

# Genyris Language Tutorial

Bill Birch

February 8, 2009

[birchb@genyris.org](mailto:birchb@genyris.org)

## **Abstract**

This paper provides an introduction to programming in the Genyris language. The language is a derivative of Scheme with a mostly parenthesis-free syntax yet retaining infix notation. It supports generic programming through macros and object-orientation. It offers a style of object-orientation aligned to the Semantic Web in which objects have many classes and are attached to objects after their construction.

# Contents

<b>1</b>	<b>Getting Started</b>	<b>4</b>
<b>2</b>	<b>Syntax</b>	<b>5</b>
2.1	Comments . . . . .	6
2.2	Numbers . . . . .	6
2.3	Strings . . . . .	6
2.4	Symbols . . . . .	6
2.5	Expressions and Sub-expressions . . . . .	7
2.6	Pairs . . . . .	8
2.7	Quoting and Special Parser Characters . . . . .	9
2.8	Line Continuation . . . . .	10
2.9	More Quote Characters . . . . .	10
<b>3</b>	<b>Variables</b>	<b>11</b>
<b>4</b>	<b>Functions</b>	<b>11</b>
4.1	Eager Functions . . . . .	12
4.2	Lazy Functions - Conditional Execution . . . . .	12
4.3	Defining Your Own Functions . . . . .	13
4.4	Anonymous Functions with Lambda and Friends . . . . .	14
4.4.1	Anonymous Lazy Functions . . . . .	14
4.5	Defining Macros . . . . .	14
4.6	Lexical Scoping Captures Environments . . . . .	15
<b>5</b>	<b>Everything is Callable</b>	<b>16</b>
5.1	Dynamic Variables . . . . .	18
5.2	Working With Pairs and Lists . . . . .	18
5.3	Dictionaries - Your Garden Variety Objects . . . . .	20
5.4	Adding Behaviour to Dictionaries . . . . .	21
<b>6</b>	<b>Namespaces</b>	<b>22</b>

<b>7</b>	<b>Using Classes to Organise Behaviour</b>	<b>22</b>
7.1	Defining Your Own Classes . . . . .	24
7.2	Type-Checked Function Arguments . . . . .	26
7.3	In-Line Type Checks . . . . .	26
7.4	Inheritance and Class Properties . . . . .	26
7.5	Class Validators . . . . .	27
7.6	Traditional Constructors and Factories . . . . .	27
7.7	Automating Classification . . . . .	28
7.8	On Ducks and Interfaces . . . . .	30
<b>8</b>	<b>Semantics and The Secret Life of Symbols</b>	<b>31</b>
<b>9</b>	<b>Function and Class Reference</b>	<b>32</b>
9.1	Input and Output . . . . .	32
9.1.1	Global Output Functions . . . . .	32
9.1.2	Global Input Functions . . . . .	33
9.1.3	Class Writer . . . . .	33
9.1.4	Class Reader . . . . .	33

**Conventions used in this tutorial:**

*italicized* Snippets of Genyris programs

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

# 1 Getting Started

## Installation

Genyris is available as a binary executable Java *jar* file. You don't need to understand Java to use Genyris. You will need the Java 1.5 JRE or later to run the Genyris interpreter. Java can be downloaded from Sun Microsystems. Check your JRE version with this command:

```
$ java -version
```

Start the Genyris command-line interpreter with this command:

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

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

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

## 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:

```
> load "testscripts/testsuite.lin"↵  
↵
```

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

## Running Examples

The release binary file includes some examples which can be extracted as follows:

```
$ jar xvf *.jar examples
```

This creates a directory called “*examples*” with a number of “.lin” files. The files can be edited with your favourite text editor and run with the *include* function. For example, to load and run the “Eight Queens” example do:

```
> include "examples/queens.lin"

~ "file:/home/birchb/workspace/Genyris/examples/queens.lin" ; String
> 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
  = i a
  = j b
  = (- i j) (- a b)
  = (+ 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
- List Pairs
- Parser macros and directives

## 2.1 Comments

All characters following a semi-colon until the end of the line are ignored by the parser. For example:

```
; mark the number 24 as a member of class 'C'  
tag C 24      ; => ~ 24 ; Bignum C
```

## 2.2 Numbers

Numbers can be either integers or floating point with any number of leading or trailing digits<sup>1</sup>. Examples:

```
-3 23.78 -100.0089
```

## 2.3 Strings

Strings are delimited by double quote characters `”`, 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
```

## 2.4 Symbols

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

```
,      comma  
!      exclamation  
:      colon  
'      single quote  
"      double quote  
'      backquote  
@      at sign
```

---

<sup>1</sup>Java *floats* and *doubles* will be added

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<sup>2</sup>. 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 colon `:` 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, and exactly the same as :

```
(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*:

```
(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
```

---

<sup>2</sup>Lisp Cons cells

## 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<sup>3</sup>:

```
list 1 2 'a 3 4 ; evaluates to: (1 2 a 3 4)
```

Quote characters are a shorthand notation to save typing. When the parser sees a single quote, 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))
```

---

<sup>3</sup>*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 and simplest is to use the colon `:` 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 parser also has a 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:

```
quote  
  1 2  
    3  
  ~ 22 ; Pair
```

which is the same as:

```
quote '(1 2 (3) 22)
```

## 2.9 More Quote Characters

Genyris also supports three other special syntactic quotes similar to single quotes. 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>)

### 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"
```

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 evaluation 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 Lambda 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 *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:

```
> lambda (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:

```
> (lambda (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
```

### 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 *lambda* 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 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 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 of 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*.

## 4.6 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<sup>4</sup> creates a function which captures the *balance* variable:

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

---

<sup>4</sup>refer to Abelson and Sussmans' book "Structure and Interpretation of Computer Programs"

```

> 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.

## 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 environment. 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

Generally '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 normal lexical environment. When prefixed with the underscore ! 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
  cons !self !classes

12 <class Bignum (Builtin) ()> ; Pair
```

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) (PRINTWITHCOLON)> ; 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 *PRINTWITHCOLON* 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")) ; PRINTWITHCOLON
```

### 5.3 Dictionaries - Your Garden Variety Objects

Genyris provides “dictionary” objects which are “objects” in the normally understood sense for programming languages. Each dictionary provides a unordered 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 “module” called *file*:

```
;; File Handling Module
define file
  dict
    !name : "File Handling Functions"
    !version : "1.2"
  file
    def !copy(from to) ...
    def !delete(filename) ...
    def !zip(file) ...

;; 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. While it is handy to be able to add behaviour to objects, it does not scale when there are many objects which require the same behaviour. For this we need namespaces or classes.

## 6 Namespaces

A way to prevent name clashes between modules 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 ; Symbol
```

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

```
;; File Handling Module (using prefixes)
@prefix file "genyris:/files/"
def file.copy(from to) ...
def file.delete(filename) ...
def file.zip(file) ...

;; Use of the file module
@prefix f "genyris:/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 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
- !vars
- !classname
- !superclasses
- !subclasses

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

```
Thing
  Builtin
    Bignum
    String
    Pair
      PRINTWITHCOLON
    Dictionary
      Symbol
    Closure
      LazyProcedure
      EagerProcedure
```

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 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...
```

## 7.1 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.

## 7.2 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.

## 7.3 In-Line Type Checks

Genyris provides another mechanism for checking type constraints. If the last element of an expression list is not *nil*, but a class, it 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
  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<sup>5</sup>.

## 7.4 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

---

<sup>5</sup>Genyris supports both “nominative” and “structural” subtyping

the superclasses of all the classes it finds in the classes list. The classes are ordered with those classes deepest in the hierarchy first.<sup>6</sup>

This inheritance of properties is the same regardless of the type of property (method or data). Hence an object can access information stored in its 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
```

## 7.5 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.

## 7.6 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
```

---

<sup>6</sup>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.

## 7.7 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.lin" 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 <sup>7</sup>:

```
load "examples/people.lin"
```

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.

---

<sup>7</sup>The *load* function reads and executes the source file from the Java classpath. Genyris source files are stored within the Java "jar" file including some initialization code and a handful of examples.

## 7.8 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 exists 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.

A simple nominative approach could be used to designate a non-validating 'semantic' class for the interface and tag the objects which allegedly met the requirements. For example:

```
@prefix streams "http://myproject.org/interfaces/2008/streams/"
class streams.IStream()
'streams.open
  var !dc.Description "A stream open method"
  var !rdf.Type 'genyris.Method
```

Usage in another file:

```
@prefix str "http://myproject.org/interfaces/2008/streams/"
class MyStream(str.IStream) ; class declaration ignored at run time - duck typing
  def !str.open() ; prefix on method name resolves ambiguity at run-time
  etc ...
```

An IStream class is defined and the base symbols declares meta-data describing the interface. A user then tags an object with a derived class, the tagger needs to

be honest to ensure the interface is actually present in the tagged objects. Users of the objects could type-check based on the base class *streams.IStream*, but this is unlikely. Duck typing relies on the existence of the correct method names. The use of the prefixed method names helps to avoid ambiguity between semantically different but identically named methods. Hence the *MyStream open* method is not just any *open*, it was declared as *http://myproject.org/interfaces/2008/streams/open*.

## 8 Semantics and The Secret Life of Symbols

As you may have deduced by now, symbols in Genyris are also Dictionaries and can hence have associated bound properties. For example our *file* module might contain the symbol *genyris:/files/* (aka *file*.) with some attached meta-data :

```
@prefix file "genyris:/files/"
'file.
  var !name    "File Handling Functions"
  var !version "1.2"
```

The symbol's properties can be accessed as usual:

```
display ('file.!version)
```

Because symbols have properties, and properties can have prefixes, we can use symbols to define the semantics of programs. The additional meta-data required for semantics are added to the symbols. Properties (in the RDF use of the term) can be defined. Here is a fragment of code which creates a Dictionary containing facts about a Wikipedia article:

```
@prefix rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
@prefix dc  "http://purl.org/dc/elements/1.1/"
dict
  !rdf.about      : "http://en.wikipedia.org/wiki/Tony-Benn"
  !dc.Title       : "Tony Benn"
  !dc.Publisher   : "Wikipedia"
```

The properties of the dictionary are defined by globally published RDF URIs. Here we are invoking the Dublin Core meta-data and the RDF definitions. This allows the programmer to unambiguously identify the properties of an object. By using the meta-data (such as Domain and Range) associated with symbol properties the programmer can even write software to make deductions about the correctness of the program data, add documentation or other do meta-programming.

## 9 Function and Class Reference

A goal of the Genyris language is to encourage re-use and generic programming. The language separates the concerns, form and function, allowing developers to of three kinds of class libraries:

1. Built-in class libraries
2. Taxonomies of class types and validation functions assembled into class hierarchies useful for automatic triage of objects.
3. Traditional executable code libraries containing functions for manipulating objects of known classes and implementing significant functionality.

This section provides more detailed descriptions of individual functions and classes provided with the language.

(To be completed)

### 9.1 Input and Output

#### 9.1.1 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 without 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))
```

### 9.1.2 Global Input Functions

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

### 9.1.3 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

~% Outputs a linefeed

~~ Outputs a tilde character

Example:

```
stdout(!format "~s ~a ~x ~%" "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.

### 9.1.4 Class Reader

#### Globals

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

(To be completed)