Genyris User Manual

March 10, 2010

Build 598

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 (11) 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 - Conditional Execution

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 *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'*.

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. The *quote* function is a lazy procedure which returns its single argument un-evaluated.

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.

4.4 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* 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
   function (x) (* x x)

def square (x)
   * x x
```

4.4.1 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
```

4.5 Executable Comments

Sometimes it's necessary to temporarily 'comment out' functions or provide in-line 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
```

4.6 Defining Macros

Macros are lazy functions which are very handy for extending the syntax of the language or creating DSLs (Domain-Specific Languages). 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 expession 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*.

4.7 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
```

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

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.

5 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.

5.1 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 Symbol (Dictionary)> # 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.

5.2 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
```

5.3 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
```

5.4 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.

5.5 More Synactic Sugar - Using the exclamation mark[4]

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

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

Input Quote Sequence	Translated Expression
a!b	(a .b)
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
```

6 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".

```
#
# 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...
```

7 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.

7.1 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
```

8 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 variable 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...
```

8.1 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.

8.2 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.

8.3 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].

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

8.4 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
```

8.5 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.

8.6 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
```

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

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

  def .init(args)                          # null .init method
      .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.

8.7 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:

```
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.

[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.

8.8 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.

9 Symbols and Semantic Triples

RDF[8] concepts such as triples and triplestores are available in Genyris as part of the language. Although these were invented for the Semantic Web these concepts 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.

[8] Resouce Description Framework

9.1 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. 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.

9.1.1 Creating triples with *triple*

```
> triple ^Lewis ^hasMother ^Mae
(triple Lewis hasMother Mae) # Triple
```

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

```
var l
   triple ^Lewis ^hasMother ^Mae
l.subject
```

9.1.3 Triple Equality

9.1.4 Accessing Type Information

```
> (23(.asTripleStore))(.asTriples)
(triple 23 type <class Bignum (Builtin)>) # Pair
```

9.2 TripleStores

9.2.1 Creating TripleStores with *TripleStore*

9.2.2 TripleStore equality

9.2.3 Adding Triples with *.add*

9.2.4 Removing Triples *.asTriples*

9.2.5 Selecting Triples with *.select*

9.2.6 Converting dictionaries to TripleStores

```
defvar ^thedict (dict(.a=3)(.b=5))
thedict(.asTripleStore)
```

9.2.7 *.asTriples*

9.3 The Global TripleStore

```
> |http://www.genyris.org/lang/types#Descriptions|(.asTriples)
```

Part II
Reference

10 Compiling From Source

To compile from source you need Ant (I used version 1.7.1). 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
```

test the program via the self-test-runner function from within the interpreter session:

```
> self-test-runner
```

At the end of the run you should see a summary of tests:

```
OK (212 tests)
```

10.1 Using a Mercurial Repository

11 Running Genyris

11.1 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.

11.1.1 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
```

11.1.2 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*.

11.2 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.

11.2.1 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.

11.2.2 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.

11.2.3 Specify a file to run on the command line

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

Arguments are passed after the file name.

11.2.4 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
```

11.3 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'
```

11.4 Tasks - Multiple Threads of Execution

11.4.1 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. No memory or data is shared between interpreters, however 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 I
```

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

11.4.2 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>
```

11.4.3 *ps* - list current running threads

The *ps* function returns a list containg *Task* objects for each of the current Java threads.

11.5 Running Web Servers

Genyris can run multiple a single-threaded HTTP servers run in Tasks in the background.

11.5.1 Starting a web server

A Genyris server can be started 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 whicch 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 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.

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

11.5.2 Stopping a web server

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

11.5.3 Processing Web Requests

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

Example

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

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

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

Http Responses in HTML list structures The 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 structure is converted to the corresponding XML string. The *verbatim* pseudo-tag passes text transparently. For example (from *examples/www-demo.g*):

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

Refer to *examples/notification-server.g* for a more complex example.

11.5.4 Starting a static web server

A static page web server can be started with the *httpd-static* function:

```
httpd-static <port> <path to root directory>
```

The function starts a very simple server with directory listings enabled and no security or logging. For example (Windows):

```
httpd-static 80 'c:/'
```

12 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.

12.1 load *<java resource name>*

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

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

12.2 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.

12.3 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.

12.4 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
```

12.5 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.

```
<body>
```

16.3 Exceptions

16.3.1 raise <expression>

16.3.2 assert

17 Predicates

17.1 Logical Functions

17.1.1 not

17.1.2 and

17.1.3 or

17.2 Comparisons

17.2.1 eq?

17.2.2 equal?

17.2.3 >

17.2.4 <

17.3 null?

18 Closures

18.1 function

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

The *function* macro creates a lexical closure at the point of defiition and returns an 'eager' procedure object. 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 colon 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.

18.2 lambadq

18.3 lambdam

19 Constants

19.1 nil

19.2 EOF

19.3 true

20 Maths Functions

20.1 +

20.2 -

20.3 *

20.4 /

20.5 %

20.6 power

21 Input and Output

21.0.1 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))
```

21.0.2 Global Input Functions

read Reads an expression from the standard input.

21.0.3 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.

21.0.4 Class Reader

Globals

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

(To be completed)

Index

22 Input

22.1 stdin

22.2 ParenParser

22.2.1 .new

22.2.2 .close

22.2.3 .read

22.2.4 .hasData

22.3 Parser

22.4 StringFormatStream

22.4.1 .new

22.5 Fetching HTTP Pages

22.5.1 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=='))
```

23 Output

23.1 write

23.2 display

23.3 print

23.4 format

23.4.1 web:get

24 File

24.1 .new

24.2 .open

24.3 .format

24.4 .close

24.5 .hasData

24.6 .read

24.7 .static-open

25 Time and Dates

25.1 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
```

25.2 now - Current system time in milliseconds since the epoch

Convenience function.

25.3 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
```

25.4 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
```

26 Classification

26.1 class

26.2 tag

26.3 remove-tag <class> <object>

26.4 is-instance?

27 Genyris Classes

27.1 Class Hierarchy

insert diagram here.

27.2 Thing

27.2.1 .vars

27.2.2 .self

27.2.3 .classes

27.2.4 .valid?

27.3 StandardClass

27.3.1 .classname

27.3.2 .subclasses

27.3.3 .superclasses

27.3.4 .vars

27.4 Builtin

27.5 String

27.5.1 .match

27.5.2 .length

27.5.3 .split

27.5.4 .+

27.5.5 .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
```

27.5.6 .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
```