inference is obtained by binding the *infer* symbol to a class. When an instance of that class is created, the inferred instance is also created.

```
1 # base class A
2 const A (class)
3 # inferred class B
4 const B (class)
5 const A:infer B
6 # create an instance from A
7 const x (A)
8 assert B (x:meta)
9 assert A (x:super:meta)
```

In this example, when the instance is created, the inferred instance is also created and returned by the instantiation process. The *preset* method is only called for the inferred instance if possible or the base instance if there is no inferring class. Because the base *preset* method is not called automatically, the inferred method is responsible to do such call.

```
1 trans B:preset (x y) {
2   trans this:xb x
3   trans this:yb y
4   if (== A this:super:meta) (this:super:preset x y)
5 }
```

Because the class can mute from one call to another and also the inferred class, the *preset* method call must be used after a discrimination of the meta class has been made as indicated by the above example.

3.6. Instance deference. In the process of creating instances, one might have a generic class with a method that attempts to access a data member which is bound to another class. The concept of class *deference* is exactly designed for this purpose. With the help of reserved keyword *defer*, a class with virtual data member accessors can be bound to a base class as indicated in the example below.

```
1 # create the base and defer class
2 const bc (class)
3 const dc (class)
4 # bind the base preset method
5 trans bc:preset nil (const this:y 2)
6 # bind the defer accessor to the base data member
7 trans dc:get-y nil (eval this:y)
8 # bind the defer class in the base class
9 const bc:defer dc
10 # create an instance from the base class
11 const i (bc)
12 # access to the base member with the defer method
13 assert 2 (i:get-y)
```

It is worth to note that the class deference is made at the class level. When an instance of the base class is created, all methods associated with the *deferent* class are visible from the base class, thus making the *deferent* class a virtual interface to the base class.

Advanced concepts

This chapter covers advanced concepts of the writing system. The first subject is the exception model. The second subject covers some properties of the namesets in the context of the interpreter object. The thread sub-system is then described along with the synchronization mechanism. Finally, some notes related to the functional system are given at the end of this chapter.

1. Exception

An *exception* is an unexpected change in the execution flow. The exception model 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.

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.

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

The example above is the general form to throw an exception. The first argument is 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.

```
1 throw
2 throw "type-error"
3 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. Within a try block, an exception can be thrown again by using the exception object represented with the *what* symbol.

```
1 try {
2   ...
3 } {
4   println "exception caught and re-throw"
5   throw what
6 }
```

1.2. Exception handler. The special form *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.

Symbol	Description
eid	Exception id
name	Exception file name
line	Exception line number
about	Exception extended reason
reason	Exception reason
object	Exception object

```

1 try (+ 1 2)
2 try (throw)
3 try (throw "hello")
4 try (throw "hello" "world")
5 try (throw "hello" "world" "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.

```

1 try (throw "hello")
2 (eval what:eid)
3 try (throw "hello" "world")
4 (eval what:reason)
5 try (throw "hello" "world" 2000)
6 (eval what:object)

```

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

```

1 # protected factorial
2 const fact (n) {
3   if (not (integer-p n))
4     (throw "number-error" "invalid argument in fact")
5   if (== n 0) 1 (* n (fact (- n 1)))
6 }
7 try (fact 5) 0
8 try (fact "hello") 0

```

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.

```

1 const nset (nameset)
2 const nset (nameset ...)

```

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.

```

1 trans a 1 # 1
2 block {
3   trans a (+ a 1) # 2
4   println ..:a 1 # 1
5 }
6 println a # 1

```

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.

```

1  const cls (class)
2  const ins (cls)
3  const ins:super (nameset)
4  const ins:super:value 2000
5  const ins:super:hello "hello world "
6  println ins:hello ins:value # hello world 2000

```

3. Delayed Evaluation

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

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.

```

1  const a (delay (+ 1 2))
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.

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.

```

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

```

4. Enumeration

Enumeration, that is, named constant bound to an object, can be declared with the reserved keyword *enum*. The enumeration is built with a list of literal and evaluated as is.

```

1  const e (enum E1 E2 E3)
2  assert true (enum-p e)

```

The complete enumeration evaluates to an *Enum* object. Once built, enumeration item evaluates by literal and returns an *Item* object.

```

1  assert true (item-p e:E1)
2  assert "Item" (e:E1:repr)

```

Items are comparable objects. Only items can be compared. For a given item, the source enumeration can be obtained with the *get-enum* method.

```

1  # check for item equality
2  const i1 e:E1
3  const i2 e:E2
4  assert true (i1== i1)
5  assert false (== i1 i2)
6  # get back the enumeration
7  assert true (enum-p (i1:get-enum))

```

Symbol	Description
argv	Command arguments vector
os-name	Operating system name
os-type	Operating system type
version	Full version
loader	The interpreter loader
resolver	The interpreter resolver
afnix-uri	Official uri name
program-name	Interpreter program name
big-endian-p	Machine big endian predicate
64-bits-p-p	Machine size predicate
major-version	Major version number
minor-version	Minor version number
patch-version	Patch version number
machine-size	The interpreter machine size

5. Logger

The *Logger* class is a message logger that stores messages in a buffer with a level. The default level is the level 0. A negative level generally indicates a warning or an error message but this is just a convention which is not enforced by the class. A high level generally indicates a less important message. The messages are stored in a circular buffer. When the logger is full, a new message replace the oldest one. By default, the logger is initialized with a 256 messages capacity that can be re-sized.

```
1 const log (Logger)
2 assert true (logger-p log)
```

When a message is added, the message is stored with a time-stamp and a level. The time-stamp is used later to format a message. The *length* method returns the number of logged messages. The *get-message* method returns a message by index. Because the system operates with a circular buffer, the *get-message* method manages the indexes in such way that the old messages are accessible with the oldest index. For example, even after a buffer circulation, the index 0 will point to the oldest message. The *get-message-level* returns the message level and the *get-message-time* returns the message posted time.

```
1 const msg (log:get-message 0)
```

In term of usage, the logger facility can be conveniently used with other derived classes. The standard i/o module provides several classes that permits to manage logging operations in a convenient way.

6. Interpreter

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

The interpreter provides also special methods which can be used to access internal features that do not operate like standard methods or functions. Some methods are also designed to modify the internal state of the interpreter. Note that some methods provide a mechanism to interact at the process level.

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

Symbol	Description
dup	duplicate the interpreter
roll	run the interpreter loop
wait	Wait for normal threads
load	Load a file and execute it
launch	Launch a normal thread
daemon	Launch a daemon thread
library	Load and initialize a library
deserialize	Deserialize an object
read-line	Get an input stream line
read-credential	Get an input credential
get-input-stream	Get interpreter input stream
get-output-stream	Get interpreter output stream
get-error-stream	Get interpreter output stream
wait-kill-signal	Wait for a process signal
get-primary-prompt	Get primary prompt
get-secondary-prompt	Get secondary prompt
set-absolute-precision	Set absolute precision
set-relative-precision	Set relative precision
get-absolute-precision	Get absolute precision
get-relative-precision	Get relative precision

```

1 zsh> axi hello world
2 (axi) println (interp:argv:length) # 2
3 (axi) println (interp:argv:get 0) # hello

```

6.2. Interpreter version. 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 is bound to the *interp:os-type* .

```

1 println "major number : " interp:major-version
2 println "minor number : " interp:minor-version
3 println "patch number : " interp:patch-version
4 println "version number : " interp:version
5 println "system name : " interp:os-name
6 println "system type : " interp:os-type
7 println "official uri : " interp:afnix-uri

```

6.3. Method load. 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. The module is loaded only once unless the force flag is used.

```

1 interp:load "module"
2 interp:load "module" true
3 interp:load "module" "tag" true

```

In the first form the module is loaded by name only once. In the second form, the module is loaded with a force flag set to *true* . In the third form, the module is given a tag

which is used to detect whether or not the module has been loaded. If no tag is given, the module name is used instead.

6.4. Method library. 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.

```
1 interp:library "afnix-sys"
2 println "random number: " (afnix:sys:get-random)
```

6.5. Method dup. The interpreter can be duplicated with the help of the *dup* method. Without argument, a clone of the current interpreter is made and a terminal object is attached to it. When used in conjunction with the *roll* method, this approach permits to create an interactive interpreter. The *dup* method also accepts a terminal object.

```
1 # duplicate the interpreter
2 const si (interp:dup)
3 # change the primary prompt
4 si:set-primary-prompt "(si)"
```

6.6. Method roll. The interpreter loop can be run with the *roll* . The loop operates by reading the interpreter input stream. If the interpreter has been cloned with the help of the *dup* method, this method provides a convenient way to operate in interactive mode. The method is not called *loop* because it is a reserved keyword and starting a loop is like having the ball rolling.

```
1 # duplicate the interpreter
2 const si (interp:dup)
3 # loop with this interpreter
4 si:roll
```

6.7. Method wait. The interpreter can wait for all normal threads to complete. When invoked, the interpreter monitors all normal threads and wait until the terminate normally. This is a standard synchronization method in a multithreaded environment.

```
1 # create a thread
2 launch f
3 # wait for completion
4 interp:wait
```

7. Librarian object

A *librarian file* is a special file that acts as a containers for various files. A librarian file is created with the `axl - 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.

7.1. 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.

```
1 zsh: axl -h
2 usage: axl [options] [files]
3 [h] print this help message
4 [v] print version information
5 [c] create a new librarian
6 [x] extract from the librarian
7 [s] get file names from the librarian
8 [t] report librarian contents
9 [f] lib set the librarian file name
```

The [c] option creates a new librarian. The librarian file name is specified with the [f] option.

```
1 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.

7.2. 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.

```
1 zsh> axi -i librarian.axl
2 (axi) interp:load "file-1.als"
3 (axi) 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.

7.3. 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.

```
1 zsh: axl -t -f librarian.axl
2 ----- 1234 file-1.als
3 ----- 5678 file-2.als
```

The one line report contains the file flags, the file size and the file name. 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.

7.4. 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.

```
1 zsh: axl -x -f librarian.axl
2 zsh: axl -x -f librarian.axl file-1.als
```

8. Librarian object

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

8.1. Output librarian. The *Librarian* object is a standard 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.

```
1 # create a new librarian
2 const lbr (Librarian)
3 # add a file into it
4 lbr:add "file-1.als"
5 # write it
6 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.

8.2. 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.

```

1 # open a librarian for reading
2 const lbr (Librarian "librarian.axl")
3 # get the number of files
4 println (lbr:length)
5 # extract the first file
6 const is (lbr:extract "file-1.als")
7 # is is an input stream - dump each line
8 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. File resolver

The *file 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 to perform any kind of file path resolution.

9.1. 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.

```

1 # create a new resolver
2 const rslv (Resolver)
3 assert true (resolver-p rslv)
4 # add the local directory on the search path
5 rslv:add "."
6 # check if file test.als exists
7 # if this is ok - print its contents
8 if (rslv:valid-p "test.als") {
9   const is (rslv:lookup "test.als")
10  while (is:valid-p) (println (is:readln))
11 }

```

10. Thread operations

The interpreter is a multi-threaded engine with a native implementation of objects locking. A thread is started with the reserved keyword *launch* . The execution is completed when all threads have terminated. This means that the master thread (i.e the first thread) is suspended until all other threads have completed their execution.

10.1. Starting a thread. A 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.

```

1 launch (nil)

```

There exists an alternate mechanism to start a thread with the reserved keyword *launch* and a thread object. Such mechanism is used when using deferred thread object creation or a thread generator object known as a *thread set* .

10.2. 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* command. The *thread-p* predicates returns *true* if the object is a thread descriptor.

```
1 const thr (launch (nil))
2 println (thread-p thr) # true
```

The thread result can be obtained with the help of the *result* method. Although the result can be accessed at any time, the returned value will be *nil* until the thread as completed its execution.

```
1 const thr (launch (nil))
2 println (thr:result) # nilp
```

Although the 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.

10.3. Future object. The *future* special form provides a simple mechanism to perform asynchronous evaluation. When a future object is created, the evaluation is pending a call to the *force* special form. When the future is complete, the evaluation result is available.

```
1 # create a future
2 const f (future 1)
3 # do not necessarily evaluates to 1
4 println (force f)
```

11. 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 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* guarantees that there is only one writer, but eventually multiple reader. When a thread starts 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.

11.1. 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.

```

1 # shared variable access
2 const var 0
3 # increase method
4 const incr nil {
5   while true (println "increase: " (var:= (+ var 1)))
6 }
7 # decrease method
8 const decr nil {
9   while true (println "decrease: " (var:= (- var 1)))
10 }
11 # start both threads
12 launch (decr)
13 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. 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.

```

1 # shared variable access
2 const var 0
3 # incrementer, decrementer and printer
4 const incr nil (while true (var:= (+ var 1)))
5 const decr nil (while true (var:= (- var 1)))
6 const prtval nil (while true (println "value = " var))
7 # start all threads
8 launch (decr)
9 launch (incr)
10 launch (prtval)

```

12. Synchronization

Although, there is 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.

12.1. Form synchronization. The reserved keyword *sync* can be used to synchronize a form. When a form, is synchronized, the engine guarantees that only one thread will execute this form. In the special case of the form being a future, the interpreter will block until the future is complete.

```

1 const print-message (code msg) (
2   sync {
3     errorln "error : " code
4     errorln "message: " msg
5   }
6 )

```

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

12.2. 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.

```

1 # simple flag
2 const flag false
3 # simple tester
4 const ftest (bval) (flag:= bval)
5 # run the thread and wait
6 const thr (launch (ftest true))
7 thr:wait
8 assert true flag

```

This example is taken from the test suites. It checks that a boolean variable is set when started in a thread. Note the use of the *wait* method to make sure the thread has completed before checking for the flag value. 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.

```

1 # shared vector of threads descriptors
2 const thr-group (Vector)
3 # wait until all threads in the group are finished
4 const wait-all nil (for (thr) (thr-group) (thr:wait))

```

12.3. Complete example. We illustrate the previous discussion with a complete example. The idea is to perform a matrix multiplication. A thread is launched when 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.

```

1 # initialize the shared library
2 interp:library "afnix-sys"
3 # shared vector of threads descriptors
4 const thr-group (Vector)
5 # waits until all threads in the group are finished
6 const wait-all nil (for (thr) (thr-group) (thr:wait))

```

The group of threads is represented as a vector. Based on the the previous discussion, a simple loop that blocks until all threads are completed is designed as a simple gamma expression.

```

1 # initializes a matrix with random numbers
2 const init-matrix (n) {
3   trans i (Integer 0)
4   const m (Vector)
5   do {
6     trans v (m:add (Vector))
7     trans j (Integer)
8     do {
9       v:add (afnix:sys:get-random)
10    } (< (j:++) n)
11  } (< (i:++) n)
12  eval m
13 }

```

The matrix initialization is quite straightforward. The matrix is represented as a vector of lines. Each line is also a vector of random integer number. It is here worth to note that the standard *math* module provides a native implementation of real matrix.

```

1 # this procedure multiply one line with one column
2 const mult-line-column (u v) {
3   assert (u:length) (v:length)
4   trans result 0
5   for (x y) (u v) (result:+= (* x y))
6   eval result
7 }
8 # this procedure multiply two vectors assuming one
9 # is a line and one is a column from the matrix
10 const mult-matrix (mx my) {

```

```

11  for (lv) (mx) {
12    assert true (vector-p lv)
13    for (cv) (my) {
14      assert true (vector-p cv)
15      thr-group:add (launch (mult-line-column lv cv))
16    }
17  }
18 }

```

The matrix vector multiplication is at the heart of the example. Each line-column multiplication is started into a thread and the thread object is placed into the thread group vector.

```

1  # check for some arguments
2  # note the use of errorln method
3  if (== 0 (interp:argv:length)) {
4    errorln "usage: axi 0607.als size"
5    afnix:sys:exit 1
6  }
7  # get the integer and multiply
8  const n (Integer (interp:argv:get 0))
9  mult-matrix (init-matrix n) (init-matrix n)
10 # wait for all threads to complete
11 wait-all
12 # make sure we have the right number
13 assert (* n n) (thr-group:length)

```

The main execution is started with the matrix size as the first argument. Two random matrices are then created and the multi-threaded multiplication is launched. The main thread is blocked until all threads in the thread group are completed.

12.4. 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.

```

1  # create a condition variable
2  const cv (Condvar)
3  # this function runs in a thread - does some
4  # computation and mark the condition variable
5  const do-something nil {
6    # do some computation
7    ...
8    # mark the condition
9    cv:mark
10 }
11 # start some computation in a thread
12 launch (do-something)
13 # block until the condition is changed
14 cv:wait-unlock
15 # continue here
16 ...

```

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.

13. 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.

```
1 (axi) 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.

```
1 (axi) println ((gamma (n) (+n 1)) 1)
2
```

13.1. 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, the writing system 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.

```
1 (axi) println ((gamma (n)
2   (if (<= n 1) 1 (* n (self (- n 1))))) 5)
3 120
```

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

13.2. Closed variables. One of the writing system 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.

```
1 trans x 1
2 const do-print nil {
3   trans x 2
4   println ((lambda (n) (+ n x)) 1)
5 }
```

The gamma expression *do-print* 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.

```

1 trans x 1
2 const do-print nil {
3   trans x 2
4   println ((gamma (n) (+ n x)) 1)
5 }

```

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 .

```

1 trans x 1
2 const do-print nil {
3   trans x 2
4   println ((gamma (n) (x) (+ n x)) 1)
5 }

```

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*.

13.3. Dynamic binding. Because there is 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.

```

1 lambda (n) {
2   if (<= n 1) (trans x 1)
3   println (+ n x)
4 }

```

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.

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

```

1 (axi) const sym (protect lex)
2 (axi) println (sym:repr)
3 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.

```

1 (axi) const sym (protect lex)
2 (axi) println (sym:to-string)
3 lex

```

13.5. 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.

```

1 (axi) const obj 0
2 (axi) const lex (protect obj)
3 (axi) const sym (lex:map)
4 (axi) println (sym:repr)
5 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.

```

1 (axi) println (sym:get-const)
2 true
3 (axi) println (sym:get-object)
4 0
5 (axi) sym:set-object true
6 (axi) println (sym:get-object)
7 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.

```

1 (axi) const sym (Symbol "symbol")
2 (axi) println sym
3 symbol
4 (axi) sym:= 0
5 (axi) println (eval sym)
6 0

```

13.6. 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.

```

1 (axi) const f (Closure)
2 (axi) println (closure-p f)
3 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.

```

1 # the simple way
2 trans f nil (println "hello world")
3 # the complex way
4 const f (Closure)
5 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.

Installation Guide

This chapter describes the installation procedures for the **AFNIX** writing system distribution. This chapter explains how to set and compile this distribution.

1. Software distribution

The complete distribution can be downloaded from the **AFNIX home page** . The result is a complete source tree that is ready for compilation. The distribution contains also the documentation as well as examples. The distribution is supported on a variety of platforms as indicated below that can be either 32 bits or 64 bits machines. The distribution is also available at the **FreeBSD port collection** .

Specific platforms like GNU KBSD or GNU Hurd have limited support. Feel free to report any issue at (afnix.org) **AFNIX bug** . Specific processors like the Alpha, M68K, ARM, MIPS, RISC-V and SUPERH are also supported on certain distributions. The PowerPC (PPC) processor has been discontinued. The Solaris SPARC platform has been discontinued. Do not hesitate to contact the development team for specific processor or platform support at (contact@afnix.org) **AFNIX contact** .

2. Installation procedure

The core software is written in C++. It has been successfully built with the latest *GNU GCC 10* . The *clang* compiler has also been successfully tested. You will also need the *GNU Make* package. With some distributions the command is called **gmake** . Note that the *Makefile* hierarchy is designed to operate safely with the `[-j]` *GNU Make* option.

2.1. Unpacking the distribution. The distribution is available as a compressed tar file. Note that the documentation is distributed in a separate file. The following command unpacks the distribution.

```
1 zsh> gzip -d afnix-src-[version].tar.gz
2 zsh> tar xf afnix-src-[version].tar
```

2.2. Quick command reference. The list of commands to execute is given in the example below. A detailed description for each command is given hereafter. The **make world** command is the default command that builds the whole tree with the default compiler.

```
1 zsh> ./cnf/bin/afnix-setup -o --prefix=/usr/local/afnix
2 zsh> make status
3 zsh> make [-j]
4 zsh> make test
5 zsh> make install
6 zsh> make clean
```

Platform	Processor	Operating system
Linux	X86-32, X86-64	Linux 4.x, 5.x, 6.x (Fedora, Debian, Ubuntu, ...)
FreeBSD	X86-32, X86-64	FreeBSD 13.x.x, 14.x.x

Option	Description	Default
-h	Print a help message	n/a
-v	Set the verbose mode	n/a
-g	Set the debug mode	yes
-o	Set the optimized mode	no
-help	Same as -h	n/a
-prefix	Set the target install directory	/usr/local
-shrdir	Set the shared install directory	/usr/local/share
-altdir	Set the alternate install directory	/usr/local
-sdknam	Set the target sdk by name	platform dependent
-sdkdir	Set the target sdk directory	platform dependent
-ccname	Set the default compiler	platform dependent
-shared	Compile and link dynamically	yes
-static	Compile and link statically	no
-openmp	Enable the optional openmp compilation	no

With some platforms, the `make` command should be replaced by the `gmake` command. The `make status` command is optional and can be used to report the internal value contents. In particular, the version and the installation parameters are reported.

2.3. Configuration. The `afnix-setup` command can be invoked to setup a particular configuration. You should have your compiler on your search path. Normally, the command given below is enough.

```
1 zsh> ./cnf/bin/afnix-setup -o --prefix=/usr/local/afnix
```

This command checks that the target platform can be detected and configured. The `[-o]` option configures the compilation in optimized mode. Use the `[-g]` option can be used to configure the build process in debug mode. The `[--prefix]` option sets the installation directory. Note that the compilation process is done in the distribution tree and that the `[--prefix]` option affects only the installation operations. The `[-v]` option is the verbose option. Other options are available for fine tuning.

The `[prefix]` option set the root installation directory. The binary and library installation directories are derived from it. The `[shrdir]` set the shared installation directory which is normally used for the installation of the manual pages on most popular systems. the `[altdir]` sets the alternate installation directory. Normally this path should be empty as it affects the path for the `etc` directory. This flag should be used when using a prefix to unusual destination. The `[ccname]` option can be used to force a particular compiler with the help of a compiler configuration file. The `[-s]` or `[--static]` option can be used to build a static executable. Normally, this option should not be used since it restrict the use of extension modules. The `[shared]` controls whether or not the dynamic libraries should be built. This option is detected automatically for a particular platform and should be used only by package maintainer. There exists also specific options which are mostly for package maintainers. At this time, the build process integrates the Debian, Ubuntu and Fedora specific packaging mechanism.

Option	Description	Default
-pkgnam	Set the distribution package	none
-pkgbin	Set the optional package bin directory	none
-pkglib	Set the optional package lib directory	none
-pkgprj	Set the optional package prj directory	none
-pkghdr	Set the optional package hdr directory	none
-pkgetc	Set the optional package etc directory	none
-pkgman	Set the optional package man directory	none
-pkgdoc	Set the optional package doc directory	none
-pkgwww	Set the optional package www directory	none
-pkgsrv	Set the optional package srv directory	none

2.4. Compiling the distribution. The compilation process is straightforward. With some platforms, the `make` accepts the `[-j]` that enables concurrent operations.

```
1 zsh> make [-j]
```

This will build the complete distribution locally. If an error occurs, it is best to report it at the (bugs@afnix.org) **AFNIX bug report** mail address.

2.5. Testing the distribution. The distribution contains all test suites. The test suites are compiled and executed with the following command.

```
1 zsh> make test
```

This command run the test suites for each library as well as the test suites for each application client. Most of the base library test suites are written in C++ with the application test suites written in the core writing system.

2.6. Installing the distribution. Once the system has been built and tested, it can be installed. By default, the distribution tree is installed into the `/usr/local` directory. This can be overwritten with the `[--prefix]` option during the configuration process.

```
1 zsh> make install
```

There are several variables that controls the behavior of the `install` rule. Each variable has its default value sets during the setup configuration. However, this variable can also be altered during the installation process

2.7. Installing the documentation. The documentation is installed independently of the software. The `doc` rule builds the documentation and the `publish` rule installs the documentation. Several variables also control the documentation installation path.

Variable	Description	Default
PREFIX	The root install directory	/usr/local
SHRDIR	The shared install directory	/usr/local/share
ALTDIR	The shared alternate directory	/usr/local/etc
SDKDIR	The system kit directory	platform dependent
BINDIR	The binary install directory	prefix/bin
LIBDIR	The library install directory	prefix/lib
HDRDIR	The header files install directory	prefix/include/afnix
ETCDIR	The extra files install directory	altdir/etc/afnix

Variable	Description	Default
DOCDIR	The documentation install directory	shrdir/doc/afnix
MANDIR	The manual pages install directory	shrdir/man

2.8. Cleaning the distribution. The distribution is cleaned with the `clean` rule.

```
1 zsh> make clean
```

This rule does not clean the configuration. For a complete cleaning the `reset` rule is more appropriate.

```
1 zsh> make reset
```

3. Running AFNIX

The `axi` command invokes the interpreter. In order to operate properly, the `LD_LIBRARY_PATH` environment variable must be configured with the directory containing the shared libraries. If the libraries have been installed in a standard location like `/usr/local/lib`, there is nothing to do.

3.1. Running some examples. The directory `exp` contains various examples which can be run. Each example is labeled according to their use in the volume 1 of the documentation set. Example `0101.als` prints the message `hello world`. Example `0501.als` prints various information about the system configuration.

```
1 zsh> axi EXP0501
2 major version number : 4
3 minor version number : 0
4 patch version number : 0
5 interpreter version : 4.0.0
6 program name : axi
7 operating system name : linux
8 operating system type : unix
9 machine size : 64
10 afnix official uri : http://www.afnix.org
```

4. Special features

The build process provides several features that permits to customize the compilation process as well as the nature of the final executable. Most of the time, these options are reserved for the package maintainer and are given below for illustration purpose.

4.1. Special target extensions. Extensions are specific libraries or executables which are not build automatically during the build process. The user is responsible to decide which extension is needed for the system All extensions are located under the `src/ext` directory. Simply going into the appropriate directory and running the `make` command will build the extension.

The `asi` extension creates a static interpreter with all libraries automatically included in the final executable. The extension is simply build with the following command. Note that this extension overwrite the previous executable in the `bld/bin` directory.

```
1 zsh> make -C src/ext/asi
```

4.2. Extra files. The distribution comes with some extra files. The most important is the Emacs mode `afnix-mode`. The original source file is written in Emacs Lisp and is available in the `etc` directory of the distribution. This file should be installed according to the current Emacs installation.

Maintainer notes

This chapter contains additional notes for the package maintainer. They are also useful for anybody who is in charge of integrating the distribution in a build process. The chapter describes the distribution tree with more details.

1. The distribution tree

The distribution tree is composed of various directories. Each of them has a `Makefile` which can be called locally or from the top level.

- **cnf**
This directory contains the configuration distribution and various utilities. Normally you should not touch it, unless you are using a compiler different than gcc.
- **src**
This directory contains the complete source tree. The source code is written in C++. Normally this directory is left untouched. If there are good reasons to modify it, please contact the development team.
- **tst**
This directory contains the complete test suites. The test suites are used by various programs including the main interpreter, the compiler and the debugger. It shall be noted that the library distribution also includes specific test suites.
- **doc**
This directory contains the complete documentation written in XML with a special DTD. It should be left untouched.
- **etc**
This directory contains various files associated with the distribution. Some files are useful to be copied.
- **exp**
This directory contains various examples. They are included for illustration purpose.

The process of building a package solely depends on the distribution type. Most likely, the standard distribution should contain the binary executables as well as some configuration file and the manual pages. The documentation and the development header files can put in separate packages.

2. Configuration and setup

The configuration process involves the use of the `afnix-setup` command located in the `cnf/bin` directory. This command is used to configure the distribution. Package maintainers are encouraged to use it with specific options.

2.1. Platform detection. The `afnix-guess` command is used during the configuration process to detect a supported platform. This command can be run in stand-alone mode. Various options can be used to tune the type of information requested.

Without option, the utility prints a platform and processor description string.

Option	Description
-h	Print a help message
-n	Print the platform name
-v	Print the platform version
-M	Print the platform major number
-m	Print the platform minor number
-p	Print the processor name

```
1 zsh> ./cnf/bin/afnix-guess
2 linux-5.4-x64
```

2.2. Platform defaults. The directory `cnf/def` contains a platform specific default file. The file determines what is the default compiler and linking mode. This file is used by the `afnix-setup` command. For example, the `afnix-darwin.def` file contains:

```
1 compiler: gcc
2 lktype : dynamic
3 lkmode : dylib
```

Such options instructs the configuration utility, that the default compiler is `gcc` and the linking mode should operate in dynamic mode by using the `[dylib]` rule. These default values can be overwritten with the equivalent option of the `afnix-setup` command. Note that the compiler version is automatically detected by the system. The `afnix-vcomp` command will return the appropriate compiler version running on the target system.

2.3. C++ source file conventions. The source tree has two types of C++ files. The first type has the extension `.cxx` and the second type has the extension `.cpp`. The `.cxx` – and the associated `.hxx` – files are only used to indicate a system dependency. These files are found only in the `src/lib/plt` directory. The `.cxx` extension indicates that the file might use system specific include files. The `.cpp` – and the associated `.hpp` – files are the normal C++ source files. The `.cpp` extension is used to indicate that these files will not use a system specific file. By default this rule is enforced in the compiler configuration file by specifying some compiler flags which do not authorize such access.

2.4. Configuration files. The configuration files are located in the `cnf/mak` directory. Normally they should be left untouched. The most important one is the `afnix-rule.mak` file that defines most of the compilation and linking rules. Additionally, during the setup operation, the `afnix-setup` command creates several files in the `bld/cnf` directory. The `bld` is the build directory. The `afnix-plat.mak` file is the platform configuration file and the `afnix-comp.mak` is a link to the appropriate compiler configuration file.

3. Compilation

Normally, the compilation process is immediate. Just invoking the `make` command will do the job. However, some package maintainers have the desire to overwrite some flags. Some options are provided to facilitate this task.

- **EXTCPPFLAGS**

This flag can be used to add some compilation flags for all `.cpp` files.

- **EXTCXXFLAGS**

This flag can be used to add some compilation flags for all `.cxx` files.

- **EXTCCDEFINE**

This flag can be used to add some compilation definitions for all source files.

- **EXTINCLUDES**

This flag can be used to add some compilation paths for the .cxx files.

For example, it is common to have some maintainer to compile with both the debug and optimize flags. This can be done with the following command (assuming an optimized configuration):

```
1 make EXTCPPFLAGS=-g EXTCXXFLAGS=-g
```

All include files, compiled libraries and executables are placed in the `bld` directory. This directory contains the `bld/bin` for binaries, `bld/lib` for libraries and `bld/hdr` for the header files.

4. Building the package

The package can be built by accessing the `bld` directory or by invoking the `install` rule. The second method is not recommended for package construction, since it might trigger some file installation without any control.

The `etc/unx` directory contains some special files that might be used for the package construction. A sample list of them is given hereafter.

- **afnix-mode.el**
This file is the Emacs mode.
- **afnix-gud.el**
This file is the debugger Emacs gud mode.

5. Specific makefile rules

The top level `Makefile` contains several rules that might be useful for the package maintainer.

- **status**
This rule show the configuration status for each parameters with the version.
- **debug**
This rule invokes the default configuration in debug mode.
- **optimized**
This rule invokes the default configuration in optimized mode.
- **build**
This rule invokes the default configuration in debug mode and compile the whole distribution. The default install directory is `/usr/local`.
- **world**
This rule invokes the default configuration in optimized mode and compile the whole distribution. The default install directory is `/usr/local`.
- **test**
This rule runs all test suites.
- **doc**
This rule builds the documentation.
- **distri**
This rule builds the distribution.
- **install**
This rule installs the distribution.
- **publish**
This rule installs the documentation.
- **clean**
This rule cleans the distribution but keep the configuration.

- **reset**

This rule resets the distribution including the configuration.

APPENDIX C

Release notes

This chapter contains the release notes for the different releases. Release notes are given in descending order for a particular version. The standard notation is *major.minor.patch* which represents respectively, the major, minor and patch number. A major version number changes indicates a substantial change in the distribution, including new tools, application interface and license. A minor version number change indicates noticeable change, with or without new tools but without application interface change nor license change. Finally, a patch number change indicates a simple change to fix problem. There is no additional features in a patch nor an application interface change.

1. Release 4.0

- **Core engine: release 4.0.0**

This is an initial major structural release with minor features. The release contains a complete refactoring of the platform structure now called driver (drv) those first target is Unix (unx). The release also incorporates numerous fixes in the tls service and full support of the tls 1.2. This release is a transition release between 3.X and 4.X.

2. Release 3.9

- **Core engine: release 3.9.0**

This is a minor release which integrates several fixes and couple of new objects.

3. Release 3.8

- **Core engine: release 3.8.0**

This is a medium release in line with the previous release which integrates some fundamental restructuring of the core interpreter in preparation for distributed execution. The module object has been refactored as well as the debugger.

- **Core engine: hyper threaded interpreter**

A new object called an hyper-threaded interpreter which operates with a grid provides a mechanism to executes module inside an executing interpreter. This is still an experimental technology.

- **Core modules: tcz service**

The csm service has been refactored as a transmutable content zone service.

4. Release 3.7

- **Core engine: release 3.7.0**

This is a medium release in line with the previous release which integrates various extension and cleanup before major release 3.8 scheduled for the end of the year.

- **Core engine: new special forms and continue**

Two new special forms, break and continue, have been added to control loops.

- **Core modules: gfx module**
The graph module has been revisited with numerous cleanup. A new state machine object called Automaton has been added.
- **Core modules: sys module**
An absolute wait time call has been added to act as a timer.
- **Core modules: nwg module**
A new class called Iso has been added. In particular, support for ISO 3166 has been added.

5. Release 3.6

- **Core engine: release 3.6.0**
This is a medium release which integrates a lot of cleanup as well as the beginning of the integration of the tls service. The unicode database has been updated to the latest 14.0.0 release. Some objects have been completely refactored, including base string vector.
- **Core modules: mth module**
Algebra with complex number has been added including computation algorithm with Hilbert space.
- **Core modules: nwg module**
The json reader/writer has been enhanced to support plist.
- **Core modules: sec module**
The pkcs signature algorithm has been added. It is designed for the support of ephemeral Diiffie-Hellman in the tls service.
- **Core services: tls service**
The tls service has been enhanced to support ephemeral Diiffie-Hellman. The integration of the tls inside applications will be able to start.

6. Release 3.5

- **Core engine: release 3.5.0**
This is a important release which integrates numerous objects that were long overdue, including the support for complex numbers and numerous code cleanup. An undetected (until now) interlocking bug in the string split method has been discovered and fixed. Various encoding bug have also been fixed including one in the readline method.
- **Core modules: mth module**
Prime number generation has been improved for speed, especially with safe prime. The quaternion object has been added.
- **Core modules: nwg module**
Form encoding and decoding has been enhanced. A json reader/writer has also been integrated.
- **Core modules: sec module**
The dsa pki has been added with support for dsa parameters. The dh key support has been added. The concept of key configuration and renewal has been added.
- **Core modules: xml module**
The module has been cleaned for simplex reader/writer design. The node forming generation has bee enhanced with indentation support.
- **Core services: tls service**
Pkcs8 support has been added with a better integration of key generation. Server name in extension is now supported.

- **Core services: csm service**

Datum and magma parts have been added along a general service cleanup. The concept of transmuter has been developed as a mixture interface. The blob view construction has been revisited.

7. Release 3.4

- **Core engine: release 3.4.0**

This is a a minor release which integrates more support for the tls service. Safe prime number generation has been added. The assert special form has been enhanced to better support real testing.

- **Core modules: mth module**

Real vector, point and matrix of size 1,2,2 and 4 has been added.

- **Core modules: sec module**

The gcm mode has been added. The elliptic curve objects and its associated arithmetic has been integrated. Tests with the standard curves have been incorporated.

- **Core modules: sys module**

The date and time object have been enhanced to support extended time with real precision below the second. A now constant is also available for setting the current date and time.

- **Core services: tls service**

The aes-gcm mode has been added as a cipher.

- **Core services: phy service**

Direct access to physical constants has been added. The unit string representation has been enhanced to support the scaling factor.

8. Release 3.3

- **Core engine: release 3.3.0**

This is a major release with a complete re-factoring of the interpreter evaluation hierarchy. Internally, some objects are being rewritten with a c++/17 semantic. Many objects have also been optimized to support very long strings. The serialization coding has also been revisited. A serious bug with negative proleptic date has been fixed. Another bug with the object iterator has also been fixed.

- **Core engine: Unicode 13.0.0**

The Unicode standard, revision 13.0.0, has been incorporated in the core engine.

- **Core engine: asynchronous evaluation**

The special form future creates a special object called a future which is used to evaluate an object asynchronously. The evaluation starts with the help of the force special form. The sync special form can be used to synchronize with the future.

- **Core modules: mth module**

The numeral block has been enhanced to support row/column order and line/block padding and the interface has been considerably enhanced.

- **Core services: phy service**

The Unit object has been added to support the definition of physical unit. The physic constants and the periodic table has been re-factored.

- **Core services: dip service**

The geo service has been removed and the image objects have been moved to a new service called dip.

9. Release 3.2

- **Core engine: release 3.2.0**
This is a minor release issued for synchronization with other projects.
- **Core engine: Numeral object**
The numeral object has been completely re-factored including the api at the math module level.
- **Core modules: nwg module**
A generic json writer has been implemented and the corresponding json mime object has been updated.

10. Release 3.1

- **Core engine: release 3.1.0**
This is a major release which new standard objects as well as the first distribution of the tls. Release 3.0 was an internal release which is will not be publicly distributed. The debian distribution has been fully integrated and the build system revisited to account for new platforms.
- **Core engine: Unicode 12.1.0**
The Unicode revision 11.2.0 has been incorporated in the core engine.
- **Core engine: Task and structure**
The core engine has been enhanced to support the concept of task. A Task is similar to a thread object, but can be used directly at the api level. A new Structure object has been added to support object aggregation. engine.
- **Core modules: net module**
The buffer read and socket sapf (socket address protocol family) have been fixed. The Autocom object has bee added to support socket re-connection.
- **Core modules: nwg module**
The IPV6 address representation in uri has been fixed. The Hyperlink object has been added.
- **Core modules: geo module**
The Netpbm object has been fixed for inconsistent block delete. The Pixmap stride has been fixed.
- **Core modules: sec module**
The Sha-3 family of hashers has been added. The Kdf2 function has bee refactored to support the Pbkdf2 standard. Support for Galois field has been added. The Gcm cipher has been added.
- **Core modules: sps module**
The csv split and validation process has been added. The Transit has been revisited.
- **Core modules: mth module**
The integer vector object Ivector has been added. The float vector Fvector has been added. Random vector generation has been added.
- **Core modules: sys module**
System wait on kill has been added.
- **Core services: csm service**
The csm objects have been considerably re-factored. Notably the Part and Blob objects.
- **Core projects: adp project**
The standard documentation processor has been enhanced with more option for better distribution support.

11. Release 2.9

- **Core engine: release 2.9.3**
Immediate release which fixes a potential deadlock in the output stream object.
- **Core engine: release 2.9.2**
Small revision release with small allocator fixes, doc improvements and 32 bits inconsistencies removal. The support for 'clang' compiler has been updated as well. A patch with support for the RISC-V processor has also been incorporated.
- **Core engine: release 2.9.1**
Small revision release with minor improvements for IPV6 address support.
- **Core modules: nwg module**
The Uri object has been enhanced to support numerical ip address notation with brackets.
- **Core modules: net module**
The Address object has been enhanced to support numerical ip address detection.
- **Core engine: Unicode 11.0.0**
The Unicode revision 11.0.0 has been incorporated in the core engine.
- **Core engine: serialization revisited**
The core engine serialization has been revisited to account for a larger number of modules and services.
- **Core engine: default hash revisited**
The core engine hash function is now based on the Fowler-Noll-Vo algorithm.
- **Core engine: logger object revisited**
The core engine logger object has been completely redesigned with a simpler interface.
- **Core modules: nwg module**
The base 64/32/16 codec have been added as a single object Basexx.
- **Core modules: sec module**
The security module has been enhanced with a new signature base class.
- **Core modules: wgt module**
The widget module has been added to the core distribution. At this time, the module contains expressible and conditional objects.
- **Core services: csm service**
The content session management service has been largely refactored.
- **Core services: geo service**
The geometry service has been enhanced to support image and pixmap. A netpbm reader/writer has been added as well.

12. Release 2.8

- **Core engine: release 2.8.3**
Release with GCC 8 support.
- **Core engine: release 2.8.2**
Intermediate release with GCC 8 support.
- **Core engine: release 2.8.1**
Incorporated patches for GCC 7.
- **Core engine: Unicode 9.0.0**
The Unicode revision 9.0.0 has been incorporated in the core engine.
- **Core engine: Collectable objects**
The Collectable object is now being deployed inside the engine. A collectable object provides a release method which can be used to remove links between object.

- **Core engine: full duplex stream**
A full duplex object has been added as a generic object. A full duplex stream is provided for certain class of object like the network socket. The default mode of operations remains the half-duplex stream since stream access are protected by mutexes.
- **Core modules: mth module**
The math module incorporates an infix notation parser module. This is a preliminary work supposed to grow over the next releases. The mean, covariance and univariate regression has been added as objects.
- **Core modules: sps module**
The bundle object literal index has been updated to follow the bundle length.

13. Release 2.7

- **Core engine: release 2.7.0**
Minor platform updates. This is an internal release in preparation for the 2.8.0 release.
- **Core modules: net module**
Revisit socket options and parameters
- **Core services: csm service**
Minor fixes with the agent accessor.

14. Release 2.6

- **Core engine: release 2.6.3**
Incorporated patches for GCC 6.
- **Core engine: release 2.6.2**
Incorporated patches for Debian issued by maintainer.
- **Core engine: release 2.6.1**
Fixed the FreeBSD build.
- **Core engine: release 2.6.0**
This is the release 2.6.0. The code has been updated to better support c++/11 for both gcc and clang.
- **Core modules: xml module**
The processing of character entity reference has been substantially changed to adhere fully with the xml specification.
- **Core modules: sps module**
The spreadsheet importer has been updated and a csv reader has been added. The importation process has also been revisited.
- **Core modules: sec module**
The security module has been updated to support the tls. This include moving to a 6 bits mersenne-twister and adding a certificate block.
- **Core modules: mth module**
The normal deviate has been added.
- **Core services: geo service**
A new geometry service has been added to the distribution. This provides support for modeling various shapes and solids.
- **Core services: cda service**
The streamable objects has been added to the crowd data analytics service.

15. Release 2.5

- **Core engine: release 2.5.2**

This is the release 2.5.2. The release includes support for the GCC 5 / CLANG 5 compilers suite. Thank's to Martin Michlmayr for pointing this out.

- **Core engine: release 2.5.1**

This is the release 2.5.1. This is an emergency release that fixes a bug in the uri percent encoding which can be downloaded from here . It's amazing that it was not found before.

- **Core engine: clang compiler with C++11 support**

The system infrastructure has been cleaned to support the clang compiler. The code has also been cleaned to support the C++11 standard which is now the compilation default.

- **Core engine: standard library**

A Style object to support in a generic way the formatting operation for the literal objects. Consequently, most of the literal objects have been cleaned and now support a format method that operates with a Style argument. The PrintTable object has been enhanced to support column extension and column style.

- **Core modules: mth module**

The ln method is now the standard preferred name for the natural logarithm. Several bugs have been fixed in the real matrix implementation with respect to the openmp implementation. A new solver based on the Modified Gram-Schmidt algorithm. has been added. Note also that the direct solver interface has been updated.

- **Core modules: nwg module**

The Uri class has been enhanced to support partial uri path extraction.

- **Core modules: sps module**

The spreadsheet module has been considerably overhauled. A new object Lstack has been added as a literal stack which can be bound to the cell as a literal array. The sheet formatting has been also revisited to make profit of the new Style object as well as the printable transformation.

- **Core modules: sys module**

The Meter object has been added as a mean to help for the performance measurements.

- **Core module: xml module**

The unicode conversion with reference has been updated to adapt itself to the stream or buffer encoding during a write process.

- **Core services: phy service**

A preliminary set of nuclear physics constants have been added.

- **Core services: csm service**

The Workspace object has been enhanced to support output stream.

- **Core services: web service**

A JsonMime object has been added. It is designed to translate various object into a Javascript object notation format. At this time, only the real data samples Rsamples object is supported.

16. Release 2.4

- **Core engine: unicode 6.3.0**

The Unicode 6.3.0 database is now supported in this release.

- **Core engine: containers**

The standard object incorporates an alias table which enables the mapping of

property name in a plist. The Trie object has been enhanced to support a reference index. The trie name mapping is now obtained with the to-names method.

- **Core engine: parallel support**

This release incorporates an experimental support for OpenMP. The OpenMP threads are compatible with the afnix threads and support is initially available in the math module. This option must be enabled explicitly in the build setup to be effective.

- **Core engine: crowd service**

The session user registration id has been enhanced. The concept of crowd service is available through the generic XaaS object, and more specifically with the SaaS object. The Workspace object has been enhanced with a public zone.

- **Core modules: sio module**

The Intercom object has been enhanced to support a buffered serialization which was somehow mandatory when operating in udp mode.

- **Core modules: net module**

Numerous deadlocks in the socket class have been fixed as well as udp inconsistencies.

- **Core modules: mth module**

The Qmr Krylov solver has been added as part as the iterative solver family. The krylov convergence test has also been improved. The Qr solver has been added as part as the direct solver family. The sparse matrix has been enhanced to support a generic iterator. The whole solver architecture has been revisited and is now a class based architecture with a type driven factory. Numerous bugs have been fixed. Vector and matrix now support row permutations.

17. Release 2.3

- **Core engine: release 2.3.2**

This is the release 2.3.2. The release fixes the real samples array serialization.

- **Core engine: release 2.3.1**

This is the release 2.3.1. Release 2.3.0 was an internal work which has not been distributed.

- **Core engine: interpreter line read**

The interpreter object can read a line or a passphrase from the attached terminal.

- **Core engine: interpreter daemon and librarian**

The interpreter can be put in daemon mode. This mean that a new detached processed is spawned with the interpreter attached to it. The librarian has been simplified and the loader integrated inside the interpreter.

- **Core engine: input stream**

The input steam nom implements a stream consumption method designed to accumulate a stream content into the stream buffer. Subsequently, the buffer can be converted into a string. The serialization of eos has been fixed.

- **Core engine: string resolver**

The resolver has been enhanced to map a file into a string. This methodology also applies to file present in a librarian.

- **Core engine: property list**

The Plist object has been enhanced to better support the merging operation.

- **Core engine: unicode 6.2.0**

The Unicode 6.2.0 database is now supported in this release.

- **Core modules: mth module**

The real matrix and vector implementation has been redesigned to support a

unique sparse representation. The serialization has also been added to these objects.

- **Core services: csm module**

The crowd object set has enhanced with an intercom crowd object which simplify the transmission of registered crowd object. A cart and a cart set object have been added as a mean to store crowd object. The session object and sessions set objects have been added. The session object has been enhanced to produce the associated session cookie, with an augmented session closing mechanism.

- **Core modules: nwg module**

The UriPath object has been added as a uri path manipulator for http server.

- **Core services: phy service**

The silicon energy gap has been fixed to the standard value. The periodic table structure has been revisited.

- **Core services: wax service**

The xhtml form elements have been added to the service. A base element class has been also added to almost all elements. The base class provides support for setting the common element attributes.

18. Release 2.2

- **Core engine: hurd platform**

The Hurd platform is now supported in this release. Thanks to our contributor for delivering this new platform.

- **Core engine: unicode 6.0.0**

The Unicode 6.0.0 database is now supported in this release.

- **Core engine: object updates**

The lexical analyzer is now an object in its own. It can be used to construct other object from a string description.

- **Core modules: mth module**

An automatic linear system verification has been added to the linear solver. Jacobi preconditionner have been added to the Krylov solvers and Newton solvers have been improved.

- **Core services: phy service**

The periodic table of the elements is under construction and should be completed soon. The table will provides the information for each elements, including name, symbol and other physical constants. The suport for intrinsic carrier concentration is now available. This is a cryptic feature for people working on semiconductors.

19. Release 2.1

- **Core engine: superh processor**

The SuperH processor is now supported in this release. The SuperH is a 32 bits processor.

- **Core engine: nan real number**

The implementation now supports the concepts of Not a Number or NAN as a whole. A real object can set and tested for NAN.

- **Core engine: indirect librarian resolver**

The resolver has been enhanced to support indirect librarian reference.

- **Core modules: csm module**

The personnal information management module has been renamed into the crowd session management or afnix-csm module.

- **Core modules: mth module**
The math module has been dramatically enhanced. The Rsamples object has been added for storing data samples. Function and polynomial objects have added to support generic function computation. The non-linear Newton system solver has been added as an object.
- **Core services: svg service**
The Scalable Vector Graphic service has been added. The service provides the support for the SVG 1.1 standard and allows the automatic generation of SVG compliant code.
- **Core services: phy service**
The Physics service has been added. The service provides the support for standard physics operations. In particular, the most common physical constants are defined in this service.

20. Release 2.0

- **Core engine: standard objects**
The BlockBuffer object has been added to the standard object library. Furthermore, the Buffer has been adapted to operate as a base class for the block buffer and the shl method has been added to the buffer object as a mean to shift the buffer. As consequence, the default operating mode for a buffer is the BYTE mode. When operating with strings, the UTF8 mode might be more suitable. The BitSet object has been renamed to Bitset and the interface has been cleaned. The Vector has been cleaned. The object-p predicate has been fixed.
- **Core engine: thread engine**
The thread engine has been completely redesigned and extensively tested on 32 and 64 bits platforms. It is no longer a problem to operate with more than 32K threads simultaneously. Furthermore, the concept of thread pool has been added to the engine. The end-p predicate has been added to the thread object to indicate a successful thread completion.
- **Core engine: form reader**
The Reader object has been added as a form reader. The reader parses an input stream and produces a form until the end-of-stream. The Reader provides the support for string based execution.
- **Core engine: default librarian module**
The Librarian object has been enhanced to support the concept of default execution module. When such module and when the interpreter is requested to do so, the module is automatically loaded during the execution.
- **Core modules: nwg module**
The HttpProto default version has been move to 1.1 for both the request and response objects.
- **Core modules: sio module**
The InputMapped class has been enhanced to provide the facility for mapping buffer as well as acting as a null character generator. The OuputBuffer object has been added as a buffer output stream. With the addition of a form reader, the interpreter communication class Intercom has been added to the standard i/o module.
- **Core modules: xml module**
The XneCond object has been enhanced to support various xml object. The XmlPi has been enhanced to support attributes derivation from the string value.

- **Core modules: itu module**
The itu module is a new module. It has been added with a complete support for the ASN.1 standard. ASN.1 is essential for the support of certificates.
- **Core clients: axs client**
The axs client has been removed from the core distribution. All of the client functionalities are now available in the spreadsheet module.

21. Release 1.9

- **Core engine: object unreference**
The long awaited unref reserved keyword has been added as a mechanism to unreference a symbol.
- **Core engine: object predicate**
The object-p predicate has been added as a standard predicate. The predicate is the negation of the nil-p standard predicate. The method-p predicate has also been added as a standard predicate.
- **Core engine: stop/resume parsing**
The file stream parsing has been enhanced with the help of the stop ◀ and resume ▶ characters. When the stop characters is found, all parsing operations are suspended until a resume character is found.
- **Core engine: extended exception attribute**
The about symbol has been added to the exception object as extended exception reason. For a given reason, the file name and line number is added to the exception reason.
- **Core engine: string objects**
The Strvec string vector class has been added to the core library. The class is similar to the Vector class except that it operates with strings and provides additional strings related methods.
- **Core engine: counter object**
The Counter object has been added as a reserved object. The counter is designed to be used directly in loop.
- **Core engine: library cleaning**
The core library has been extensively cleaned in preparation for the next major release. In particular, numerous memory leaks have been removed and some classes derivations have been fixed. A major bug in the closure argument counting has also been discovered and fixed during this process.
- **Core module: sio module**
The Pathname object has been enhanced to detect the type of path associated with the object. Additionally, a normalize method has been added.
- **Core module: sio module**
The FileInfo object has been added to the module. The class provides an immediate access to the principal file parameters such like it size or its modification time.
- **Core module: sio module**
The NamedFifo object has been added to the module. The class provides the support for a large string based fifo with file saving capabilities.
- **Core modules: nwg module**
Several predicates and functions related to media type conversion have been added to the module. In particular, a media type extension conversion has been implemented. The HttpResponse class has been enhanced with several methods for status code checking.
- **Core modules: sec module**
Support for the Digital Standard Algorithm, (aka DSA) as specified by FIPS-PUB

186-3 has been added to the library. The implementation incorporates several new objects to manipulate signatures.

- **Core modules: sec module**
The RC2 block cipher algorithm has been added to the module.
- **Distribution: documentation**
The documentation distribution rules have been rewritten and the "publish" rule has been added.

22. Release 1.8

- **Build process: reset rule**
The distclean top level makefile rule has been renamed as reset.
- **Core engine: stream object**
The stream engine has been cleaned with a new architecture. Two new objects `InputStream` and `OutputStream` acts as the foundation of this new design.
- **Core modules: nwg module**
The `HttpProto`, `HttpRequest` and `HttpResponse` objects have been completely rewritten. In the new model, both objects can operate on the server and client side. The `HttpReply` object has been removed.
- **Core modules: sec module**
The `Sha224` hash function has been added. This class concludes the implementation of all SHA family hash functions. The `Des` class that supports the DES stream cipher has been added to the library.
- **Core modules: xml module**
The `XmlRoot` class has been enhanced in order to ease the declaration node existence verification as well as the encoding mode extraction.

23. Release 1.7

- **Core clients: random engine seeding**
A new option controls the seeding of the random engine. By default, in debug mode, the random engine is not seeded unless requested by the user. In optimized mode, which is the normal mode, the random engine is seeded at initialization.
- **Core engine: base number object**
The long awaited base number object has been added. The `Number` object serves the `Integer`, `Real`, and `Relatif` objects. The base number object is designed to ease the task of formatting numbers.
- **Core engine: relatif number enhancements**
The `relatif` number object has been enhanced to support extra methods that are used for large number computation. This include the power and gcd computation which are used by the cryptographic engine. In addition, the base arithmetic `relatif` methods have been optimized and certain corner bugs in the division fixed.
- **Core engine: unicode database**
The core engine has been updated with the new Unicode 5.1.0 database.
- **Core engine: serious bugs**
A serious bug in the form synchronize engine that would cause an engine crash when a form is nil has been fixed.
- **Core modules: sio module**
A new object called `Pathlist` has been added to support the manipulation of path list. The object is designed to ease the file name resolution in the presence of search path. The module has also been extensively cleaned.

- **Core modules: mth module**

A new module called afnix-mth has been added to the standard distribution. The module is designed to integrate the base mathematical functions and objects available in the engine. With such introduction, the random number generation has been moved into this module. Additionally, the functions needed to generate prime numbers have been added to this module.

- **Core modules: sec module**

A new module called afnix-sec has been added to the standard distribution. The module is designed to integrate the security functions and cryptographic objects. Two new hasher objects have been added to the security module. The Md2 object implements the MD2 message digest algorithm as described in RFC 1319. The Md4 object implements the MD4 message digest algorithm as described in RFC 1320. The standard key derivation functions KDF1 and KDF2 have been added to the security module. The asymmetric cipher RSA has also been added to the security module and the Key object has been updated to reflect this.

24. Release 1.6

- **Core engine: object collection redesign**

The core engine has been seriously modified to accommodate for a new object collection system (aka garbage collection). The new system is more robust and provides new mechanism that will permit to reclaim cyclic structure as well as destroying global object on demand.

- **Core engine: macos x support**

The core engine has been adapted to support the new MACOS X Leopard operating system.

25. Release 1.5

- **Core engine: unicode 5.0 support**

The core engine continues to be updated in order to better support the Unicode 5.0 standard. With this release, the string normalization scheme is now in place and used by default internally. This implies among other things, a better support for multiple diacritics as well as the beginning of the standard collation algorithm.

- **Core engine: log file support**

The Logger base class has been enhanced to support the generation of a log file. An output stream can now be bound to the object.

- **Core engine: class defer support**

The concept of class defer object has been added to the Class object. The defer mode is the opposite of the infer mode and provides a mechanism for base class creation.

- **Core engine: print table header**

The PrintTable object has been enhanced to support the concept of table header.

- **Core engine: exception re-throwing**

The exception object what can be thrown with the reserved keyword throw. This provides a mechanism to re-throw an exception.

- **Core engine: critical bug with return form**

A critical bug in the core engine affecting the behavior of the return reserved keyword in a try block has been fixed. A return form inside a try block was incorrectly generating an exception which was subsequently caught by the try block.

- **Core modules: net module**
The base network module has been enhanced to better operate with IPV6. In particular, when both IPV4 and IPV6 stacks are present and a host name (typically localhost) have an address entry, the socket constructor make sure it can build an object. The IPV6 address display has been rewritten.
- **Core modules: sio module**
A new object called Pathname has been added to support the manipulation of system path. In addition, two new functions mkdir and mmdir have been also added to support the directory creation, both normally and hierarchically.
- **Core modules: nwg module**
The Uri has been dramatically enhanced to conform to the RFC 3986. In particular, the path representation for urn is now working properly. The cookie object has been massaged to support the cookie version 1, although it does not seem to be supported (yet!) by the browsers.
- **Core modules: xml module**
The xml module has been enhanced with a new parsing system called the simple model. In the simple model, nodes are parsed in a linear fashion. The node content is available in the form of a string and its interpretation is at the user discretion.
- **Core service: wax service**
The afnix-wam service has been renamed as afnix-wax. The service has also been updated with two new objects, namely the XmlMime and XhtmlMime which permits to build a mime representation of an xml tree. Several xhtml objects have also been added to complete the collection. This include the XhtmlScript for example.
- **Core service: xpe service**
The afnix-xpe service has been added as a new service. The xml processing environment (xpe) provides a xml processor that permits manipulate the whole xml tree with the help of various xml processor features. In particular, the service provides the support for the xml include extension.
- **Core projects: apx project**
This release incorporates for the first time, the concept of core project, which represents a librarian or an application. The first project is the AFNIX protocol extension or apx which is a message based protocol designed to transport request/reply messages within a client/server environment. The message is built with the xml library and the librarian provides the encapsulation layer.
- **Core projects: amd project**
The AFNIX media dumper or amd project is a complete application designed to illustrate the design of an application. The application permits to dump an uri content into a file.

26. Release 1.4

- **Core engine: unicode 5.0 support**
The core engine has been substantially changed to support the new Unicode 5.0 standard. As of now, the engine is in place internally, but not fully activated. In particular, the string normalization is implemented but not activated. The next release should incorporate the full system with a change that should be transparent to the user.
- **Core language: instance inference**
An instance inference mechanism – which is equivalent to the concept of virtual constructor – has been added to the core engine. Such system permits to derive top instance from a base instance construction.

- **Core language: print table object**
The PrintTable object has been enhanced with a dump method similar to the format method.
- **Core language: property list object**
The Property and the Plist objects have been added to the standard library. a property is name/value pair. The property list object is an iterable object that stores property objects.
- **Core modules: xml module**
A new module called afnix-xml has been added. The module provides the foundation for a full xml 1.0/1.1 support. The module also includes a parser that permits to build xml tree. A xml tree writer is also part of the module functionality. A xml processor is not yet available and is expected in the next release.
- **Core modules: nwg module**
A new module called afnix-nwg has been added. The module provides the support for the network working group objects such like Uri object. The module also provides the foundation for the mime support.
- **Core modules: web module**
The afnix-web module has been removed and replaced by the afnix-wam service.
- **Core service: wam service**
The afnix-wam service has been added as the first service into the core distribution. A service differs from a module in the sense that it is a combination of different modules. The web application management service depends on the xml and nwg modules. The service provides all the functionality to support a http session, including xhtml page generation and cgi request reply.

27. Release 1.3

- **Core language: ISO-8859 transcoding support**
The core engine has been modified to integrate a character transcoder that permits the support all ISO-8859 codesets which are mostly used for the encoding of european and arabic characters. Depending on the locale settings, the transcoder automatically remaps the 8 bits characters into their respective unicode character. All clients have been updated to detect their associated locale and to set automatically the appropriate transcoder. A new option -e has been added to force a particular encoding.
- **Core language: Logger base class**
A logging base class has been added. The logging facility provides the interface to store messages by time and level. This class is further extended in the modules.
- **Core language: Heap class**
A heap class has been added. The heap can operate in ascending or descending mode. This class can be used to support priority queue.
- **Core language: Option class**
An option class has been added in order to ease the option capture when designing an application. The class permits to define the valid options and offer a powerful retrieval mechanism.
- **Core language: Date class**
The Time class has been completely changed and a new Date class has been added. The new mechanism provides a better separation between the time and the date, increase the date range and authorizes the support for multiple calendar.
- **Client: cross spreadsheet client**
The axs client has been modified to support the axs:insert-marker, axs:insert-header and insert-footer control commands.

- **Core modules: spreadsheet module**

The Folio and Sheet classes have been substantially updated to support additional information. The Sheet also supports the concept of markers that marks the sheet columns by literals. The concept of column tagging has been added with the associated search methods. All classes also contain an information field. The importation mechanism now supports a cons cell that defines both the cell name and the cell value.

- **Core modules: web module**

The Table class has been modified to support the concept of table data header. The associated methods have been added to the class and a new HtmlTh has been added. The concept of tag propagation has also been added. If a tag element already exists, this one is not added. This is particularly true for the class tag that is now part of the class constructor. The HtmlPage class has been put in strict conformance with xhtml 1.1 and the XHtmlpage class has been removed.

- **Core modules: pim module**

A new module called the afnix-pim module has been added to the base distribution. The personal information management or pim module is designed to ease the management of personal information and agenda.

- **Core modules: gfx module**

A new module called the afnix-gfx module has been added to the base distribution. The module contains the base class that supports the graph data structure which was previously part of the standard library.

28. Release 1.2

- **Core language: Unicode support**

The core engine has been substantially modified to integrate the support for Unicode characters. Depending on the system settings the reader automatically adjust itself to operate in byte mode or in UTF-8 mode. The String and Character classes are now operating with a Unicode representation. The design of an Unicode based engine also impacts several classes like the Regex, Buffer and stream classes. A new class called Byte is also designed to handle byte character. A new stream model with a base Stream class has also been added. The full support with Unicode character is not yet completed. In particular, certain codesets are not supported at all. This is particularly true with case-conversion functions.

- **Core language: orphan instance and reparenting**

The object model now supports the creation of orphan instance which is an instance without a class attached to it. The instance can be later bound to a class and such class can even be changed during the course of the program execution.

- **Core modules: network module**

The Address class has been updated to reflect the access to address aliases.

- **Core modules: text processing module**

The Literate class has been updated to reflect the support of Unicode characters. The class can operate both in byte mode or in Unicode character mode.

29. Release 1.1

- **Core language: Large file support**

Support for the large file system has been added in the base distribution. All input/output operations as long as they are supported by the operating system are now done in 64 bits mode.

- **Core libraries: Secure hash algorithm**
The cryptographic library incorporates the support for the SHA-1, SHA-256, SHA-384 and SHA-512 hash algorithms.
- **Core libraries: Standard symmetric cipher**
The cryptographic library incorporates the support for the Advanced Encryption Standard (AES) as a symmetric cipher.
- **Core libraries: ODC library renamed**
The ODC library has been renamed to SPS which stands for spreadsheet library. This new name is considered more appropriate for the function the library achieves.
- **Core libraries: xhtml 1.1 support**
The XhtmlPara class is now configured to support XHTML 1.1 with utf-8 encoding.
- **Documentation: XML based documentation**
The documentation has been rewritten completely in XML. A DTD as well as the necessary XSLT style sheets have also been designed to produce a professional documentation which can be used for printing or for online browsing.

30. Release 1.0

The 1.0 release is the initial release. This release replaces the old *ALEPH programming language* which has been discontinued.

- **04/19/2005: release 1.0.3**
This release incorporates the necessary files that support GCC 4. It also provides some minor fixes that were preventing the compilation on some 64 bits platforms.
- **03/02/2005: release 1.0.2**
This release incorporates a minor fix that could cause the build process to fail.
- **02/16/2005: release 1.0.1**
This release incorporates a minor fix that could cause the build process to fail.
- **01/16/2005: release 1.0.0**
This is the primary release 1.0.0 which originated from the ALEPH programming language and which has been discontinued. A complete history of the language is provided in the description page.

Index

- * , 12, 19
- + , 12, 19
- , 12, 19
- ., 44
- ... , 8, 44
- ..., 8
- / , 12, 19
- ;, 8
- #, 4

- about, 44
- abs, 20, 23
- Accuracy, 24
- add, 31, 32
- advice, 58
- alpha-p, 25
- and, 12
- args, 7, 9
- Argument binding, 7
- Arguments, 9
- Arguments vector, 46
- argv, 46
- asin, 23
- assert, 12
- axl, 48

- Binding, 8, 9, 14
- Bitset, 34
- blank-p, 25
- block, 12
- Block form, 5
- Boolean, 12
- boolean-p, 13

- car, 31
- cdr, 31
- ceiling, 23
- Cell, 14
- Character, 24
- character-p, 13
- Class, 14, 35
- class, 14, 35
- class-p, 13
- Closed variables, 55
- Closure, 6, 35
- closure-p, 13
- Combining character, 26

- Command arguments, 3
- Comments, 4
- Compile client, 4
- Completion, 17
- Condition variable, 17, 54
- Condvar, 17, 55
- Cons, 14, 31
- const, 4, 7–9
- Constant argument, 7
- Container, 31
- cos, 23

- defer, 41
- delay, 16, 45
- Delayed evaluation, 45
- digit-p, 25
- do, 10
- dup, 48
- Dynamic binding, 56

- eid, 44
- enum, 45
- eol-p, 25
- eos-p, 25
- eval, 11
- eval-p, 13
- Evaluation, 5
- even-p, 20
- Exception, 15, 43
- exp, 23

- Factorial, 6
- false, 12
- Fibonacci, 20
- File loading, 3
- File resolver, 50
- fill-left, 24, 26
- fill-right, 26
- floor, 23
- for, 15, 33
- force, 16, 45
- format, 24
- Forms, 5
- Function expression, 55

- Gamma expression, 6, 14
- get, 32
- get-enum, 45

- get-object, 15
- Hash code (String), 27
- Hello world, 2
- if, 10
- infer, 41
- Inheritance, 39
- insert, 32
- Instance, 14, 35, 36
- Instance deference, 41
- Instance inference, 41
- Instance method, 37
- Instance operator, 38
- Instance re-parenting, 40
- instance-p, 13
- Integer, 19
- Integer arithmetic, 19
- Integer calculus, 20
- Integer format, 19
- integer-p, 13
- Interactive mode, 2
- Interp, 18
- interp, 46
- Interpreter, 18
- Interpreter loop, 48
- Interpreter object, 46
- Interpreter options, 2
- Interpreter version, 47
- Item, 45
- iterable, 33
- Iteration, 14
- Iterator, 15
- iterator, 33
- Lambda expression, 5, 6
- launch, 16, 50
- Legendre polynomial, 23
- length, 26, 33
- Lexical name, 56
- lexical-p, 13
- Librarian, 48
- library, 48
- Line editing, 3
- link, 33
- List, 32
- Literal, 19, 21
- literal-p, 13
- load, 47
- Locking, 51
- log, 23
- Logger, 46
- Logger message, 46
- loop, 11
- map, 33
- meta, 37
- method-p, 13
- mute, 40
- Nameset, 8
- nameset, 44
- nameset-p, 13
- Native objects, 4
- nil-p, 13, 25
- not, 12
- Number, 19, 21, 22, 24
- number-p, 13
- object-p, 13
- odd-p, 20
- or, 12
- Parsing, 4
- Predicate, 13
- preset, 14, 36, 41
- promise, 45
- promise-p, 13
- protect, 11, 56
- Qualified name, 8, 56
- qualified-p, 13
- Queue, 34
- Real, 22
- Real functions, 23
- real-p, 13
- reason, 44
- Regex, 16
- regex-p, 13
- Regular expression, 16
- Relatif, 21
- relatif-p, 21
- return, 11
- roll, 48
- scalar-product, 33
- self, 55
- Self reference, 55
- Set, 32
- set, 32
- set-absolute-precision, 24
- set-car, 31
- set-relative-precision, 24
- Shared objects, 51
- sin, 23
- Special form, 5, 9
- split, 26
- String, 26
- string-p, 13
- strip, 26
- strip-left, 26
- strip-right, 26
- sub-left, 26
- sub-right, 26
- super, 39
- switch, 11
- Symbol, 5, 8
- Symbol access, 57
- Symbol unreferencing, 9
- sync, 17
- Synchronization, 17, 52
- System name and type, 47
- tan, 23

this, 14, 36
Thread, 16, 50
Thread completion, 52
Thread object, 51
Thread set, 51
thread-p, 13
throw, 15, 43
to-hexa, 21
to-hexa-string, 21
to-literal, 21
to-lower, 27
to-string, 21
to-upper, 27
trans, 4, 7, 8
true, 12
try, 15, 43

unref, 9

valid-p, 15
Vector, 32

what, 15, 43
while, 10
Writing structure, 4

►, 4
◄, 4