# Genyris User Manual

June 2, 2010

Build 671

Peter William Birch

birchb@genyris.org

**Abstract**

This document is the user manual for the Genyris scripting language.

# Contents

**Conventions used in this document:**

*italicized* Snippets of Genyris programs

`fixed-font` Larger programs are in a fixed font. Interactive sessions are show with the > prompt of the command-line interpreter and the results printed underneath.

# Part I
# Tutorial

## 1 Getting Started

### Getting Genyris

Genyris can be downloaded from the sourceforge project "Download" pages as a binary release. The project is hosted here: *http://sourceforge.net/projects/genyris/*.

### Installation

Genyris is available as a *ZIP* file. This file needs to be unpacked into an empty directory such as "genyris". This becomes the home directory for Genyris.

Genyris is developed in Java hence needs a Java runtime. You don't need to understand Java to use Genyris. However you will need the Java 1.6 JRE or later to run the Genyris interpreter. Java can be downloaded from Sun Microsystems. Check your JRE version with this command:

```
$ java -version
```

First change directory to the genyris home, then you can start the Genyris command-line interpreter with this command:

```
$ java -jar dist/genyris-bin-nnn-xxxxxxxxx.jar
```

Where *nnn-xxxxxxxx* is the version number. You will see a prompt indicating the interpreter is ready for your input:

```
*** Genyris is listening...
>
```

Refer to section (16) for details on how to simplify running Genyris using PATH settings.

### Executing Expressions

Genyris commands can now be typed at the prompt, use two carriage returns (↩) to terminate a statement. For example to add two numbers type:

```
> + 42 37 ↩
↩
```

Genyris responds with the answer and a comment about the result:

```
~ 79 # Bignum
```

### Verifying the Install

To test the installation run the self test suite with the following command:

```
> self-test-runner↩
↩
```

All being well, it will print "OK" and the number of tests passed.

### Running Examples

The release binary file includes some examples in the *examples* folder. The files can be edited with your favourite text editor and run with the *include* function. For example, to load an run the "Eight Queens" example do:

```
> include 'examples/queens.g'

~ 'file:/home/birchb/workspace/Genyris/examples/queens.g' # String
> run-queens 8↩
↩
```

## 2 Syntax

The syntax of Genyris uses indentation to convey program structure. This is in common with other languages such as Python. However Genyris preserves the "prefix" notation of Lisp and Scheme. Here is an example of some code defining a function:

```
def threat (i j a b)
   or
      equal? i a
      equal? j b
      equal? (- i j) (- a b)
      equal? (+ i j) (+ a b)
```

Instead of curly braces or *begin* and *end* tokens, the indentation defines the blocks of code. Genyris reads lines one-by-one until it reaches the end of an expression. An expression ends when there are no more indented lines. The interactive command-line ends an expression whenever two blank lines are read. Within a line, tokens are separated by white-space. Genyris recognizes the following syntactic elements:

- Comments
- Numbers
- Strings

- Symbols
- Sub-expressions
- Lists (Pairs)
- Parser macros and directives

## 2.1 Comments

All characters following a hash (pound) until the end of the line are ignored by the parser. For example:

```
# This whole line is a comment
- 4 3    # this comment goes to the end
```

## 2.2 Numbers

Numbers can be either integers or floating point with any number of leading or trailing digits. Examples:

```
-3 23.78 -100.0089  34.45e7
```

## 2.3 Strings

Strings are delimited by either double quote characters " or single quotes '. Within a string quotes and special characters are escaped with backslash \. For example *"She said \"sea shells\""* "yields the string:

```
She said "sea shells"
```

Other escape sequences are encoded as follows:

| | |
|---|---|
| \n | New Line |
| \r | Carriage Return |
| \f | Form Feed |
| \\ | Backslash |
| \t | Tab |
| \' | Quote |
| \a | Bell |

The two styles of string are internally identical, which allows you to avoid escaping. For example this string contains double quotes:

```
'"After all," said the young man, "golf is only a game."'
```

## 2.4 Symbols

Symbols are a group of any printable characters with the following exceptions:

| | |
|---|---|
| , | comma |
| ^ | carat |
| . | period |
| ' | single quote |
| " | double quote |
| ` | backquote |
| @ | at sign |
| [ | square |
| ] | brackets |
| { | curly |
| } | brackets |
| = | equal sign |
| ! | exclamation |

The following are all examples of valid symbols:

```
Wednesday-12
_age
*global*
+$
<variable-name>
```

In Genyris symbols are "interned" by the parser so that there is only ever one instance of a particular symbol. Symbols are case sensitive so for example *Kookaburra* and *kookaburra* are different symbols.

## 2.5 Expressions and Sub-expressions

All Genyris expressions are parsed and stored as linked-lists. A single line is converted into a single list. Sub-expressions are denoted in two ways, either within parentheses on a single line, or by an indented line. For example the following line contains two sub-expressions:

```
Alpha (Beta Charlie) (Delta)
```

Sub-expressions made using parentheses must remain within a single line, they are not permitted to wrap. Indented lines are deemed to be sub-expressions of the superior, less indented, lines above. The above expression can be written in indented form as follows:

```
Alpha
   Beta Charlie
   Delta
```

Indentations must line up with previous indentations of the same level as follows (spaces indicated with periods):

```
Alpha
...Beta Charlie
......Delta
...Beta              # correct indentation
```

The parser is unable to cope with random indentation levels since it does not know what depth is required. The following example will generate an error:

```
Alpha
...Beta Charlie
......Delta
....Beta              # *** ERROR
```

## 2.6 Pairs

Within Genyris lists are composed of pairs of references to objects[1]. Pairs have two elements, the left and right, which are references to other Genyris objects. The left and right halves of a *Pair* can be delimited with the equals = character, an infix operator. For example:

```
(1 = 2)
```

denotes a *Pair* referring to the numbers 1 and 2. Genyris expressions are also composed of linked lists of *Pairs*, hence the expression:

```
(A B C D)
```

is shorthand for :

```
(A = (B = (C = (D = nil))))
```

Lists are terminated with the special symbol *nil*. An indented expression can be expressed in terms of *Pairs*. Consider:

```
Alpha
   Beta
```

This is the same as

```
(Alpha = ((Beta = nil) = nil))
```

Lists do not always have to be terminated with *nil* , the colon = operator can be used to squeeze one more object reference into the end of the list. For example the following list has *C* instead of *nil*:

[1] Lisp Cons cells

```
(A B = C)
```

New pairs can be created explicitly with the cons function which takes two parameters - the left and right parts of a new pair.:

```
> cons 123 456

123 = 456 # Pair
```

## 2.7 Quoting and Special Parser Characters

Lists and atoms can be quoted in Genyris. Quoting is used to prevent execution of expressions. A single atom can be quoted within an expression[2]:

```
list 1 2 ^a 3 4    # evaluates to: (1 2 a 3 4)
```

Carat characters are a shorthand notation to save typing. When the parser sees a carat, it collects the expression following and wraps it within a *quote* expression. So ^<*exp*> becomes *(quote <exp>)*. When the *quote* function is evaluated it does not evaluate its argument. So the above expression is actually:

```
list  1 2 (quote a) 3 4
```

Embedded lists can be quoted, in which case the embedded list is not evaluated:

```
func 1 2 ^(x y z) 3 4
```

If the quote falls at the beginning of the line, only the first element is quoted, not the entire line. So:

```
list 1 2
    ^x y z
```

is the same as:

```
list 1 2 ((quote x) y z)
```

To allow entire sub-trees to be quoted, the quote function needs to be used as in this example:

```
list 1 2
   quote
      x y z
```

which is the same as:

```
list 1 2 (quote (x y z))
```

[2] *list* is a function we will cover later.

## 2.8 Line Continuation

Sometimes long expressions become unwieldy and must be continued on following lines. There are two mechanisms for this. This first is to use the equal operator and an indented line as follows:

```
list 1 2 4 5 =
   6 7 8
```

This is equivalent to:

```
list 1 2 4 5 = (6 7 8)
```

which is the same as:

```
list 1 2 4 5 6 7 8
```

The second and preferred is the special line continuation character (the tilde ~) which continues the previous line indentation level at the start of the line under which it is placed. This allows arbitrary continuations such as:

```
^
  1 2
      3
      ~ 22
      99
```

which is the same as:

```
^ (1 2 (3) 22 (99))
```

if the tilde was not there the expression would become:

```
^ (1 2 (3) (22) (99))
```

## 2.9 More Quote Characters

Genyris also supports three other special syntactic quotes similar to the carat. They are all used to simplify writing macros with the *template* function, but can be used for anything else. These are converted by the parser into expressions as follows:

| **Input Quote Sequence** | **Translated Expression** |
|---|---|
| ,<exp> | (comma <exp>) |
| ,@<exp> | (comma-at <exp>) |
| ‘<exp> | (template <exp>) |

### 2.10 Square and Curly Brackets

The square [] and curly {} brackets are parsed specially so they can be used by the user in the expressions. In both cases they are converted to a list with a symbol as follows:

| **Input Sequence** | **Translated Expression** |
|---|---|
| *[<list>]* | *(squareBracket <list>)* |
| *{<listL}* | *(curlyBracket <list>)* |
| [1 2 3] | (squareBracket 1 2 3) |
| {a b c} | (curlyBracket a b c) |

# 3 Variables

New variables are created with the *defvar* or *define* functions. These functions also take an initial value for the variable:

```
define name 'William'
defvar ^name 'William'
```

In both examples, the symbol *name* is bound to the value *"William"* in the current environment. After the variable has been bound, its value can be used in any expression in the scope. The *define* function has an alias *var* which is quicker to type:

```
var name 'William'
```

When the interpreter sees a symbol in an argument list it looks for a binding in the current environment and all parent environments right up to the global execution environment. If you define a variable at the command line, it is bound in the global execution environment and hence is available everywhere. If you try to access a variable when there is no binding, an "unbound variable" error will be reported.

Variable values can be updated with the *set* or *setq* functions, for example:

```
set ^name 'William Pitt'
setq name 'William Pitt'
```

As shorthand for *setq* the pair operator = can be used as in:

```
name = 'William Pitt'
```

A predicate function *bound?* is provided to test whether a symbol has a binding in the current environments. It returns the symbol *true* if the variable is defined otherwise *nil*.

# 4 Functions

As we have seen, Genyris can execute statements immediately at the command line. The expression:

```
+ 42 37
```

Yields the addition of the two numbers (79). Let's explore how this works. The interpreter looks for list expressions and assumes the first token (or sub-expression) is a procedure. The rest of the list constitute the arguments to the procedure. In this case + is a symbol which yields a procedure object. The arguments are also evaluated and the results are passed to the procedure to be evaluated. Lets have a look at + by getting its value:

```
> the +

~ <org.genyris.math.PlusFunction> # EagerProcedure
```

The function *the* is the identity function - it simply returns the value of its argument. Since the symbol + is an argument to *the,* its value is the underlying procedure. A "Procedure", or "Closure", is an object which keeps a reference to the environment in which it was originally defined and the executable code to be run when called. In addition it knows how its arguments are to be handled before the executable code is run.

## 4.1 Eager Functions

Eager functions are the default in most programming languages. These evaluate their arguments prior to applying the underlying procedure. Mathematical functions such as + - * and / are eager functions. Let's experiment with some simple math function calls. All the following expressions evaluate to 12:

```
+ 6 6
+ (* 2 3) (+ 2 4)
+ 2 2 2 2 2 2
```

Notice that the + function can have many arguments. Another function that takes multiple arguments is *list*. This function constructs a list from its arguments. Here's an example:

```
> list (* 34 8) 'pears' (/ 34 5) 'kilos'

272 'pears' 6.8 'kilos' # Pair
```

Note that the interpreter always prints a comment after the result. This comment is the list of classes the result belongs to. Since *list* returns a list, which is composed of Pairs, *"Pair* "is printed.

## 4.2 Lazy Functions

In contrast to Eager functions, Lazy functions do not evaluate their arguments. In other words, the interpreter passes the **source code** of their arguments to the function. This allows the function to defer evalution or even exclude evaluation altogether, as is the case in conditional (flow control) constructs. The *quote* function is a lazy procedure which returns its single argument un-evaluated. Example:

```
> quote (* 5 6 7 8)
(* 5 6 7 8) # Pair
```

Notice that no multiplication has been performed here and the first paraeter expression is returned.

## 4.3 Defining Your Own Functions

Functions in Genyris are defined in the usual way for functional programming languages. The *def* function binds a name to a lexical closure containing the current environment and the code to be applied in future calls. The body of the function is a sequence of expressions to be executed in the lexical environment, the last expression's value is returned. Here's a definition of the identity function:

```
def identity (arg) arg
```

Genyris has two kinds of user-defined functions 'eager' and 'lazy'. An eager function evaluates its arguments before it applies them, whereas a lazy function does not. Traditional functions such as '+' and *the* are eager. *list* is an eager function which returns all its arguments in a list.

Here is a more complex function definition:

```
def factorial (n)
   if (< n 2) 1
      * n
        factorial (- n 1)
```

The *if* function is lazy, since, depending on the value of the first argument, it executes only one of its other two arguments . In fact, *if* is a macro - a special kind of lazy function which we introduce later.

# 5 Control Flow

## 5.1 Conditional Execution

### 5.1.1 cond

The *cond* function is a lazy function that allows program flow to change depending on the outcome of conditional expressions. Here's the syntax of *cond*:

```
cond
   (<condition 1>)
      <sequence 1>
   (<condition 2>)
      <sequence 2>
   ...
   (<condition N>)
      <sequence N>
```

Each condition is evaluated in turn until one returns which is not *nil*. The associated sequence is evaluated and the value of the last expression in the sequence is returned. If there is no sequence, the value of the condition is returned. Typically the last condition is a non-nil constant and its sequence is the default. The symbol *else* is provided for this purpose. Here's an example:

```
cond
   (equal? foo 1)
      'One'
   (equal? foo 2)
      'Two'
   else
      'Other'
```

The function *equal?* returns *true* if the two arguments are the same otherwise *nil*. So if the symbol *foo* is bound to the value 2 this expression will return *'Two'*.

### 5.1.2 *if*

The *if* statement is a shortened form of the *cond* fuction, it only allows a single condition with the following syntax:

```
if <condition>
   <evaluated if condition is not nil>
   <evaluated if is nil>
```

It could be called the 'if-not-nil' statement because it's action depends on whether the condition expression is *nil* or not. Example:

```
if (< n 2) 1
      * n
        factorial (- n 1)
```

## 5.2 Looping and Iterators

### 5.2.1 Simple looping with *while*

Syntax:

```
while <expression>
    <body>
```

Example:

```
define names ^(Church McCarthy Russell)
while names
   print names
   names = (right names)
```

outputs:

```
Church McCarthy Russell
McCarthy Russell
Russell
```

### 5.2.2 Creating Iterators with .mkIterator

Syntax:

```
<expression>(.mkIterator)
```

Example:

```
define L ^(1 2)
define iter (L(.mkIterator))
assert
   equal?
      list (iter) (iter) (iter)
      ^(1 2 sys:StopIteration)
```

### 5.2.3 Traversing with *for*

Syntax:

```
for <var> in <expression-with-.mkIterator>
    <body>
```

List example:

```
define n ^(1 2)
define z ^(A B)
for v1 in n
   for v2 in z
        u:format "%s %s\n" v1 v2
```

prints:

```
1 A
1 B
2 A
2 B
```

A dictionary example:

```
var D (dict (.a=1)(.b=2))

for v in D
   u:format "%s = %s\n" v (D (symbol-value v))
```

prints:

```
.a = 1
.b = 2
.self = (dict (.a = 1) (.b = 2))
.vars = (.a .b .self .vars .classes)
.classes = (<class Dictionary (Builtin)>)
```

#### 5.2.4 Ranges

Syntax:

```
range <low> <high>
```

Example:

```
> var R (range 1 3)
(1 3) # Range Pair
> var iter (R(.mkIterator))
<EagerProc: <anonymous lambda>> # RangeIterator EagerProcedure
> iter
1 # Bignum
> iter
2 # Bignum
```

An example with *for:*

```
> var array ^(A B C D E F)
> for i in (range 3 5)
     print (nth i array)
~ D
~ E
~ F
```

## 5.3 Exceptions

The Genyris interpreter signals errors via exceptions. Exceptions are 'raised' by the code that detected the error. When the interpreter is told to raise an exception it unrolls the execution stack until either the exeption is caught by a *catch* statement, ot the program halts. In some cases (perhaps due to a bug in the interpreter, or terminal conditions such as out of memory, the program prints the internal java stack and halts.) Sinec the interpreter has to do a lot of work to pop the stack, exceptions are not an efficient mechanism and are desogned for handling exceptional circumstances.

### 5.3.1 Popping the execution stack with *raise*

To signal an error in a Genyris program, and to ask the interpreter to unroll the stack, use the *raise* function. This takes an object as its single argument which is passed up to the stack and is given to any *catch* functions which are above it in the stack. This way, details of the state of the system can be passed to error handling routines.

Syntax:

```
raise <exception object>
```

For example, here we raise an exception in a base class if an abstract method is used:

```
def .toMeters()
   raise "Oops - you invoked an abstract method."
```

### 5.3.2 Catching exceptions with *catch*

The *catch* function executes a sequence of expressions, if no exception is raised it returns the value of the last expression. If an exception is raised during evaluation of the expressions, the exception object is assigned to a nominated variable. Without this function all errors would cause program execution to halt.

Syntax:

```
catch <variable name>
   <expression>
   <expression>
   ...
```

Here is a simplified top-level command line interpreter loop by way of example. Expressions are read from the keyboard and executed. If any exceptions are raised by the system or by the *raise* function they are caught and printed. Execution resumes. If the *catch* function was not used, the interpreter would stop execution of the loop at the first exception:

```
while true
   display '\n>> '
   catch errors
       define expression (read)
       define result (eval expression)
   cond
       errors
          print errors
       else
          print result
```

### 5.3.3 Inserting program checks with *assert*

Often it is useful to insert checks in programs for invariant conditions that must be met. The *assert* macro provides a handy way to insert checks. If the expression passed to assert for evaluation is not *nil*, an exception is automatically raised including the source code of the expression. Syntax:

```
assert <expression>
```

Example execution:

```
> assert (equal? 1 2)
*** Error - ("assert failed on expression: " (equal? 1 2))
~ '<LazyProcedure: cond>'
~ '<LazyProcedure: <assert>>'
~ '<EagerProc: eval>'
~ '<LazyProcedure: <define>>'
~ '<LazyProcedure: catch>'
~ '<LazyProcedure: while>'
~ '<EagerProc: <|http://www.genyris.org/lang/system#read-eval-print-loop|>>'
```

# 6 Advanced Functions

## 6.1 Anonymous Functions with function and Friends

Actually the *def* and *defmacro* functions are lazy functions. They bind a variable name to procedure compiled from the function body. But what if we want a function without the binding? Genyris provides three kinds of in-built procedure-building functions. The function *function* (aka *lambda*) creates a user-defined eager procedure object which is a closure at the point of definition. For example we can create an anonymous function at the command-line:

```
> function (x) (* x x)

~ <org.genyris.interp.ClassicFunction> # EagerProcedure
```

To actually call it we place it wherever a function is expected, such as a parameter to a function, or at the beginning of a list:

```
> (function (x) (* x x)) 3

~ 9 # Bignum
```

Notice the parentheses are required around the expression to trigger the execution. The argument 3 is passed to the resulting closure. Functions are 'first class' and can be assigned to variables, which is how *def* works. The following two expressions are equivalent:

```
define square
   lambda (x) (* x x)

def square (x)
   * x x
```

### 6.1.1 Anonymous Lazy Functions

To defer evaluation, a lazy function can be defined using the *lambdaq* or *lambdam* macros. *lambdaq* is just like *function* except it builds a lazy procedure, *lambdam* builds anonymous macros. The next example creates an anonymous function which prepends its argument (without evaluation) to a list:

```
> (lambdaq (x) (list x 'World')) (+ 'Hello')

(+ 'Hello') 'World' # Pair
```

## 6.2 Executable Comments

Sometimes it's necessary to temporarily 'comment out' functions or provide inline text. This is done via the function *'//'* which is a lazy function that ignores it's arguments and return *nil*. It is defined as:

```
df // (&rest body)
```

and can be used as follows:

```
// def allcommentedout(arg) # to comment out an entire function
  function (x)
    cond
      (eq? nil (cdr x))
         car x
      else
         last (cdr x)
def myfun()
```

```
// 'in-line  documentation'
+
   // + 1      # however this causes an error because // returns nil.
       + 34
       + 45
   99
```

## 6.3 Defining Macros

Macros are lazy functions which are very handy for extending the syntax of the language or creating Domain-Specific Languages (DSLs). Macros re-evaluate the returned value in the environment of the caller. Here's an example:

```
defmacro trace(&rest body)
  print body
  body
```

This macro prints an expression which is then evaluated. The keyword *&rest* tells the interpreter to collate the values of all remaining arguments into the single variable *body*. So when called with:

```
> trace (+ 1 2)
```

It prints the expression and the result is calculated:

```
(+ 1 2)
~ 3 # Bignum
```

Here is a more complex example - definition of a control flow function:

```
defmacro my-if (test success-result failure-result)
   template
      cond
         ,test ,success-result
         else ,failure-result
```

This macro uses the *template* function and *comma* to splice the arguments into a formulaic expression. Here's an example if its use:

```
define test 3    # binding in the caller's environment
my-if (equal? test 3) 1 2
```

This returns 1. Notice how the variable *test* is defined in the caller's environment used in the condition, not the binding of the same name within *my-if*.

## 6.4 Lexical Scoping Captures Environments

Genyris is "lexically scoped" - when a function is defined it remembers the environment in which it was defined and re-uses that environment when it executes. This provides a way of hiding data and giving functions stateful side effects. The following example[3] creates a function which captures the *balance* variable:

```
def make-withdraw (balance)
   function (amount)
      setq balance (- balance amount)
define W1 (make-withdraw 100)

> W1 25
W1 25

~ 75 # Bignum
~ 50 # Bignum
```

Repeated execution of the function *W1* reduces the value of the balance each time. The sequence of evaluation is as follows:

1. the lazy *def* expression is executed which results in a procedure object bound to the symbol *make-withdraw*
2. the *balance* argument (*100*) to the eager *define* expression is evaluated and *make-withdraw* is called.
3. the eager *make-withdraw* creates a new environment in which it binds *balance* to *100*
4. the body of *make-withdraw* is evaluated resulting in another procedure object which captures a reference to *balance* and contains the executable code starting with *setq*
5. the procedure object is bound to W1
6. *W1* is called with the argument 25
7. the procedure *W1* subtracts 25 from the *balance* binding in the environment created in step 3

Note that there is no way to directly access the *balance* variable.

[3] refer to Abelson and Sussmans' book "Structure and Interpretation of Computer Programs"

# 7 Everything is Callable

The Genyris evaluator expects the first element of a list to be some kind of procedure object - something that can compute its arguments and apply them. This is the role of traditional functions such as + or user-defined functions. In Genyris, all objects are callable, even atomic types. For example an integer can be called as a function thus:

```
> 12 (+ 33 44) (- 4 3)

~ 1 # Bignum
```

Lets analyse what happens. The integer *12* was called with two argument *(+ 33 44)* and *(- 4 3)*. *12* is a lazy function and does not evaluate its arguments. It treats its arguments as a sequence of expressions to be evaluated in a new environment. So it calculated 333 + 44 = 77, and then 4 -3 = 1. When it reached this last expression it returned the value 1. This expression can be written in indented form with the same result as follows:

```
12
   + 33 44
   - 4 3
```

If an atom is called with no arguments, it simply returns itself. So at the command line typing a number alone returns the number:

```
> 1024
~ 1024 # Bignum
```

As well as executing the sequence of expressions, an execution environment was created in which the number 12 is a bound to the dynamic variable *.self*. The variable can be used as follows:

```
> 12 (+ .self .self)

~ 24 # Bignum
```

Here the number is added to itself. The environment can also be used to create local bindings with the *define* functions:

```
12
  define foo 987
  + foo .self

~ 999 # Bignum
```

If a symbol is called as a function it is by default evaluated to locate the binding in the current evironment. If we make the value an atom, we can use the symbol as a a keyword. For example, the symbol *my-do* could be defined like this:

```
define my-do ^my-do
```

Now whenever we call *my-do* as a function, it acts as a code block which can be used in a function:

```
def my-function()
   my-do
      some-function 'Hi!'
      define a-variable 42
      print .self
```

However there is a catch - within the context the do block of *.self* is bound to *my-do*. Hence the above function prints "my-do". A better way to add new syntax is to create a macro, since *.self* is not affected.

## 7.1 Dynamic Variables

'Dynamic' variables are those which are bound in the environment of the caller and hence depend on who is evaluating the expression. In Genyris dynamic variables are limited to being properties of the currently called object, and called objects are part of their environment. In other words when an object is used as a procedure, the environment created to make the call is a merge of the object itself and a lexical environment. When prefixed with the period . character, the binding for the symbol is looked up in the dynamic context. An example will make this clearer:

The number 12 above has two dynamic variables, *.self* and *.classes*. They can be accessed as follows:

```
12
  print .self .classes
```

Here we see that the *.classes* variable is referring to the class list of *12*. It has a single class, *Bignum,* which is printed. This behaviour is the same for the other atomic types: Bignums, Pairs and Strings. Consider the following examples:

```
> 'What am I?'.classes

<class String (Builtin)> # Pair
```

However where symbols are concerned, the evaluator always looks up the value binding. So to work with a symbol we must first quote it:

```
> ^a-symbol.classes

(<class SimpleSymbol (Symbol)>) # Pair
```

Likewise the interpreter assumes a list is a normal function call so a quote is needed to see this behaviour:

```
> ^(3).classes
<class Pair (Builtin)> # Pair
```

Most atomic types have only a single dynamic variable, richer examples lie in the compound object types.

# 8 Working With Pairs and Lists

Like its forbears Lisp and Scheme, Genyris is a list-processing language - its source code is expressed as lists and it has inbuilt functions for parsing and manipulating list data. Since programs and data are stored in the same form, Genyris is an ideal platform for developing DSLs or even new programming languages. Happily, manipulating lists is easy. Lists are a kind of binary tree. Trees are constructed with the *cons* function which accepts two arguments for the left and right halves of the *Pair*:

```
> cons 'A' 'B'
('A' = 'B') # Pair
```

Note the interpreter prints a colon between the left and right halves of the Pair. The individual elements of a Pair can be accessed with the *left* and *right* functions:

```
> left (cons 'A' 'B')
'A' # String
```

Alternatively the dynamic variables *.left* and *.right* can be used when the list is called:

```
> var my-pair (cons 'A' 'B')
my-pair .right
'B' # String
> my-pair
   setq .left 33
   .self

33 = 'B' # Pair
```

To construct a proper List, the final right hand element will be *nil*:

```
> cons 'A'
    cons 'B'
       cons 'C' nil
'A' 'B' 'C' # Pair
```

The printing of trees (by default) assumes that the tree is a kind of list, hence you don't see the parentheses in this case. See how the interpreter identified the list as a *Pair*, since it only has a reference to the first *Pair*.

To help view *Pairs* explicitly, a list can be tagged with the PRINTWITHEQ class, which forces the printer to display the full tree structure. The parser does this automatically, so Pairs which the user types with a colon are printed the same way. For example:

```
> ^('A' = ('B' = ('C' )))
('A' = ('B' = ('C'))) # Pair
```

# 9 Dictionaries - Your Everyday Objects

Genyris provides "dictionary" objects which are "objects" in the normally understood sense for programming languages. Each dictionary provides a un-ordered set of dynamic symbols and bindings - called "properties". A dictionary is created with the *dict* function call, e.g.:

```
define pitt
  dict
    .name  = 'Willam Pitt'
    .title = 'Prime Minister'
    .date-of-birth   = '28 May 1759'
```

Here we have created a *dict* with three properties. The *dict* function takes a variable number of property definitions in Pairs, more formally:

```
  dict
     <dynamic symbol1> = <initial value1>
     <dynamic symbol2> = <initial value2>
     etc...
```

If there are no initial values given , the symbol *nil* is used as the value as in this example:

```
> dict
   .foo
   .bar

dict
      .bar = nil
      .foo = nil # Dictionary
```

Having properties is all very well, but we need a way to access them. As we have seen all objects are callable - including dictionaries. So to access the above *dict* object we call it and use the dynamic variables as follows:

```
> pitt .name

~ 'Willam Pitt' # String
```

Here the *.name* dynamic variable is bound to the *.name* property in the dict. To set the property value we use the *setq* function:

```
pitt
   setq .name 'William Pitt The Younger'
```

New properties can be created with *define* since the object acts as an environment in its own right. e.g.:

```
pitt
   define .father 'William Pitt the Elder'
```

Dictionaries also have a "magic" variable *.vars* which lists all the variables defined in the dictionary. This is handy for debugging. For example:

```
> pitt .vars

.date-of-birth .father .name .title .vars # Pair
```

## 9.1 Adding Behaviour to Dictionaries

Since functions in Genyris are bound to variables, and dictionaries have variables, behaviour can be added to dictionaries. It suffices to define a function with a dynamic name in the scope of a dictionary:

```
define jeb
   dict
      .firstName= 'Joe'
      .middleName= 'E.'
      .lastName= 'Brown'

jeb
   def .displayName()
       list .firstName .middleName .lastName
```

Once defined, the function is callable in the context of the *jeb* dict:

```
> jeb (.displayName)

'Joe' 'E.' 'Brown' # Pair
```

A dict can be used a mechanism for organisation. Consider the following fictional example - a set of functions organised in a dictionary called *file*:

```
## File Handling code
define file
  dict
    .name  = 'File Handling Functions'
```

```
    .version = '1.2'
file
   def .copy(from to) etc
   def .delete(filename) etc
   def .zip(file) etc

## Use of the file module
def archive(filename)
    (file.copy) filename '/tmp/foo'
    (file.zip) '/tmp/foo'
    (file.delete) filename
```

Here we define three functions bound to a single dict object. The functions can only be called by referencing the *file* object.

# 10 Synactic Sugar - Using the exclamation mark[4]

The exclamation mark, ! (aka pling) provides a shorthand way of accessing the properties of an object. This is implemented in the parser like quotes. Here are some examples

| Input Quote Sequence | Translated Expression |
|---|---|
| a!left | (a .left) |
| a!b!c | ((a .b) .c) |
| (f).a!b | (f)(.a .b) |
| (dict)!self | ((dict) .self) |
| a!b(var .x 23) | (a .b)(var .x 23) |
| .x!y | (.x .y) |

Tra

The previous example can be written succinctly:

```
def archive(filename)
    file!copy filename '/tmp/foo'
    file!zip '/tmp/foo'
    file!delete filename
```

# 11 Modules

The previous example shows how names can be grouped within a Dictionary object. This idea is made easier in the langugae through modules. A module is simply a dictionary into which the contents of a source file have been bound. Continuing with our example, first we provide a file of functions and definitions called 'file.g".

```
#
```

[4]The choice of this syntax is a tribute to Martin Richards.

```
# File Handling Module
#
define .doc 'File Handling Functions'
define .version '1.2'
def .copy(from to)
   etc
def .delete(filename)
   etc
def .zip(file)
   etc
def helperFunction(args)
   etc
```

Notice we don't need to declare a dictionary since this is provided by *import*. To use the module we first need to tell the interpreter where to find the file containing the module's code. This is done by adding to the *sys:path* global variable which is a list of directories to search:

```
sys:path = (cons 'examples' sys:path)
```

Now we read the source code:

```
import file
```

This creates a variable *file* in the environment of the caller (not necessarily the global environment). Using the module's dictionary we can now access the functions in the module:

```
## Use of the file module
import file
file!copy filename '/tmp/foo'
```

Note that *helperFunction* is defined with a normal lexical variable. It cannot be reached from outside the module since only dynamic symbols can be dereference via adictionary. Use this to create private variables and functions inside modules.

If we change the source code of *file.g* and need to load the new code, we must use the *reload* macro, since import checks to see if the module already is in memory, and uses the already loaded module.

```
> reload file
etc...
```

# 12 Namespaces

A way to prevent name clashes is to use symbol prefixes to define 'namespaces' for symbols. A namespace can be defined with a Parser Directive "*@prefix*" as follows

```
@prefix magic 'http://my.org/2008/spells/'
```

Here the expression starting with *@prefix* is consumed by the parser, and any subsequent symbols it sees starting with the prefix *magic.* are replaced with *http://my.org/2008/spells/*. Hence the true name of *magic.accio* is *http://my.org/2008/spells/accio*. We can print the full name of the symbol by quoting it:

```
> @prefix magic 'http://my.org/2008/spells/'
^magic:accio
~ http://my.org/2008/spells/accio # URISymbol
```

Here is the previous example re-worked using a namespace:

```
## File Handling Module (using prefixes)
@prefix file 'http://my/files/'
def file:copy(from to) etc
def file:delete(filename) etc
def file:zip(file) etc

## Use of the file module
@prefix f 'http://my/files/'
def archive(filename)
    f:copy filename '/tmp/foo'
    f:zip '/tmp/foo'
    f:delete filename
```

Prefixes apply to all symbols seen by the parser in the current parse, and can be used in a variety of ways (for example) to define global properties or interfaces.

## 12.1 Using Namespaces for Semantic Markup

Namespaces are compatible with RDF and allow you to embed the semantics of the information within the program. This can be used either for human consumption (to nail the meaning of the information) or for subsequent processing and linkage with external programs which also understand RDF. Here's an example:

```
> @prefix us "http://places.org/usa#"
var boston                  # variable declaration
  dict                      # new dictionary object
    .us:location = "Boston"
    .us:zip = "02110"
    .us:location-lat = 42.37
    .us:location-long = 71.03
dict
      .|http://places.org/usa#location| = "Boston"
      .|http://places.org/usa#location-lat| = 42.37
      .|http://places.org/usa#location-long| = 71.03
      .|http://places.org/usa#zip| = "02110" # Dictionary
```

# 13 Using Classes to Organise Behaviour

All Genyris objects, be they atomic (like numbers) or composite (like dictionaries) can belong to one or more classes. As such Genyris is a fully "Object-Oriented" language. The interpreter looks at the classes for functions to execute if the function name is dynamic (starts with a .). This way you can add behaviour to many objects in a single place. Classes are dictionaries with special variables which hold the relationships between classes. The standard classes all have the following variables:

**.classes** The list of classes to which the object belongs.

**.vars** List of property names.

**.classname** The String name of the class.

**.superclasses** The list of classes from which the class inherits.

**.subclasses** The list of classes which inherit from this class.

Genyris has a number of built-in classes beginning with *Thing*, root of the class hierarchy. Here is the builtin class hierarchy:

```
Thing
   Task
   Module
   SetList
   Alist
   Builtin
      PairEqual
      Pair
         ListOfLines
         PRINTWITHEQ
      Closure
         LazyProcedure
         EagerProcedure
      Symbol
         DynamicSymbolRef
         URISymbol
         SimpleSymbol
            |http://www.genyris.org/lang/syntax#Keyword|
      Triplestore
      Triple
      Sound
      StringFormatStream
      ParenParser
      Parser
      System
```

```
Writer
Reader
CSV
File
String
   ShortDateTimeString
Bignum
Dictionary
   Object
   StandardClass
```

To add behaviour to a class, we need to add a dynamic variable bound to a closure object - in other words we need to define a method. For example a method to compute the square of a number is added to the *Bignum* class:

```
Bignum
   def .square() (* .self .self)
```

Notice that the method uses the *.self* variable which will be automatically bound to an object. Now all Bignums can compute their own square e.g.:

```
> 4234389 (.square)

~ 17930050203321 # Bignum
```

We need to call methods in the correct way to ensure they refer to the right object since they are dynamic, not lexical variables. So if we tried to say:

```
> (4234389.square)
```

We would get an error. There is a big difference between *4234389 (.square)* and *(4234389.square)*. In the first case we are creating an environment around the Bignum *4234389*, then we execute the function bound to the dynamic variable *.square* from Bignum. In the second however, we are getting the Bignum's *.square* function but then applying it in the context of the caller. This is most likely not what was intended. In general, methods should almost always be called in the first way as in:

```
<object> (<method> <arg1> <arg2> ... <argn>)
```

Or if there are multiple method calls to be made:

```
<object>
     <method> <arg1> <arg2> ... <argn>
     <method> <arg1> <arg2> ... <argn>
     etc...
```

## 13.1 Defining Your Own Classes

Classes are relatively complex objects so the language provides a built-in macro for creating new classes and binding them. The syntax is straight forward - let's define a class for length units:

```
> class Inches()

<class Inches (Thing)> # StandardClass Dictionary
```

This simply creates a class which is a subclass of *Thing*. By convention class names begin with an upper-case character. We can use this class to annotate existing objects. For example:

```
> tag Inches 12

~ 12 # Inches Bignum
```

The *tag* function adds a class to an object's list of classes and returns it. Notice the interpreter prints out the list of classes 12 belongs to, now including *Inches*.

So far so good, now let's add a method to convert to meters. Lets assume an *Inches* object is a kind of *Bignum*, and add a method to it:

```
class Inches(Bignum)
   def .toMeters()
      * .self 0.0254
```

The second parameter to *class* a list of superclasses, in this example, just *Bignum*. We can now define a foot and convert it as follows:

```
define a-foot
   tag Inches 12

> a-foot (.toMeters)

~ 0.3048 # Bignum
```

This is fine, but we are still returning a *Bignum*. Let's refactor to add a *Meters* class and tag the return appropriately:

```
class Length()
   def .toMeters()
      raise 'Oops - you invoked an abstract class!'

class Inches(Length)
   def .toMeters()
      tag Meters (* .self 0.0254)

class Meters(Length)
   def .toMeters() .self
```

Here we have defined an abstract base class and two derived classes which both have the *.toMeters* method. The *raise* function catches invalid use of the *Length* class. Lets try the conversion again:

```
> (tag Inches 12) (.toMeters)

~ 0.3048 # Meters Bignum
```

Here we are using a sub-expression which returns *12 Inches* and this object is the focus of the call to *.toMeters*. Note the result is now in *Meters*. With this new class structure in place we can now add a method to add two lengths in any units:

```
Length
   def .add(other)
      tag Meters
         + (.toMeters)
            other (.toMeters)
```

This method converts both the current object and the argument to *Meters,* performs the addition and returns the result in *Meters*. Here's how it runs:

```
define a-meter
   tag Meters 1
define a-foot (tag Inches 12)
> a-foot (.add a-meter)

~ 1.3048 # Meters Bignum
```

This is useful however in the above example a user of our classes could get errors by tagging objects which cannot be added together. For example a string will fail:

```
(tag Inches '12') (.add (tag Meters 1))          # Error
```

To provide protection and we use type checking features and class validators.

## 13.2 Type-Checked Function Arguments

Genyris supports type annotations found in most statically typed languages. These type checks are purely optional. When a function is defined, the arguments and return value may be annotated with a class. The actual arguments are checked with a validator (if present). Here's an example of a sensitive function protected by type checks:

```
def safe-call ((a = Bignum) (b = Bignum) = Bignum)
    fragile-function a b
```

This function only allows Bignums to be passed in or returned. The last element of the arguments list specifies the return type. If a type check fails an exception is raised.

## 13.3 In-Line Type Checks

Genyris provides another mechanism for checking type constraints. The *is?* function will check if the result of the expression is *.valid?*, or if it is an instance of the class. For example this function will always raise an error because *3* is not a subclass of *String*:

```
def fails()
    define x 3
    is? x String
```

If a class validator is provided, this will be used otherwise a simple "nominative" type check of class membership based on the object's tagged classes is used. The *.valid?* method is a stronger check, however nominative checking is sometimes preferred[5].

## 13.4 Inheritance and Class Properties

The inheritance mechanism in Genyris works by searching on object's classes list for classes which have the dynamic symbol required, it also recursively searches the superclasses of all the classes it finds in the classes list. The classes are ordered with those classes deepest in the hierarchy first. [6]

This inheritance of properties is the same regardless of the type of property (method or data). Hence an object can access information stored in it's classes and superclasses. Here's an example where a base class supplies a boolean value to an object:

```
class Orange()
   define .pips ^true

> (tag Orange 'my lunch').pips

~ true # Symbol
```

## 13.5 Class Validators

To help define membership of a class, the class can provide a *.valid?* predicate method. This can assess an object and return *true* if it is a valid member of the class. The *tag* function calls *.valid?* if provided and fails when *.valid?* returns *nil*. We can add a validator to our example base class:

```
Length
   def .valid?(obj)
      is-instance? obj Bignum
```

The *is-instance?* function only returns true if the object is an instance of *Bignum* or its subclasses. This prevents anything except numbers being tagged.

[5] Genyris supports both "nominative" and "structural" subtyping

[6] This order is listed when the classes are printed by the command-line interpreter.

## 13.6 Traditional Constructors and Factories

While the philosophy of Genyris is to classify objects after construction, it does not inhibit using traditional constructors in classes. 'Factory' functions for object construction are preferred even in traditional languages. Factory functions are simply functions which construct the appropriate kind of object based on the inputs given. A factory/constructor can be as simple as this:

```
class Person ()
   def .new (name date-of-birth)
      dict
         .name = name
         .dob = date-of-birth
         .classes = (list Person)
Person!new 'Jo' 23
```

A more general approach is to provide a "*new*" function which calls a class-specific "*init*" function. For example here is a class with an *.init* member which creates properties in a dict created by *.new*:

```
class PersonTraditional (Object)
   def .init((name = String) (age = Bignum))
      define .name name
      define .age age
```

*Objects* are created by calling *.new*:

```
> PersonTraditional (.new 'Abe' 99)
dict
      .age = 99
      .name = 'Abe' # PersonTraditional Dictionary
```

Here is a simple implementation of *.new* in a base class:

```
class Object (Dictionary)
  def .new(&rest args)
      (tag .self (dict))
         apply .init args
         .self

   # NOOP .init method
   def .init(&rest args)
      .self
```

This *.new* collects all the input arguments via the *&rest* keyword, it creates an empty *dict*, tags it with the derived class and passes the collected arguments to the class's *.init* function. The default *.init* function returns the new object un-modified.

## 13.7 Automating Classification

This language embodies the opinion that objects are created first, then they are classified - rather than the classification being determined during object construction. Let's explore how the *.valid?* predicates can be used to automate classification.

Validator functions can be developed to any complexity required. For example validators can inspect the values of properties and objects rather than just their type. Here's an example which is *true* for even numbers:

```
class EvenNumber()
  def .valid?(x)
      equal? (% x 2) 0
```

Validators provide a way to automatically categorize unknown objects - an important tool for input validation.

The Genyris distribution includes file "examples/classify.g" which shows this pattern. It defines a simple *classify* function which recursively walks the class hierarchy testing an object's compliance with validators. There is an example of classification of people into classes based on age and possessions. We load the source files [7]:

```
include 'examples/people.g'
```

This creates an un-classified object, assigns it to a variable *kevin*, and calls *classify*:

```
define kevin
  dict
     .name= 'Kevin'
     .age= 49

classify Person kevin
```

We now display the object

```
> kevin

dict
      .age = 49
      .name = 'Kevin' # Boomer Dictionary
```

The result shows the classifier has recognised *kevin* as a *Boomer*. Here are the classes that make this happen:

[7] The *load* function reads an executes the source file from the Java classpath. Genyris initialization code source files are stored within the Java "jar" file.

```
class Person ()
   def .valid? (obj)
      obj
         bound? .age
class Boomer (Person)
   def .valid? (obj)
      obj
         between 45 .age 60
```

To be a valid *Person* kevin must have an the *.age* property, and to be a *Boomer* it's value must be between 45 and 60. The *classify* function only calls the derived class's validator if object is in the base class. It tagged *kevin* with the *Boomer* class.

This technique can be used to categorise a program's inputs or validate output data, and even re-validate previously classified objects.

## 13.8 On Ducks and Interfaces

'Duck' typing in a language is jargon for 'structural' subtyping - *If it looks like a duck and quacks like a duck - then it is a duck.* Duck typing relies on programmers to ensure that objects passed around actually do have the properties and methods expected by the downstream code. If there is a mismatch then eventually an error will result. For example if we could define a *.copy* function which expects some kind of stream object with *.next* and *.last?* methods. There is no need to perform type-checking in the interface since if the methods exits all will be well. Duck typing is perfectly adequate for most programming tasks, however many developers like to formalize the interfaces.

In Genyris an Interface could be defined by either providing an appropriate validator or by simply tagging objects with their supported interface classes. For example here is a class validator for the above scenario:

```
class Stream-Interface()
   def .valid?(object)
      object
         and
            bound? .next
            is-instance? .next Closure
            bound? .last
            is-instance? .last Closure
```

The validator here checks whether the object has *.next* and *.last* properties, and whether they are procedure objects.

# 14 Symbols and Semantic Triples

RDF[8] concepts such as triples and triplestores are available in Genyris as part of the language. Although these were made popular by the Semantic Web these Description Logic ideas can be used in general programming. The Genyris implementation removes some of the constraints of RDF and allows these ideas to applied at the program object level. The goal is for all developers to be beneficiaries of these elegent data structures and algorithms, not just those concerned with Semantic Web problems.

Triples and triplestores which are derived from Description Logic also form the foundation for the Genyris *type system*. The type system will allow data, classes and functions to be 'marked-up' with meta-data describing their allowed properties and behaviours.

Description Logic techniques allow great flexibilty and power in the addition of meta-data and inference. We hope in time Genyris developers will have access to higher-order tools which are available for RDF and OWL within the context of their Genyris application programming.

## 14.1 Triples

Genyris includes triples as a first-class system object. Triples are described in the W3C RDF Primer document, amongst others. A triple consists of three atoms, a *Subject* , *Predicate* and *Object*. Each triple is a simple descriptive statement about the relationships between objects or their properties. For example the triple:

```
UnixBox1 hasUser root
```

Says that the machine UnixBox1 has a user account 'root'.

In Genyris the *Subject* of a triple may be any language object, including atoms and 'literals' in RDF terms. This allows statements to be made about any object in the system. Another way of viewing this is that any object in the system may have arbitrary properties, stored in triples.

The *Predicate* of a triple however, must be a symbol. This ensures that the statement is at least, readable.

Any language item may be the *Object* of a triple. This allows relationships between any two language items to be recorded.

Triples alone are not particularly useful without a way of comparing them and managing them. *Triplestores* (as the name suggests) are in-memory sets of triples with functions for searching for, adding and removing triples. Because Genyris triples may have any object as their subjects, Triplestores are used where other languages may rely on hash-tables. Triplestores in their turn can be used as sets, with comparison, intersection and union functions.

[8] Resouce Description Framework

### 14.1.1 Creating triples with *triple*

Triple objects are created by the *triple* function which takes three arguments, the *subject, predicate* and *object.* The *predicate* must be a Symbol. *Examples*:

```
> triple ^Lewis ^hasMother ^Mae
(triple Lewis hasMother Mae) # Triple
> triple 23 ^hasUnits ^Inches
(triple 23 hasUnits Inches) # Triple
```

### 14.1.2 Accessing parts of a triple with *.subject*, *.predicate* and *.object*

The components of the triples can be accessed with the above dynamic symbols. For example:

```
> var l
   triple ^Lewis ^hasMother ^Mae
(triple Lewis hasMother Mae) # Triple
> list l!object l!subject l!predicate
(Mae Lewis hasMother) # Pair
```

### 14.1.3 Triple Equality

Two triples are considered *equal?* if the subjects and predicates are *eq?* and the objects are *equal?.* The *equal?* test for the *object* means that strings, integers and lists which look the same will be treated as equal. Refer also to the definitions of *eq?* and *equal?.* This style is designed to allow simple literals to be used as objects such as *12.3* and *"Joe"*. If networks of triples are required in an application, then the objects are also subjects. In this case Dictionaries or Symbols are recommended since their equality is specific.

For example:

```
> equal? (triple ^S ^P 12.3) (triple ^S ^P 12.3)
true # SimpleSymbol
> equal? (triple 123 ^p "yes") (triple 123 ^p "yes")
nil # SimpleSymbol
```

## 14.2 TripleStores

Genyris provides a special container class for triples, the 'triplestore'. Triplestores are really just a set of triples, since all triples within the store are unequal. Triplestores are not persisted (saved on disk) they are purely in-memory objects. Triplestores are used as lookup tables (instead of a hashtable) and they are used for more complex comparisons between sets of triples. The speed of the current implementation is inversely proportional to the number of triples in the store.

### 14.2.1 Creating TripleStores with *tripleStore*

The *triplestore* function constructs a triplestore object. It accepts an option list of 3-lists which are converted to triples. Example:

```
> triplestore
   ^(joe age three)
   ^(joe age ten)
(triplestore) # Triplestore
```

### 14.2.2 TripleStore equality

Two triplestores are *equal?* if their constituent triples have an equal triple in the other triplestore and vice versa.

### 14.2.3 Adding Triples with *.add*

The *.add* method adds a triple to a triplestore. If the triple is already present no error is thrown. The triplestore object is returned. Syntax is:

*.add <triple>*

Example:

```
ts(.add (triple ^s ^p ^o))
```

### 14.2.4 Removing Triples with *.remove*

The *.remove* method removes a single triple from a triplestore. If the triple is already absent no error is thrown. The triplestore object is returned. Syntax is:

*.remove <triple>*

Example:

```
ts(.remove (triple ^s ^p ^o))
```

### 14.2.5 Querying Triples with *.select*

The *.select* method provides a way of iterating over the set of triples in the triplestore and returning a new triplestore with triples of interest. Syntax:

```
.select <subject> <predicate> <object> [<procedure>]
```

The function iterates through every triple in the triplestore, if the *<subject>*, *<predicate>* and *<object>* parameters match those of the triple it is added to the result set. Nil parameters are ignored[9]. Matching is performed as for triple equality rules. Thus *(.select nil nil nil)* matches all triples. Matching rules:

| Parameter | Predicate |
|---|---|
| subject | eq? |
| predicate | eq? |
| property | equal? |

[9] This is likely to change since *nil* is not matchable.

If the last parameter is present it must be a procedure which takes three parameters of the form:

```
function (<subject> <predicate> <object>)
```

If the triple is matched, then this function is called with the triple's subject, predicate and object as actual parameters. If the function returns a non-*nil* value then the triple is added to the result set. The matching logic is equivalent to:

```
and
   eq? <subject> .subject
   eq? <predicate> .predicate
   equal? <object> .object
   <procedure> .subject .predicate .object
```

Here is a simple example: of select in use:

```
var ts
   triplestore
      ~ ^(joe age three)
      ~ ^(joe age ten)
      ~ ^("John" age 22)
      ~ ^("John" height 223)
var result
   ts
     .select ^joe nil nil
assert
   equal?
      result
      triplestore ^(joe age three) ^(joe age ten)
```

### 14.2.6 Retrieving unique values with *.get subject predicate*

Often triplestores will contain unique triples, ie there is only one triple with the same *subject* and predicate. The *.get* method retrieves the *object* of this triple. If there is more than one matching triple, or none, an error is raised.

Syntax:

```
<triplestore> (.get subject predicate)
```

### 14.2.7 Retrieving multiple values with *.get-list subject predicate*

The *.get-list* method retrieves a list containing all the *objects* of the triples which have the *subject* and *predicate* specified. If there is no matching triple *NIL* is returned. Syntax:

```
<triplestore> (.get-list subject predicate)
```

### 14.2.8 Updating with *.put subject predicate object*

The *put* method removes all triples from the triplestore with the matching *subject* and *predicate*, then adds a single triple with the *subject predicate object* specified. If there are no matching triples no error is raised, the triple is simply added. This operation is intended to support the pattern of a unique triples, where it serves as an update operation.

### 14.2.9 Fetching all the triples with *.asTriples*

The *.asTriples* method returns a list containing all the triples in the triplestore.

### 14.2.10 Triplification with *.asTripleStore*[10]

All objects in Genyris can be converted into a series of triples depending on their type. Typically this results in a type triple for each of the object's classes, and properties as individual triples. Examples:

```
> class Inches
<class Inches (Thing)> # StandardClass
> var foot (tag Inches 12)
12 # Inches Bignum
> (foot(.asTripleStore))(.asTriples))
((triple 12 type <class Inches (Thing)>)
 (triple 12 type <class Bignum (Builtin)>)) # Pair
```

and

```
> (^(1 2)(.asTripleStore))(.asTriples)
((triple (1 2) type <class Pair (Builtin)>)) # Pair
```

### 14.2.11 Adding Meta-Data to Triples

Triples can belong to multiple classes just like any other Genyris object. This allows them to be annotated and have classes and methods. For example here we tag a triple as being factual:

```
> class Fact
<class Fact (Thing)> # StandardClass
> var kr (triple ^KevinRudd ^role ^PrimeMinister)
(triple KevinRudd role PrimeMinister) # Triple
> tag Fact kr
(triple KevinRudd role PrimeMinister) # Fact Triple
```

Triples can also be the subject of other triples which means complex meta-descriptions are possible without resorting to reification. Here is the example above expressed with another triple.

[10] This method is not fully implemented for all types.

```
> var meta-kr (triple kr ^Belief ^Fact)
(triple (triple KevinRudd role PrimeMinister) Belief Fact) # Triple
```

Triples and triplestores allow the programmer to store information about objects which may not have a means of adding properties of their own beyond their class membership list.

# Part II
# Reference

## 15 Compiling From Source

To compile from source you need install *Ant* (version 1.7.1 or later). Obtain a copy of the source release file *genyris-source-NNN-DDDDDDD.zip* , unzip the file to a working directory. Change directory to the root of the unpacked directory. Then run the following command to build the program:

```
$ ant
```

At the end of the build you should see a summary of tests passed:

```
OK (212 tests)
```

### 15.1 Using a Mercurial Repository

TODO

## 16 Running Genyris

### 16.1 Simplified Launching via the Path

The distribution includes shell and batch files for simplified launching, both which require a change to the user's environment.

#### 16.1.1 Unix

Add an environment variable called *GENYRIS_HOME* which contains the path of the unpacked release. Modify your *PATH* environment variable to include *$GENYRIS_HOME/bin*. The interpreter can then be launched from the shell with

```
$ genyris
```

### 16.1.2 Windows

On Windows the arrangement is similar. Add an environment variable called *GENYRIS_HOME*. Modify the *PATH* environment variable to include *%GENYRIS_HOME%\bin:* (from the GUI Start > Control Panel > System > Advanced > Environment Variables > New). Now the interpreter can then be launched from the command prompt with *genyris*.

## 16.2 Command line options

Genyris is started by the Java interpreter from the command line. Java options are used to specify which main class. to use. There are multiple *main* classes in the genyris-bin jar so the main class can be specified explicitly with *-cp* syntax.

### 16.2.1 Using Genyris standard IO command-line interpreter.

The java *-jar* option specifies the jar to execute. For example:

```
$ java -jar genyris-bin-nnn-xxxxxxxxx.jar
```

Since the default main class is the REPL commandline it is automatically chosen by the -jar option. The same REPL class above can be run identically as as follows:

```
$ java -cp genyris-bin.jar org.genyris.interp.ClassicReadEvalprintLoop
```

In both cases all subsequent shell command line arguments are ignored. The session is terminated when the interpreter read and End-Of-File from the standard input (^D on UNIX or ^Z on Windows).

**The top level** Genyris commands can be typed at the prompt, using two carriage returns (↩) to terminate a statement. The CLI reads the next indented expression, evaluates it in the global environment and prints the result. It also appends a comment consisting of the names of the classes of the result. This allows the user to verify the correct classes have been returned.

### 16.2.2 Running a script in the command line

```
DOS> genyris -eval print 'hello world'
unix$ genyris -eval print 'hello world'
```

If the return value of the expression is printed, the process exits with a 0 status, otherwise 1.

### 16.2.3 Specify a file to run on the command line

```
$ genyris examples/queens.g 5
```

Arguments are passed after the file name.

### 16.2.4 Making a script executable (Unix)

If a script begins with *#!* it allows the Unix kernel to run the script directly. Here's an example script:

```
#!/usr/bin/sh /opt/home/birchb/workspace/genyris/bin/genyris
display 'Hello World'
```

if the script is made executable it can be run directly:

```
$ chmod u+x myscript.g
$ ./myscript.g
```

## 16.3 Interpreter Boot sequence

Regardless of the enclosing main class, the interpreter boot sequence is the same. After initialisation the interpreter searches for the resource "*org/genyris/load/boot/init.g*" in the Java classpath and executes it. This file is provided in the genyris-bin jar file and contains a Genyris bootstrap which in loads other source files also provided within the jar file. This is equivalent to executing :

```
load 'org/genyris/load/boot/init.g'
```

# 17 Loading Source Files

Functions are provided to allow files of source code to be read and executed. The parser to be used is automatically selected based on the fule suffix. The following suffixes and parsers are supported:

| suffix | Parser Class | Syntax |
|---|---|---|
| lin | Parser | Genyris indented syntax |
| lsp | ParenParser | Free format LISP Syntax with parentheses in place of indentation. |

## 17.1 load *<java resource name>*

The *load* function reads and executes a source file from the Java classpath. Some initialization code files are stored within the distributed binary Java "jar" file . The *<java resource name>* parameter is a full path to the file. For example:

```
load 'examples/people.g'
```

## 17.2 include *<path to filename>*

The *include* function reads and executes files of scripts in the global environment. The <path to filename> parameter refers to a normal operating system file path. Examples:

```
include 'examples/people.g'   # relative path
include 'c:/workspace/genyris/examples/queens.g'  # absolute path - Windows
```

The return value is a string with the actual file path loaded. If the file cannot be found an exception is raised.

## 17.3 sys:import <filename name>

The *sys:import* function is similar to the include function in that it reads and executes a file of Genyris source code. However the code is executed *in the environment of the caller*. Therefore in a lexical environment, new bindings created by *define* and *def* et. al. will be created locally. In a deynamic environment such as in a Dictionary, the dynamic bindings in the top level of the file will affect the caller's object. This function is used by the *import* facility to define modules.

## 17.4 import <module name>

The *import* macro uses *sys:import* to create dictionaries in which the file's bindings are held. To allow users to have multipe calls to *import*, yet all refer to the same object, *import* stores all module dictionaries in the *sys:modules* triplestore. This triplestore contains a binding between the module name and the module object. When the import is called it checks to see if the module has already been loaded, if so it binds the module to a variable of the same name in the caller's environment. Hence there may be many bindings of the module name to the same physical object.

To simplify loading, the *import* macro searches for a file with the name of the module plus the *'.g'* suffix. It uses the list of directories in the *sys:path* variable for the search path and uses the first matching file it finds. Example:

```
sys:path = (cons 'examples' sys:path)
import queens
queens!queens 5  # call the module's public function
```

## 17.5 reload <module name>

The *reload* macro uses the existing module's filename to reload the source code of the module. If there are multiple bindings to the module, only the caller will have a binding to the new module. The binding in the *sys:modules* triplestore is replace with the new object. Other references to the module will still refer to the old module.

# 18 Syntax

## 18.1 Indented Format

## 18.2 ~ for line continuation

## 18.3 Lisp Format

## 18.4 carat ^

## 18.5 Equals =

## 18.6 Parentheses ()

## 18.7 comments #

## 18.8 accessing dynamic variables with .

## 18.9 comma ,

## 18.10 comma-at ,@

## 18.11 backquote ‘

## 18.12 [] and {}

## 18.13 parser directives

## 18.14 @prefix

# 19 Evaluation

## 19.1 eval <expression>

## 19.2 apply <closure> <parameter list>

## 19.3 Type Checks with =

## 19.4 symbol-value <symbol>

Example:

```
> (dict (.a = 1)(.b = 2))
    (symbol-value ^.b)
2 # Bignum
```

More complex example:

```
> var D (dict (.a=1)(.b=2))

> for v in D
```

```
   u:format "%s = %s\n" v (D (symbol-value v))
.b = 2
.a = 1
.self = (dict (.a = 1) (.b = 2))
.vars = (.b .a .self .vars .classes)
.classes = (<class Dictionary (Builtin)>)
```

### 19.5 dynamic-symbol-value <symbol>

### 19.6 template <expression>

### 19.7 quote <expression>

### 19.8 the <expression>

## 20 Bindings

### 20.1 Definition of Variables

#### 20.1.1 var

#### 20.1.2 defvar

#### 20.1.3 define

#### 20.1.4 bound?

### 20.2 Assignment

#### 20.2.1 setq

#### 20.2.2 set

### 20.3 Definition of Functions

#### 20.3.1 def

#### 20.3.2 df

#### 20.3.3 defmacro

## 21 Predicates

Predicates are a general class of functions which either return *nil* to signify falsehood, or a non-*nil* value for truth.

## 21.1 Comparisons

### 21.1.1 eq? <a> <b>

The *eq?* function compares two expressions to see if they are the same physical object in the interpreter memory. In other words it returns *true if* the by-value references passed in as parameters are the same, otherwise *nil* is returned. Examples:

| Expression | *eq?* return |
|---|---|
| eq? ^a ^a | true |
| eq? 1 1 | nil |
| eq? 'c' 'c' | nil |

### 21.1.2 equal?

The *equal?* function compares the contents of two expressions and returns *true* if they have the same value, otherwise *nil.* If two expressions are *eq?* they are always *equal?.*

| Expression | *equal?* return |
|---|---|
| equal? ^a ^a | true |
| equal? 1 1 | true |
| equal? 'c' 'c' | true |
| equal? ^(1 2) ^(1 2) | true |
| equal? (dict (.a=2))(dict(.a=2)) | false |

### 21.1.3 >

The greater than function returns *true* if its first argument is numerically greater than its second argument.

### 21.1.4 <

The greater than function returns *true* if its first argument is numerically less than its second argument.

## 21.2 Logical Functions

Genyris has a set of boolean functions which expect arguments which are either *nil* to signify falsehood or non-nil for truth.

### 21.2.1 not <x>, null?

If the argument is *nil* these functions return *true* otherwise *nil* is returned.

### 21.2.2 and <a1> <a2> ...

This macro executes its arguments in turn, if an argument evaluates to a *nil* value execution stops and it returns *nil*. Otherwise execution continues to the last argument whose value is returned. Examples:

| Expression | Result | Comment |
|---|---|---|
| and 1 2 3 4 5 | 5 | |
| and 1 2 3 nil 5 | nil | |
| and (print 1) (print 2) | true | Both 1 and 2 are printed |
| and (not (print 1)) (print 2) | nil | 1 is printed |

### 21.2.3 or

This macro executes its arguments in turn, if an argument evaluates to a non-*nil* value execution stops and it returns *true*. Examples

| Expression | Result | Comment |
|---|---|---|
| or 1 2 3 4 5 | 1 | |
| or nil 2 3 nil 5 | 2 | |
| or (print 1) (print 2) | true | 1 is printed |
| or (not (print 1)) (print 2) | true | 1 and 2 are printed |

# 22 Closures

Closures are executable objects which retain the environment in which they were created for use during execution at a later time.

## 22.1 *lambda* or *function*

```
[function|lambda] ([arg1]... [&rest restargs] [= return-validator]) [body]
```

The *function* macro creates a lexical closure at the point of definition and returns an anonymous 'eager' procedure object. *lambda* is an alias for *function* and is kept for historical reasons. The procedure object can be invoked as a function. When the procedure is called it's real arguments must match the formal arguments specified in the function definition. The formal arguments may have one of the following forms:

```
() - no arguments
(arg1 arg2 .. argn) - alist of required arguments
(... &rest arglist) - variable arguments
```

If the argument list includes &rest, all the remaining real arguments are collected into a list and passed to the function in a single argument, the name of which is specified after the &rest.

The function body - a list of expressions - is executed each expression in turn, the last evaluation is returned by the function. Formal arguments may

be specified in two forms, either as a symbol or as a type specification. If a symbol is used, the real arguments are bound to the symbol in the execution environment of the function when it is called. The type of the formal argument may be supplied as the right hand side of a cons pair:

```
(arg-name = validator)
```

The validator must be a symbol bound to a dictionary or class which has a validator function (*.valid?*). When the procedure is called, the interpreter executes the .valid? function with the real argument as its single parameter. If the validator returns false, a type miss-match error is raised.

The expected type of the return value can be specified after a trailing = in the argument list.

**Examples:**

```
function ()
```

returns a procedure which returns nil

```
function (x y) (cons x y)
```

returns a procedure with two mandatory arguments

```
function (x &rest y) (list x y)
```

returns a procedure with one mandatory and one rest argument

```
function ((i = Bignum) (j = Bignum) = Bignum) (+ i j)
```

returns a procedure with two mandatory arguments with expected types and a return value with expected type.

## 22.2 *lambadq*

This macro is identical in syntax to *function* but it creates an anonymous lazy procedure (which does not execute it's arguments when called).

## 22.3 *lambdam*

This macro is identical in syntax to *function* but it creates a lazy procedure which re-executes it's return value in the environment of the caller - in other an anonymous macro.

# 23 Classification

## 23.1 *class*

## 23.2 *tag*

## 23.3 *remove-tag* <class> <object>

## 23.4 *is-instance?*

# 24 Constants

The interpreter pre-loads a number of bindings in the global environment which are treated by users as constants.

## 24.1 *nil*

This symbol is used to denote an empty list, the end of a list. It is used by the conditiona macro *cond* amongst others to denote falsehood. *nil* is bound to itself in the global environment. The empty list *()* is a synonym for nil.

## 24.2 *EOF*

Used to denote the End Of File, this symbol causes a parser to exit if seen.

## 24.3 true

A convenient symbol used to explicitly signal the opposite of *nil*, *true* is bound to itself in the global environment.

# 25 Tasks - Multiple Threads of Execution

*Tasks* in Genyris are modelled on Unix processes. They are seperate interpreter instances where shared memory is minimal, and communication is conducted via system objects outside of the interpreters themselves. This approach relieves the programmer with the burden of debugging multi-threaded code yet retains the advantages of asynchronous execution and multi-core hardware. This implementation uses a single Java thread per interpreter and uses Java in-process objects for inter-task communication.

## 25.1 Running Multiple Interpreters in the Same Process with *spawn*

```
spawn <source file to include> <args ...>
```

Spawn creates a new Genyris interpreter instance which is then executed by a seperate Java thread. Only the objects passed as parameters are shared between interpreter threads. They all share the same operating system interfaces, files, sockets etc since they are running in the same process. The function returns a *Task* object, which is a Dictionary containing, the name, id and state of the underlying Java thread. For example:

```
spawn 'testscripts/spawn-example.g' 1 2 3 4 5
```

returns:

```
(dict (.id = 9) (.name = 'examples/spawn-example.g') (.state = 'RUNNABLE')) # Task Dict
```

The *spawn-example.g* file contains:

```
@prefix sys 'http://www.genyris.org/lang/system#'
while true
   print sys:argv
   sleep 5000
```

The output is:

```
'examples/spawn-example.g' 1 2 3 4 5
'examples/spawn-example.g' 1 2 3 4 5
...
```

## 25.2 Termination with *kill* :

Tasks and the underlying Java threads are killed with the *task:kill* function. This is also available as the *.kill* method in the *Task* class. A string is returned after completion. Note there is no guarantee from Java that the thread will be terminated.

```
<Thread>(.kill)
```

or

```
@prefix task 'http://www.genyris.org/lang/task#'
task:kill <integer Java thread id>
```

## 25.3 Thread Safety

The *spawn* function provides a way of multi-tasking sections of an application. This allows seperate interpreter instances to execute, and share some common value objects passed by main arguments. The interpreter does not cater for multiple threads of execution running within the same interpreter scope. Although it is technically possible to any genyris objects including functions between tasks,

this is not supported. This is because execution of stored closures will mean two interpreters are operating in the same environment.

Genyris *Tasks* are not *Threads* the intent is that they are self-contained, supporting single-threaded applications only. However since the underlying Java objects must be shared between tasks for inter-task communication, a means of making this thread-safe is provided.

## 25.4 *task:synchronized* - Controlling access to shared objects

The *spawn* and *httpd* functions allow Genyris objects to be passed from one thread of execution to another. This is provided so that system objects such as pipes, files etc can be shared.

Both tasks have references to the same in-memory objects, and there is real danger that the actions of one thread will corrupt the objects used by another. The *synchronized* macro ensures that developers can serialise access to shared objects. The syntax is as follows:

```
task:synchronized <object>
    <code body>
    ....
```

The effect of the macro is to wait for an implicit lock associated with the *<object>* to become free. When the lock is free the thread sets the lock and executes the *code body*. When the execution completes the lock is freed ready for another thread. A test-case:

A task loops, watching a shared *Pair*, checking to see if the *left* value is ever 1. If so it raises an error.

```
task:synchronized shared
     read-value = (shared.left)
cond
     (equal? 1 read-value)
            raise "error"
```

A second task loops, changing the value from 0 to 1 and back.

```
task:synchronized shared
        shared (.left = 1)
        sleep 13
        shared (.left = 0)
        sleep 23
```

Because both tasks are executing this code in the scope of a *synchronized* call, with the same shared lock object, the first task never sees the value altered to 1 by the second task. Without the *synchronized* macro the first task sees the manipulations of the second.

### 25.5 *ps* - list current running threads

The *ps* function returns a list of *Task* objects for each of the current tasks.

# Part III
# Libraries

Genyris is, like Scheme, a small language which allows users to develop their own paradigms and languages. Since Genyris has access to Java classes via *java:import* a programmer can readily create his or her data types, I/O libraries and so on. Hence the libraries provided with Genyris documented here are not a mandatory part of a Genyris implementation and their interfaces may change over time without changing the core language. The main emphasis is to provide common data types and functions useful for every-day scripting in a manner that fits well with the unique Genyris classification paradigm and its not-so-unique syntax.

## 26 Maths Functions

Genyris includes basic arithmetic operations on Bignum obects.

### 26.1 +

This function computes the sum of its arguments, which may be many.

### 26.2 -

This function computes the accumulated difference of its arguments, which may be many. Example:

```
(- 1 2 3) = -4
```

### 26.3 *

This function computes the product of its arguments, which must number two or more.

### 26.4 /

This function computes the quotient of its arguments, which must number at two or more.

### 26.5 %

This function computes the remainder of its arguments, which must number at two or more.

### 26.6 power

This function calculates the first argument raised to the power of the second. Example: *(power 2 8) => 256*

## 27 Time and Dates

### 27.1 *System!ticks* - Current system time in milliseconds since the epoch

```
> System!ticks
1266995579117 # Bignum
> format-date 0 "dd MMM yyyy HH:mm:ss Z" 'GMT'
'01 Jan 1970 00:00:00 +0000' # String
```

### 27.2 *now* - Current system time in milliseconds since the epoch

An alias for *System!ticks*.

### 27.3 *format-date* <epoch milliseconds> <format String> <timezone>

For example:

```
> format-date (System!ticks) "dd MMM yyyy HH:mm:ss z" "Australia/Melbourne"
'24 Feb 2010 18:11:41 EST' # String
```

### 27.4 *calendar* <epoch milliseconds> <timezone>

Returns a Calendar (dictionary) with the following properties:

**.am-pm** = symbol indicating AM or PM

**.day-of-month** = Integer day of the month starting with 1.

**.day-of-week** = Integer day of the week starting with Sunday which is 1.

**.day-of-week-in-month** = Ordinal number of the day of the week within the current month.

**.day-of-year** = The day number within the current year starting with 1.

**.dst-offset** = Daylight savings offset in milliseconds

**.era** = symbol AD or BC

**.hour** = Hour of the day (12 hour clock)

**.hour-of-day** = Hour of the day (24 hour clock)

**.millisecond** = e.g. 93

**.minute** = e.g. 2

**.month** = Integer month starting with January, which is 0

**.second** = e.g. 42

**.week-of-month** = Week in the month starting from 1.

**.week-of-year** = Week number in the year starting from 1.

**.year** = e.g. 2010

**.zone-offset** = Timezone offset in milliseconds from GMT

For example:

```
> print (calendar (System!ticks))
dict (.am-pm = pm)
.day-of-month = 10
.day-of-week = 4
.day-of-week-in-month = 2
.day-of-year = 69
.dst-offset = 1
.era = AD
.hour = 10
.hour-of-day = 22
.millisecond = 93
.minute = 2
.month = 2
.second = 42
.week-of-month = 2
.week-of-year = 11
.year = 2010
.zone-offset = 10
```

# 28 Input and Output

### 28.0.1 Global Output Functions

***print*<arg1>...<argn>** Outputs its arguments to the current standard output stream as if typed in by a user in indented format. - strings are quoted, escape characters are output. Arguments on the output are seperated by a newline. Example:

```
print ^(1 'w\n' (x y))

1 'w\n'
   x y
```

***write*<arg1>...<argn>** Outputs its arguments to the current standard output stream with parenthesis syntax. - strings are quoted, escape characters are output. Arguments are output sequentially without space padding. Example:

```
write ^(1 2 (e) 'w')
(1 2 (e) 'w')
```

**display<arg1>...<argn>** Outputs its arguments to the current standard output stream without syntax. - strings are not quoted, escape characters are not output. Arguments are output sequentially without padding. Example:

```
display ^(1 'w\n' (x y))
(1 w
 (x y))
```

### 28.0.2 Global Input Functions

**read** Reads an expression from the standard input.

### 28.0.3 Class Writer

A class which accepts a stream of characters.

#### Methods

**.close()** Closes the current output stream.

**.flush()** Forces all buffered output to be written to the device.

**.format** <format-string><arg1>...<argn> Outputs the args as dictated by the format-string. The format string is a normal string with the special format sequences. Each format sequence must be matched by a corresponding argument to format, used in order.

**%a** Outputs the argument without syntax - strings are unquoted, escape characters are not output

**%s** Outputs the argument using if entered by a user - strings are quoted, escape characters are not output

**%x** Outputs the argument as XML using an XmlWriter

**%u** Outputs the argument as a URL encoded string suitable for HTTP URLS

**%n** Outputs a linefeed

**%%** Outputs a % character

Example:

```
stdout(.format '%s %a %x %n' 'Hello' 'World' ^(img ((width= 23))))

'Hello' World <img width="23"/>
```

**Global Variables**

**stdout** A global variable which holds a *Writer* pointing the current Standard output device, typically the console.

#### 28.0.4 Class Reader

**Globals**

**stdin** A global variable which holds a *Reader* pointing the current Standard Input device, typically the console.

(To be completed)

# 29 Input

## 29.1 stdin

## 29.2 ParenParser

### 29.2.1 .new

### 29.2.2 .close

### 29.2.3 .read

### 29.2.4 .hasData

## 29.3 Parser

## 29.4 StringFormatStream

### 29.4.1 .new

## 29.5 Fetching HTTP Pages

### 29.5.1 web:get <url> [<request headers>]

Content can be fetched from web servers using the web: get function. This takes as input a URL string and an option alist of request headers. It sends the GET request to web server and the returns the response content in a Reader stream. The response must have the 200 'OK' status code otherwise an exception is raised.

Example, fetching the root file from the localhost server on port 80, providing a 'Basic' authentication header.

```
web:get 'http://localhost:8080/' =
        ^(('authorization' = 'Basic Zm9vOmJhcg=='))
```

# 30 Output

## 30.1 write

## 30.2 display

## 30.3 print

## 30.4 format

### 30.4.1 web:get

# 31 File

## 31.1 .new

## 31.2 .open

## 31.3 .format

## 31.4 .close

## 31.5 .hasData

## 31.6 .read

## 31.7 .static-open

# 32 Loading Data in Comma-Separated-Value Files

The builtin *CSV* class is provided for reading data from CSV files.

## 32.1 .read <InStream> field-separator quote-char

The CSV class has a single static method *.read* which takes an input stream as it's first parameter. Despite the name CSV files can come in a variety of formats, hence the method expects the field-seperator character and field quote character as parameters.

The result is a list of rows. Each row is itself a list of the fields parsed.

Example. Read a comma-delimited file:

```
CSV
   .read ((File(.new 'my file')) (.open ^read)) =
       ',' '""'
```

# 33 Running Web Servers

Genyris can run multiple a single-threaded HTTP 1.0 servers in Tasks in the background. The web server is implemented in Java and provides a simple interface.

### 33.0.1 Starting a web server

A Genyris new interpreter can be started as a server with the *httpd* function:

```
httpd <port> <path to source file> [args...]
```

The *httpd* function takes parameters:

- the path of a Genyris source file to load and run
- the TCP/IP port number on which the server will listen for requests.
- all the remaining parameters are collected in a list and bound to *sys.argv*

The return value is a *Task* object for the web server. Many web servers can be started in the same JVM, each server is allocated a unique interpreter instance, there is no sharing between interpreters apart from the *args* passed in.

The URL for the browser would be *http://localhost:8080/*. The server is terminated by killing the thread, or by coding an explicit exit function within the web application.

**Example**

```
httpd 7777 'examples/www-demo.g' 1 2 3 4 5
```

### 33.0.2 Stopping a web server

The server's thread can be stopped with the *kill* functions described above.

### 33.0.3 Processing Web Requests

The web server main loop expects to fnd a function called *httpd-serve* which it calls for each web GET or POST request recieved from clients. The request from the remote client is passed to the function as a single parameter. The return from the function call must contain the response to be sent to the browser.

**Example**

```
df httpd-serve (request)
   list 200 'text/plain' 'Hello World'
```

**Requests - HttpRequest class** The request parameter is an object of class *HttpRequest*. The class has the following methods which return components of the request:

**.getMethod()** GET or POST

**.getPath()** Path component of the UL

**.getHeaders()** an *AList* containing the headers

**.getParameters()** an *aList* containing GET or POST parameters

**.getClient()** gives a Pair containing the client's IP address and hostname

**.getBasicUsernamePassword()** Returns the Basic Authentication scheme header as a dictionary with *.username* and *.password* strings. If no auth is present, the username and passwords are empty strings.

**.toHTML()** a method returning an HTML list structure in %x format.

**Responses** The return value from the *httpd-serve* function is anlaysed by the web server and executed as required. There are two types of response:

- Instruction to serve a file from the file system as a 'static' page
- A data structure to be converted to XHTML and transmittted

**Static Pages** If the return value is a list of the form:

```
SERVE-FILE <root Directory Path> <Path to File> [<list directory flag>]
```

Then the file referred to by the path <root Directory Path>/<Path to File> is served to the client. If the file is a directory, and the <list directory flag> is the symbol *ls*, then the directory listing is returned to the client in HTML. The following function provides a simple static web server:

```
df httpd-serve (request)
    list ^SERVE-FILE '/' (request(.getPath)) ^ls
```

**Http Responses in XML list structures** If the response is a list of the form:

```
<integer status code> <mime type> <tree>
```

then the web server converts <tree> into the relevant encoding and transmits it to the client. The conversion depends on the <mime type>.

**text/plain** This type of response uses the parenthesised Genyris output formatter to display the <tree>. It is equivalent to the %s directive in the %format instruction.

**text/html** This type of response must be in the XML format as used by the *%x* directive in the *format* function. This format requires XML tags to be symbols, followed by an attribute list. Strings an sub-tags are interleaved. The general grammar is:

```
xml=tree :== html-tag ((attribute = 'value')...)
    ['inner text' |<xml-tree> ] ...
```

For example, the expression:

```
u:format '%x'
   quote
     html()
      head()
        title() "Lamp Server"
          body()
             form()
                input((name="op") (size="100") (value =none)) ""
                   verbatim() "   "
                   input((type="submit") (value="Go"))
```

Is converted to the corresponding XML string:

```
<html><head><title>Lamp Server<body><form><input name="op" size="100" value="none">&nbs
```

The *verbatim* pseudo-tag passes text transparently. This example (from *examples/www-demo.g*) echoes the input request in HTML:

```
df httpd-serve (request)
   list 200 'text/html'
      template
          html()
             head()
                title() 'Genyris demo'
             body()
                pre() ,sys.argv
                ,(request (.toHTML))
```

### 33.0.4 Starting a static web server

A static page web server can be started with the *www-static* script:

```
httpd <port> "testscripts/www-static.g" <root directory>
```

# 34 Genyris Classes

## 34.1 Class Hierarchy

insert diagram here.

## 34.2 Thing

### 34.2.1 .vars

### 34.2.2 .self

### 34.2.3 .classes

### 34.2.4 .valid?

## 34.3 StandardClass

### 34.3.1 .classname

### 34.3.2 .subclasses

### 34.3.3 .superclasses

### 34.3.4 .vars

## 34.4 Builtin

## 34.5 String

### 34.5.1 .match

### 34.5.2 .length

### 34.5.3 .split

### 34.5.4 .replace <searchstring> <replacement>

Returns a new string with every occurence of <*searchstring*> replaced with <*replacement*>

### 34.5.5 .+

### 34.5.6 .toBase64

Returns the string converted into a Base64 encoded string. The returned string is tagged with the class string *Base64EncodedString*. For example:

```
> 'foo:bar'(.toBase64)
'Zm9vOmJhcg==' # Base64EncodedString String
```

### 34.5.7 .fromBase64

Decodes a Base 64 encoded string into the binary original. This is also available in the *Base64EncodedString* class as the *.decode* method. Example:

```
> 'Zm9vOmJhcg=='(.fromBase64)
'foo:bar' # String
```

access is generated in the *versioninfo* module which has properties including the title. Example:

```
import versioninfo
print versioninfo!version versioninfo!title
```

### 35.2 symlist

### 35.3 self-test-runner

## 36 Using Java Classes

The Genyris interpreter allows standard Java objects to be created and stored in variable bindings, and allows Java methods to be called from Genyris. Java objects are contained within a builtin 'JavaWrapper' class, whose class list includes a Genyris class with Procedure objects corresponding to the Java methods. This allows Java objects and classes to be used in the same way as Genyris ones.

Java symbol names used for fields, classnames etc have a different syntax to Genyris, so rather than using escaped symbols, the default is to map Java Symbols to Genyris ones. The Genyris name based on the java name with some characters translated as follows:

| Java character | Genyris Character |
|---|---|
| . | _ |
| [ | * |

This is used for class names method names, fields names or anywhere a Java name is used in a Genyris program.

### 36.1 Importing a Java Class with *java:import*

The *java:import* function creates a Genyris class corresponding to a Java class. The properties of the new Genyris class correspond to the methods and constructors of the Java class. The syntax is:

```
java:import <java class string> [as <shorthand name>]
```

The full java class name is supplied as a string. Optionally a shorthand symbol to use as well as the default name can be specified. The default name is translated as described above. The return value is the Genyris class created. This class is bound to the corresponding full name in the global environment. If the *as* clause is present the class is also bound to the shorthand name in the local environment of the caller.

Example :

```
> java:import 'java.lang.Object' as Jobject
 <class java_lang_Object (Java)> # StandardClass
> assert (eq? java_lang_Object Jobject)
```

Continuing with the *java.io.File* example, we can list the method with *.vars*:

```
Jobject.vars
```

produces

```
(.notifyAll .getClass .wait-long-int .equals-java_lang_Object .hashCode .wait-long .cla
```

The methods names include the argument types seperated by a hyphen since Java allows multiple methods with the same name and different arguments (overloading). Constructors, which are nameless in Java, are given the prefix *new-*.

### 36.1.1 Adding to Java Classes

Because *java:import* creates a standard Genyris class, you are free to add, remove, update and delete methods of the class.

## 36.2 Creating Instances of Java Objects

Once a class has been imported, it is then possible to call the constructrs from the Genyris class. The constructor returns a new instance wrapped in a *JavaWrapper* object. For example:

```
> java:import 'java.io.File' as FileJ
> var file (FileJ(.new-java_lang_String '/'))
[java.io.File /] # java_io_File JavaWrapper
```

A call to a Java method involves several steps:

- The arguments are converted to the closest Java type if possible, if a *JavaWrapper* is provided, the embedded Java Object is used.
- The method is called, and exceptions caught
- The return value is converted to a Genyris type if possible, otherwise the return value is placed in a *JavaWrapper* object.
- Returned *JavaWrapper* objects are tagged with the relevant Genyris Java class if it has already been imported.

## 36.3 Accessing Fields

Java objects are contained within *JavaWrapper* objects. When used as a function the the wrapper allows access to the fields of the Java objects using the dynamic symbol approach used for all other Genyris objects. In the above example we can list the Java fields with:

```
> file!vars
(.separatorChar .separator .pathSeparatorChar .pathSeparator
.self .vars .classes .http://www.genyris.org/lang/java#class)
# Pair
```

The special field java:class provides the real Java class of the object.

```
> file!java:class
'java.io.File' # String
```

Read access to the fields is the same as for Dictionaries for example:

```
> file!separator
'/' # String
```

Write access is provided with *setq* as usual, however Java restrictions still apply:

```
> file(setq .separator ';')
*** Error - 'Can not set static final java.lang.String field java.io.File.separator to
```

## 36.4 Calling Methods

Calling Java method is identical to calling Genyris methods. Java static methods are treated the same as Genyris functions, in that they do not require a dynamic environment with *.self* bound. Java methods must be called within the context of a *JavaWrapper* object. Continuing with an example to list the files in a directory:

```
> file(.list)
('sys' 'srv' 'vmlinuz' 'selinux' ...
```

Notice that the Java *list()* method returns an array of *String*, which is automatically converted into a list of Genyris strings.

Parameters are passed au usual, however basic types are automatically converted to Java objects if possible. Refer to the table below for details. For example:

```
> file(.compareTo-java_lang_Object file)
0 # Bignum
```

## 36.5 Conversions and Coersion

The following table lists the conversions of base types in method calls, constructor calls and in fields accesses.:

| Java | Genyris |
|---|---|
| byte | Bignum |
| short | Bignum |
| int | Bignum |
| long | Bignum |
| float | Bignum |
| double | Bignum |
| BigDecimal | Bignum |
| char | String |
| String | String |
| Array | List |
| boolean | NIL or true |
| java.awt.event.ActionListener | <- Function Closure |
| other | JavaWrapper |

### 36.5.1 Conversion To Java Objects

The conversion of Genyris atoms and lists implicit in the calling of methods can be used explicitly with the *java:toJava* function. This takes two arguments, a string containing the name of a Java class and a Genyris object to be converted. The java class naming convention is is used for arrays. For example to convert a list of Bignums into a Java array of Integer:

```
> java:import  '[Ljava.lang.Integer;'
> java:toJava '[Ljava.lang.Integer;' ^(1 2 3)
[[Ljava.lang.Integer; [Ljava.lang.Integer;@148aa23] # *Ljava_lang_Integer; JavaWrapper
```

Note that since we imported the class *[Ljava.lang.Integer;* the conversion automatically tags the *JavaWrapper* with that class.

### 36.5.2 Conversion from Java To Genyris Objects

Basic Java objects are converted into Genyris objects in a straightforward manner. These conversions implicit on the return from Java method calls can be called explicitly using the *java:toGenyris* function, which takes a single argument. For example here we convert a list of bignums to a Java array of Integer and back:

```
> var y     java:toJava '[Ljava.lang.Integer;' ^(1 2 3)
[[Ljava.lang.Integer; [Ljava.lang.Integer;@2087c268] # JavaWrapper
> java:toGenyris y
(1 2 3) # Pair
```

## 36.6 Using Java Reflection from Genyris

Java Reflection methods can be called (just like any other Java interface) to get more detailed information about the Java world. Here is an example which lists every method of *java.lang.class* in detail:

```
> var c
  java_lang_Class
      .forName-java_lang_String 'java.lang.Class'
[java.lang.Class class java.lang.Class] # java_lang_Class JavaWrapper
> for m in (c(.getMethods)) (print m)
[java.lang.reflect.Method public static java.lang.Class java.lang.Class.forName(java.la
...
```

## 36.7 Hooking Swing Events in Genyris

Swing programs can be written in Genyris using the FFI descibed here. All the Swing classes can be accessed as usual, however there is one special case - handling of events from Swing. When Swing has an event to process (typically from a user action such as a mouse click) it needs to call a Genyris function. The Genyris *java:actionListener* function creates a bridging object that holds a closure which can be called later by Swing. The syntax is:

```
java:actionListener <closure>
```

Either lazy or eager procedures can be used in the closure parameter. Here's an example:

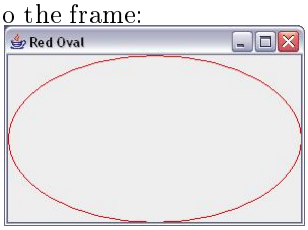
```
def onCancel(event)
    window(.dispose)
cancelButton(.addActionListener-java_awt_event_ActionListener (java:actionListener onCa
```

Here we have supplied an ActionListener to a Button object. The action listener saves the closure created by the *def*. When the user presses the button the onCancel function is called. Notice how the closure captures the *window* binding.

Refer to the source file '*examples/swing.g*' for a complete example which looks like this:


## 36.8 Plotting 2D Figures on a Swing Canvas

Sometimes it's useful to be able to draw shapes and place images on a simple canvas. The supplied examples include *swing-canvas.g* which draws a red oval onto the frame:



To allow the frame to be re-painted when it is moved about or re-sized, a special Java class is provided which allows the *paintComponent()* to be executed in Genyris. Instead of using the *JPanel* component, one uses a *GPanel* component, supplying a closure as an argument. The closure is executed by Swing when it needs the frame to be re-drawn. Let's review code.

First a callback function is defined which takes three parameters, a *Graphics* object (where drawing is done) and and a string:

```
java:import 'java.awt.Graphics' as Graphics
def paint(graphics id command)
  tag Graphics graphics # set up the methods
  cond
     graphics # did we get an object?
         graphics   # ok , draw on it
             .drawLine-int-int-int-int 0 0 100 100
```

Next the special *GPanel* is created. The constructor expects a Genyris closure object:

```
java:import 'org.genyris.java.swing.GPanel' as GPanel
var panel
  GPanel
   .new-org_genyris_interp_AbstractClosure paint
```

Finally the GPanel is added to a frame:

```
frame
    .add-java_awt_Component panel
    .setVisible-boolean true
```

This approach of adding a Java class can be used wherever a *paintComponent* callback is required. The Java code is minimal:

```
public class GPanel extends JPanel {
private static final long serialVersionUID = 1L;
private AbstractClosure repaint;
public GPanel(AbstractClosure closure) {
super();
repaint = closure;
}
public void paintComponent(Graphics g) {
if (repaint != null) {
Exp args[] = {
               JavaUtils.wrapJavaObject(repaint.getEnv(), g)
             };
try {
   repaint.applyFunction(repaint.getEnv(), args);
```

```
} catch (GenyrisException e) {
e.printStackTrace();
}
}
}
}
```

### 36.9 Limitations

The following limitations exist which may be removed in later versions:

- Genuine methods called '*new*' may clash with constructors.
- There may be name collisions between Genyris and Java. e.g. a Java method called *foo-org_ bar_ Class()* would clash with a Java method *foo(org.bar.Class c)*.
- Java Readers and Writers are not converted to Genyris streams.
- The Java class hierarchy is not duplicated in the imported Genyris classes since the method list is a closure over the superclasses of a class.
- There is no universal facility for allowing Java programs to call back Genyris functions, only *java.awt.event.ActionListener* is supported.

## 37 Extending and Modifying Genyris

### 37.1 Adding Builtin Java Functions to Core Genyris Classes

All Genyris functions are implemented as normal Java classes. This include the core classes for data such as Bignum, String etc. For example the Bignum objects are internally implemented as Java BigDecimals. Therefore it is easy to leverage existing Java functionality for the internal representation by using the Java FFI. It suffices to define a Genyris function or method which access the internal object via a reference and manipulates it. Consider:

```
> java:import 'java.math.BigDecimal'
> java:toJava 'java.math.BigDecimal' 123
[java.math.BigDecimal 123] # java_math_BigDecimal
                                    JavaWrapper
```

This expression *(java:toJava 'java.math.BigDecimal' 123)* obtains a reference to the internal BigDecimal object, which is in turn wrapped in a *JavaWrapper* object and tagged with the *java_math_BigDecimal* class. This allows direct access to the underlying object via its methods.

Here is a Genyris method which returns the precision of a Bignum:

```
> Bignum
    def .precision()
        (java:toJava 'java.math.BigDecimal' .self)
             .precision
> 2134(.precision)
4 # Bignum
```

Here is another example, character replacement method for all strings:

```
String
    def .replace(old new)
        (java:toJava 'java.lang.String' .self)
               .replace-char-char old new
> "Shop"(.replace 'p' 'e')
'Shoe' # String
```

# 38 Alist

### 38.0.1 .lookup(key)

### 38.0.2 .getKeys()

### 38.0.3 .hasKey(key)

### 38.0.4 .render()

# 39 Pair

## 39.1 .each

# 40 ListOfLines

# 41 Object

### 41.0.1 .new

### 41.0.2 .init

# 42 Index

# Index