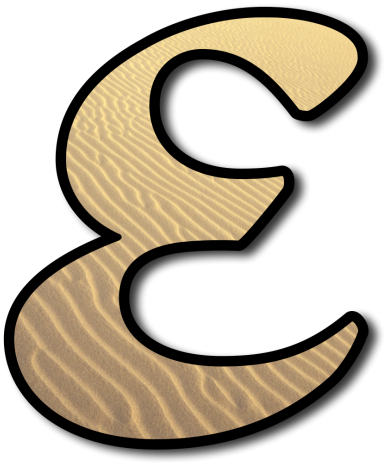
Eidos: A Simple Scripting Language

Benjamin C. Haller

Dept. of Biological Statistics and Computational Biology
Cornell University, Ithaca, NY 14853

Correspondence: bhaller@benhaller.com



Last revised 28 September 2019, for Eidos version 2.3.1 (SLiM version 3.3.1).

Acknowledgements:

Thanks to the designers of the C, R, and Objective-C languages, which paved the way for Eidos in so many ways. Thanks to Terence Parr for his excellent book *Language Implementation Patterns*, which guided much of the implementation of Eidos's tokenizer and parser; and thanks to him also for his grammar of the C language, which provided guidance in the development of Eidos's grammar. Thanks to Jean Bovet and Terence Parr for ANTLRWorks, which was very helpful in designing Eidos's grammar, and which generated the railroad diagrams used in this manual. Thanks to Philipp Messer for believing me when I said that adding a whole language to SLiM would make it much cooler.

Citation:

Haller, B.C. (2016). Eidos: A Simple Scripting Language.
URL: http://benhaller.com/slim/Eidos_Manual.pdf

URL:

http://messerlab.org/slim

License:

Disclaimer:

Contents

1. Eidos overview 5
   - 1.1 Introduction 5
   - 1.2 Why Eidos? 6
   - 1.3 A quick summary of the Eidos language 7
2. Language features 8
   - 2.1 Types, literals, and constants 8
     - 2.1.1 The integer type 8
     - 2.1.2 The float type 8
     - 2.1.3 The logical type 9
     - 2.1.4 The string type 9
   - 2.2 Vectors 10
     - 2.2.1 Everything is a vector 10
     - 2.2.2 Sequences: operator : 10
     - 2.2.3 Concatenation: function c() and type promotion 11
     - 2.2.4 Subsets: operator [] 12
   - 2.3 Expressions 13
     - 2.3.1 Arithmetic expressions: operator +, -, *, /, %, ^ 13
     - 2.3.2 Logical expressions: operator |, &, ! 14
     - 2.3.3 Comparative expressions: operator ==, !=, <, <=, >, >= 15
     - 2.3.4 String concatenation: operator + ... 15
     - 2.3.5 The ternary conditional: operator ?else 16
     - 2.3.6 Nested expressions: precedence and the use of () for grouping 17
   - 2.4 Variables 18
     - 2.4.1 Assignment: operator = ... 18
     - 2.4.2 Everything is a vector (a reminder) 19
   - 2.5 Conditionals 19
     - 2.5.1 The if statement and testing for equality 19
     - 2.5.2 The if–else statement 20
     - 2.5.3 A digression: the semicolon, ; 21
     - 2.5.4 Compound statements with { } 21
   - 2.6 Loops 22
     - 2.6.1 The while statement 22
     - 2.6.2 The do–while statement 22
     - 2.6.3 The for statement 23
     - 2.6.4 The next statement 23
     - 2.6.5 The break statement 24
     - 2.6.6 The return statement 25
   - 2.7 Functions 26
     - 2.7.1 Calling functions: operator () 26
     - 2.7.2 The NULL type 27
     - 2.7.3 The void type 29
     - 2.7.4 The functionSignature() function, and introducing function signatures 30
     - 2.7.5 Optional arguments and default values 31
     - 2.7.6 Function signatures summarized 32
     - 2.7.7 Named arguments in function calls 33
   - 2.8 Objects 34
     - 2.8.1 The object type 34
     - 2.8.2 Element access: operator [] and sharing semantics 34
     - 2.8.3 Properties: operator 35

2.8.4 Multiplexed assignment through properties: operator = revisited 36
2.8.5 Comparison with object: operator ==, !=, <, <=, >, >= revisited 38
2.8.6 Methods: operator . and the methodSignature() method 39
2.9 Matrices and arrays 42
2.9.1 Defining matrices and arrays 42
2.9.2 Matrix and array attributes 43
2.9.3 Using Eidos operators and built-in functions with matrices and arrays 44
2.9.4 Subsets of matrices and arrays using operator [] 46
2.10 Comments 49
2.10.1 Single-line comments with // 49
2.10.2 Block comments with /* */ 49
3. Built-in functions and methods 51
3.1 Math functions 51
3.2 Statistics functions 55
3.3 Distribution drawing and density functions 56
3.4 Vector construction functions 59
3.5 Value inspection & manipulation functions 60
3.6 Value type testing and coercion functions 65
3.7 Matrix and array functions 65
3.8 Filesystem access functions 68
3.9 Color manipulation functions 69
3.10 Miscellaneous functions 71
3.11 Built-in methods 75
4. User-defined functions 77
4.1 Declaring and defining new functions 77
4.2 Complex type specifications 78
4.3 Type specifications that include objects 79
4.4 Optional parameters and default values 81
4.5 Scope with user-defined functions 81
5. EidosScribe 84
5.1 EidosScribe overview 84
5.2 Interactive scripting 85
5.3 File-based script execution 85
5.4 Code completion 85
5.5 Debugging controls 86
5.5.1 Showing tokenization 86
5.5.2 Showing the abstract syntax tree (AST) 87
5.5.3 Showing the evaluation trace 87
6. Colors in Eidos 89
7. Eidos language reference sheet 91
8. Railroad diagrams 95
9. The future of Eidos 101
10. Credits and licenses for incorporated software 102
11. References 104

1. Eidos overview

1.1 Introduction

To get the obvious question out of the way at the outset: Eidos is pronounced "A-dose", with the accent on the long "a" (as in "day"). It is a Classical Greek word (εἶδος) meaning "form", "essence", "type", or "species"; it is the word that Plato used to refer to his Forms.

Eidos is, by design, a very simple and unoriginal language. It is intended to be easy to learn for anyone with any experience with programming – or indeed, even for those with none. For example, the traditional "hello, world" program in Eidos is about as simple as it could possibly be:

```
print("hello, world!");
```

This manual will set out the language in a very methodical and perhaps tedious manner; it may be possible – particularly for those with experience with any ALGOL-based language such as C, Java, or R – to learn Eidos largely from the overview given in section 1.3 and the language reference sheet in section 7. However, reading the rest of this overview is recommended, at least.

Eidos is mostly a hybrid between the C and R languages, with a bit borrowed from Objective-C as well. From R it takes the fact that everything in the language is a vector, and many of the built-in functions are vectorized – they are built to operate directly upon vector variables that contain any number of individual values, making the use of `for` loops and similar constructs unnecessary in many cases. The names and patterns of functions in the `base` package of R are also borrowed liberally, both because they are generally well-designed, and because this borrowing will allow users familiar with R to hit the ground running. R is also the source of the sequence operator, `:`, and the way that subsetting works in Eidos with the `[]` operator. From C, on the other hand, Eidos takes the use of the `.` operator to address object members, the mandatory use of semicolons to terminate statements, zero-based instead of one-based indexing of vectors – and of course a great many properties of its grammar and syntax, which R also inherited from C or its relatives.

However, Eidos also departs from its ancestors – mostly in the direction of simplicity. Unlike C, there are no variable declarations, and typing is entirely dynamic; variables are created simply by assignment, and types are checked only at runtime. There are no compound assignment operators (+=, −=, ∗=, etc.), no scalar logical operators (&&, ||), no bitwise operators (>>, <<, etc.), no pre/post increment or decrement (−−, ++), and no pointers. Although built-in objects exist, you cannot define a new class (or even a `struct` – or even an `enum`). There is no `switch` statement and no `goto`. There is only one scope, the global scope, except that user-defined functions execute in their own private scope. The hope is that this simplicity will not be excessively limiting, since the tasks to which Eidos will be put are expected to be quite simple.

What sort of tasks are those? Eidos is intended to be used to control other software, referred to here as "the Context"; section 1.2 will discuss this in detail.

It is worth noting that Eidos is an interpreted language – it is not "compiled" to the assembly language that is run natively by your computer, but instead "interprets" your code on the fly. You might imagine that, rather than the Eidos interpreter being a native speaker of its own language, it is a tourist frantically looking up words in a dictionary as it encounters them, piecing together meanings from the definitions it finds. This means that Eidos is relatively slow (but not *that* slow; see section 2.6.5). For maximal performance, minimize your use of Eidos, particularly by taking advantage of vectorization when possible rather than writing `for` loops, and using the built-in functions in Eidos to do as much work as possible for you.

Eidos has been a lot of fun to develop; I hope it is a pleasure to use as well. Enjoy!

1.2 Why Eidos?

There are lots of programming languages in the world; do we really need another? Surprisingly, the answer is a resounding YES! Eidos is a rather unusual language, designed to fit a rather unusual niche. There might be another language out there somewhere that would be well-suited to that niche, but certainly no popular mainstream language – C, C++, R, Java, Python, whatever – would fit the bill; the closest is perhaps Lua, but even Lua is not quite right. This section will explain exactly what Eidos is designed to do and why a new language was needed to do it.

First of all, Eidos is intended to provide a scriptable layer on top of existing C++ objects. The idea is that objects that are designed and written in C++, and are instantiated and controlled principally by C++ code, are nevertheless visible and manipulable in Eidos code. These C++ objects are referred to as the Context; a given use of Eidos is tightly integrated with a Context that the script controls. The C++ object and the Eidos object should be, in some sense, one and the same object; scripting proxies, message forwarding, etc., should not be needed. The built-in types in Eidos, similarly, should be C++ types "under the hood", allowing C++ code to implement Eidos functions and primitives without the added complexity of type translation or bridging. Few languages exist that fit this bill; most languages are designed principally to be standalone entities, even if some, like R, provide a "back door" to a lower-level language. Lua is designed to interface with a Context, like Eidos, but since it is ANSI C, providing a front end to C++ would be complex.

Second of all, Eidos is intended to support an interactive workflow; it needs to be an interpreted language so that the user can work at an interactive console prompt, typing Eidos commands and getting results back without a compile-run cycle. This is a reason that C++ itself could not fit the niche of Eidos, even though Eidos is designed to control a C++ Context.

Third, Eidos needs to be cross-platform, open-source, and compatible with the GNU copyleft license. This requirement is because the Context that Eidos is particularly designed to control – the SLiM evolutionary simulation package – has those requirements.

Fourth, Eidos needs to be lean, with as few compile-time and run-time dependencies as possible. This is in part because it needs to be easily buildable on a wide variety of platforms, including places like high performance computing clusters. This is also, in part, because Eidos needs to be tight on memory and quick to load, since it will be used in environments where those resources are scarce. Eidos is thus self-contained; it uses some code from the GNU Scientific Library (GSL) and Boost, but that code has been integrated into Eidos itself, such that Eidos has no link-time or run-time external dependencies.

Fifth, it needs to be blazingly fast to set up and tear down a new interpreter. For some applications, a new interpreter will need to be instantiated millions or even billions of times, and that overhead may be the main execution time bottleneck. For example, in a SLiM simulation a `fitness()` callback may be called for each mutation of a given type, in each individual, in each subpopulation, in each generation, but the callback may do only a very simple calculation to get the resulting fitness value. Each call to the callback needs a fresh Eidos interpreter with its own state variables. Eidos is highly optimized for this usage case.

Sixth, Eidos needs to be very simple to use; the target users are not professional programmers, but rather biologists (in the case of SLiM). Lua and C++, among others, fail this test.

Seventh, it needs to be able to be tightly integrated into a complex custom GUI environment (SLiMgui); the language needs to be designed as a module that can be incorporated into a larger software project that introspects into, and even modifies, language elements and language objects dynamically. Few languages have been designed for this level of integration with other software.

Perhaps this makes the need for Eidos more clear. While it is unfortunate to have to learn yet another language, we hope that the simplicity of Eidos will make the learning curve painless.

1.3 A quick summary of the Eidos language

This section is a one-page summary of the Eidos language, intended for experienced programmers who don't want to slog through the whole manual. If it is gibberish to you, skip it.

Types. Eidos defines six built-in value types: `NULL`, `logical`, `integer`, `float`, `string`, and `object`. `NULL` is similar to a null pointer; it often represents the absence of a well-defined value, often due to an error. The `logical` type represents Boolean values, and may be either true (`T`) or false (`F`). The `integer` and `float` types represent 64-bit integers and IEEE double-precision floating-point values, respectively. The `string` type represents a string of characters; there is no character type in Eidos. Finally, the `object` type represents a C++ object of a particular *class*, as defined by the Context that Eidos is controlling (see section 1.2); Eidos itself defines no `object` classes.

Vectors. All values in Eidos are vectors of zero or more *elements*. Elements themselves are not accessible in Eidos; you can get a vector containing a single element, but you cannot extract the element itself, since Eidos provides no syntax to do so. Internally, elements are C++ types: `bool` for `logical`, `int64_t` for `integer`, `double` for `float`, `std::string` for `string`, and a subclass of `EidosObjectElement` for `object` (`NULL` is always length zero and thus has no element type). A vector of exactly one element is called a *singleton*. Many Eidos operators and functions work with whole vectors, allowing well-designed Eidos code to run fairly quickly.

Expressions. Eidos defines a familiar set of operators. Arithmetic operators include `+` (addition, string concatenation), `-` (negation, subtraction), `*` (multiplication), `/` (division), `%` (modulo), and `^` (exponentiation). A range operator, `:`, produces a vector sequence between its operands. Logical operators include `|` (or), `&` (and), and `!` (not). Comparison operators are `==`, `!=`, `<`, `<=`, `>`, and `>=`, with `!=` being not-equals. A subset operator, `[]`, is provided that is similar to that in R, taking either a `logical` vector specifying which corresponding values to take, or an `integer` vector specifying which indices to take. These operators all work on vector operands, typically allowing operands which are either identical in length (in which case the operation is conducted between matching pairs of elements) or one of which is a singleton (in which case the singleton is paired with every element of the other operand). A non-vectorized ternary conditional operator, `?else`, is also available, similar to the `?:` operator in C/C++. Operator precedence is very similar to C/C++ precedence; parentheses `()` may be used to modify the standard order of evaluation.

Variables. Variables are defined by assigning a value to a symbol with the `=` operator; no declaration is needed. Eidos variables all live in a single global scope, except within user-defined functions, which get their own private scope. However, the Context may choose to modify the Eidos symbol table in a way that resembles scoping, parameter passing, or other such paradigms.

Statements. Statements are semicolon-terminated. Statement types include null statements (a lone `;`), expression statements, assignments, `if` and `if-else` statements, loops (`while`, `do-while`, and an iterator-type `for-in` statement), control-flow statements (`next`, `break`, and `return`, essentially as in R), and compound statements enclosed by braces, `{}`.

Functions. Eidos supplies a broad range of built-in functions (see section 3), and you may also define your own functions within Eidos (see section 4). You may also call a *lambda*, a string that is dynamically interpreted as Eidos code, using functions such as `executeLambda()` and `sapply()`.

Objects. As mentioned above, `object`-type elements are C++ objects defined by the Context. Objects may have Eidos properties and methods (not the same as their C++ instance variables and methods). Properties are lightweight – typically getter/setter access to simple values – whereas methods are heavyweight, performing computation or having wider-reaching effects. The dot operator, `.`, is used both to access object properties with the syntax `x.property` and to call object methods with the syntax `x.method()`. The dot operator is also vectorized, which is often useful.

And there you have it – the Eidos language in one page. Now let's go into more detail.

2. Language features

This section will lay out essentially all of the fundamental grammar and syntax of Eidos, beginning with the basic types of Eidos (section 2.1) and the vector-based approach taken (section 2.2), and then proceeding on to expressions (section 2.3), variables (section 2.4), conditionals (section 2.5), loops (section 2.6), functions (section 2.7), and objects (section 2.8).

2.1 Types, literals, and constants

2.1.1 The `integer` type

The `integer` type is used in Eidos to represent integers – whole numbers, with no fractional component. Unlike in many languages, exponential notation may be used to specify `integer` literals ("literals" means values stated literally in the script, rather than derived through calculations). For example, the following are all `integer` literals in Eidos:

```
0
-179483
12e8
```

The `integer` type is advantageous primarily because it is exact; it does not suffer from any sort of roundoff error. Exact comparison with `integer` constants is therefore safe; roundoff error will not lead to problems caused by `0.999999999` being deemed to be unequal to `1`. However, `integer` is disadvantageous because it can only represent a limited range of values, and beyond that range, results will be unpredictable. Eidos uses 64 bits to store `integer` values, so that range is quite wide; from `-9223372036854775806` to `9223372036854775807`, to be exact. That is broad, but it is still enormously narrower than the range of numbers representable with the next type, `float`.

2.1.2 The `float` type

The `float` type is used in Eidos to represent all non-`integer` numbers – fractions and real numbers. Exponential notation may be used to specify `float` literals; in particular, numeric literals with a decimal point or a negative exponent are taken to be of type `float`. For example, the following are all `float` literals:

```
0.0
-1.2
12e-3
1.2e9
```

Note that this rule means that some literals, such as the first and last shown above, are represented using `float` even though they could also be represented using `integer`.

The `float` type is advantageous primarily because it can represent an enormously wide range of values. Eidos uses C++'s `double` type to represent its `float` values; the range of values allowed will depend upon your computer's architecture, but it will be vast. If that range is exceeded, or if numerical problems occur, type `float` can also represent values as infinity or as "Not A Number" (`INF` and `NAN`, respectively, in Eidos). The `float` type is thus more robust for operations that might produce such values. The disadvantage of `float` is that it is inexact; some values cannot be represented exactly (just as 1/3 in base 10 cannot be represented exactly, and must be written as 0.3333333...). Roundoff can thus cause comparison errors, overflow and underflow errors, and the accumulation of numerical error.

Several `float` constants are defined in Eidos; besides `INF` and `NAN`, `PI` is defined as π (3.14159...), and `E` is defined as e (2.71828...).

2.1.3 The `logical` type

The `logical` type represents true and false values, such as those from comparisons (see section 2.3.3). In many languages this type is called something like `boolean` or `BOOL`; Eidos follows R in using the name `logical` instead.

There are no `logical` literals in Eidos. However, there are defined constants that behave in essentially the same way as literals. In particular, `T` is defined as true, and `F` is defined as false. These are the only two values that the `logical` type can take. As in a great many other languages, these `logical` values have equivalent numerical values; `F` is `0`, and `T` is `1` (and in fact any non-zero value is considered to be true if converted to `logical` type). Values of type `integer` or `float` may therefore be converted to `logical`, and vice-versa, as will be seen in later sections.

2.1.4 The `string` type

The `string` type represents a string of characters – a word, a sentence, a paragraph, the complete works of Shakespeare. There is no formatting on a `string` – no font, no point size, no **bold** or *italic*. Instead, it is just a character stream. A `string` literal must be enclosed by quotation marks, but those may be either single or double quotation marks, `'` or `"`. This simplifies writing Eidos strings that themselves contain quote characters, because you can delimit the string with the opposite kind of quote. For example, `'You say, "Ere thrice the sun done salutation to the dawn"'` is a string that contains double quotes, whereas `"Quoth the Raven, 'nevermore'."` is a string that contains single quotes. Apart from this consideration, it does not matter whether you use single or double quotes; the internal representation is the same. The suggested convention is to prefer double quotes, all else being equal, since they are more universally used in other programming languages.

A complication arises if one wishes to include both single and double quotation marks within a `string`; whichever delimiter you choose, one or the other quote character will terminate the `string` literal. In this case, the quotation marks inside the `string` must be "escaped" by preceding them with a backslash, `\`. The backslash can be used to "escape" various other characters; to include a newline in a string, for example, use `\n`, and to include a tab, use `\t`. Newlines, in particular, can only be included in quoted `string` literals using a `\n` escape sequence, since actual newlines are not legal – but see below for an alternative. Since the backslash has this special meaning, backslashes themselves must be escaped as `\\`. Keeping those escape sequences in mind, these are `string` literals in Eidos:

```
"hello, world"
'this is also a string'
"this\nis\nfive\nlines\nlong."
"this contains a tab\tand a backslash: \\."
'The Foo exclaimed, "foo!"'
'\'I never said most of the things I said.\' – L. P. "Yogi" Berra'
```

Beginning users may wish to skip the rest of this subsection, but for more advanced users, there is another way of representing `string` literals in Eidos. Sometimes the limitations and complications of standard `string` literals just get in the way; you just want a block of text to be interpreted as a `string` literal, including any newlines that it contains, and without any quoting issues, any special interpretation of escape sequences, and so forth. This is often desirable for including a snippet of Eidos code as a `string` literal in your code, for example (perhaps because you plan to pass the `string` to a function like `sapply()` or `executeLambda()` that will interpret it as executable code; see section 3). For this purpose, Eidos provides a multiline `string` literal format, roughly following a style commonly called a "here document" in other languages such as Perl, PHP, and Ruby. The idea of these multiline "here document"-style `string` literals is quite simple; since a code sample is worth a thousand words, let's start with an example:

```
<<---
hello,
world
>>---
```

That is exactly equivalent to the quoted string literal `"hello,\nworld"`; the start delimiter `<<---` and the newline following it are removed, the `>>---` delimiter and the newline preceding it are removed, and everything remaining that was between the delimiters is taken literally, including the newline after the comma.

Stated more precisely, an Eidos multiline string literal starts with a user-defined delimiter of the form `<<DELIMITER`, where `DELIMITER` may be any sequence of characters whatsoever, or may be zero-length. The `<<DELIMITER` start delimiter must be immediately followed by a newline. The contents of the `string` literal begin after that newline, and continue until the point at which a newline followed by an end delimiter is encountered, where the end delimiter is of the form `>>DELIMITER` (`DELIMITER` being the same character sequence as in the start delimiter). The start and end delimiters thus comprise a matched pair, like `<<FOO` and `>>FOO`, `<<==========` and `>>==========`, or simply `<<` and `>>`. As long as you choose a delimiter that does not occur within your string literal's text, there will be no problem, and no quoting or escaping will be necessary. Indeed, character sequences such as `\n` that would be interpreted as special escape sequences in a quoted `string` literal will be treated just like any other characters in a multiline `string` literal.

2.2 Vectors

2.2.1 Everything is a vector

In Eidos (following R), everything is a vector. A vector is simply an ordered collection of zero or more values of the same type. That may sound rather arcane; the point is that when you write `9` in Eidos, you are not, in fact, referring to a single integer of value `9`. Rather, you are referring to an `integer` vector; that vector happens to contain one value, which is `9`. Often, if you are doing calculations with single values (called "scalars"), you can ignore this fact – but not always. It might seem like an unnecessary complication, at first blush, that you cannot simply work with scalar values; but in fact the opposite is true. If Eidos supported scalars in addition to vectors, it would have twice as many data types as it has; and if it supported only scalars, and not vectors, then a great deal of useful functionality would be missing from the language. Once you get used to working with vectors, it will come to seem very intuitive. There are good reasons why R is a vector-based language, and why Eidos follows in its footsteps.

So `9` is an `integer` vector containing one value, `9`; OK. (In Eidos a vector containing exactly one value is called a "singleton", by the way; we will be using that term a lot.) How do we get vectors with more than one value, then? Section 2.7 and section 3 will introduce a variety of built-in functions that can produce vectors, but until then, we will follow the lead established in the next two sections, which will introduce an operator and a function that both produce new vectors.

2.2.2 Sequences: operator `:`

An "operator" is an entity that performs an operation on operands. That probably sounds deliberately obscure, but in fact it is quite simple. In the statement "`1+1` equals `2`", the `+` symbol is an operator, and the two `1` values are the operands upon which the `+` operator acts. The `+` operator performs addition; it adds its operands. Thus do programming-language geeks talk about things. (The word "equals" is actually an operator here also; see section 2.3.3).

The first Eidos operator we'll discuss is the `:` operator. It is used to construct vectors with (usually) more than one value. In particular, it is used to construct *sequences*, and so it is called the sequence operator. Given operands `x` and `y` (standing for any two numbers), the sequence

operator starts at `x` and counts, by `1` (or `-1`, as appropriate) toward `y` without passing it. It yields a vector containing all of the numbers it encounters along the way.

To illustrate this, we'll look at an interactive session in EidosScribe, the interactive Eidos interpreter (described in detail in section 5). Here, lines beginning with a `>` have been entered by the user; the `>` character is the "prompt" shown by EidosScribe to request input from the user, so the user did not type the `>` character but did type what follows it. The lines following show the result produced by Eidos as a result of interpreting the user's input:

```
> 1:5
1 2 3 4 5
> 5:1
5 4 3 2 1
> -1.2:6.5
-1.2 -0.2 0.8 1.8 2.8 3.8 4.8 5.8
```

Note that the sequence operator can count down as well as up, that it can handle `float` as well as `integer` operands, and that negative numbers are allowed.

2.2.3 Concatenation: function `c()` and type promotion

This section will provide a sneak preview of *functions*, which are covered in more depth in section 2.7. Here we will look at just one function, called `c()`. The "`c`" stands for "concatenate", meaning to paste together end-to-end, and that is exactly what `c()` does. You can supply it with any number of values, as a comma-separated list inside its parentheses, and `c()` will stick them together and give you back a single vector. For example, here is another interactive session within EidosScribe:

```
> c(1, 5, 9, 18392, -17, 3)
1 5 9 18392 -17 3
> c(6.5, 3:9, 0)
6.5 3 4 5 6 7 8 9 0
> c(5:7, "foo")
"5" "6" "7" "foo"
```

The first example shows that `c()` simply returns a vector with the values it was passed, in order. It might look like it did nothing at all; but in fact it turned the things that were passed to it – six `integer` vectors, each containing one value – into a single `integer` vector containing six values.

The second example shows that you can supply `float` values to `c()` as well, and even sequences constructed with the sequence operator, and it will do the same thing: paste them together, end to end, to make a single vector.

The third example shows what happens if you mix types: `integer` and `string`, here. The `c()` function produces a single vector, necessarily of a single type. Type `string` cannot be changed to type `integer`, in general (what `integer` would `"foo"` be?), so instead the `integer` values have to be changed to `string`, and a `string` vector results. In fact, a similar thing happened in the second example; `c()` was given a mix of `float` and `integer`, and it converted the `integer` values to `float` to avoid losing information, producing a float vector as a result. These are both examples of what is called "type promotion", which Eidos does automatically in some cases.

A digression about type promotion in Eidos might be called for here, while we are on the subject. This digression will reference many topics that are covered later in this manual, so you might wish to skip it for now if you are not an experienced programmer. Type promotion is done by Eidos (1) when multiple values of different types are concatenated together to form a single vector, as done by the `c()` function, the `sapply()` function (see section 3.9), and the results from method calls (see section 2.8.6), (2) when values of different types are compared using the

comparative operators ==, !=, <, <=, >, and >= (see section 2.3.3), and (3) with the string concatenation operator +, which promotes its operands to `string` (see section 2.3.4). The arithmetic operators +, –, *, /, %, and ^ (see section 2.3.1) also generally accept a mixture of `integer` and `float` operands, promoting `integer` to `float` as needed; they will not promote `logical` upward, however. Conceptually, the arithmetic operators work with "numeric" type, which is either `integer` or `float`. Many functions also effectively work with "numeric" type, by declaring their parameters to accept either `integer` or `float`, but this is not actually automatic type promotion so much as a policy choice to accept parameters of particular types, so the function signature (see section 2.7.4) of a given function should be checked to see what types that function allows. Apart from these specific cases, automatic type promotion is not done in Eidos; it is not done for the operands of operators, or the parameters of functions, for example, except for the specific cases mentioned here. When automatic type promotion does apply, it will promote types upward according to a strict hierarchy: `logical` is the lowest type on the hierarchy and can be promoted upward to `integer`, `float`, or `string`, (2) `integer` is next and can be promoted to `float` or `string`, (3) `float` can be promoted upward to `string`, and (4) `string` cannot be promoted at all since it is the highest type in the hierarchy. Type `object` (see section 2.8) does not participate in automatic type promotion at all; it is not promoted to any other type, and other types are never promoted to it. It should also be noted that when Eidos expects a `logical` value, such as in `if`, `while`, and `do–while` statements (see sections 2.5 and 2.6) or with the logical operators &, |, and ! (see section 2.3.2), values of `integer`, `float`, and `string` type will be interpreted as either T or F according to specific rules (see section 2.3.2); this is not automatic type promotion, however, but rather type coercion applied by those statements and operators. In any case, you can always explicitly convert Eidos values from one type to another using the `as...()` family of functions (see section 3.6). Again, this digression brought in many future concepts, so don't worry if it didn't make sense; it just seemed best to summarize type conversion in Eidos in one place.

2.2.4 Subsets: operator `[]`

While we are on the topic of vectors, it makes sense to introduce the operator in Eidos that most explicitly operates on vectors: the `[]` operator. This operator selects a subset of the vector upon which it operates; it is thus often called the subset operator. It can work in one of two different ways, depending upon whether it is given an `integer` vector of indices, or is given a `logical` vector of selectors. These two methods will be described below.

First of all, a subset can be selected with an `integer` vector of indices. These indices are zero-based, like C but unlike R; the first value in a vector is thus at index `0`, not index `1`. Here, for example, are a few subset operations, also using the methods of vector construction discussed in the previous sections:

```
> 10:19
10 11 12 13 14 15 16 17 18 19
> (10:19)[5]
15
> (10:19)[c(2,3,5,3,3,9)]
12 13 15 13 13 19
```

Note that a given index can be used multiple times.

Second, a subset can be selected with a `logical` vector of selectors. In this case, the `logical` vector must be the same length as the vector being selected; each `logical` value indicates whether the corresponding vector element should be selected (`T`) or not (`F`). For example:

```
> (10:19)[c(T,T,F,F,F,T,F,F,F,T)]
10 11 15 19
```

This may look a bit clumsy, shown as it is here with a `logical` vector constructed with `c()`; but it is, in fact, enormously powerful and useful when combined with the power of expressions, as explored in the next section.

2.3 Expressions

2.3.1 Arithmetic expressions: operator +, –, ∗, /, %, ^

These are the standard operators of arithmetic; `+` performs addition, – performs subtraction, `∗` performs multiplication, `/` performs division, `%` performs a modulo operation (more on that below), and `^` performs exponentiation. Not a great deal needs to be said about these operators, which behave according to the standard rules of mathematics. They also follow the standard rules of mathematical "precedence"; exponentiation is the highest precedence, addition and subtraction are the lowest precedence, and the other three are in the middle, so `4^2+5∗6^7` is grouped as `(4^3)+(5∗(6^7))`, as expected if you remember your grade-school math.

There are only a few minor twists to be discussed. One is the meaning of the `%` operator, which many people have not previously encountered. This computes the "modulo" from a division, which is the remainder left behind after division. For example, `13%6` is `1`, because after `13` is divided evenly by `6` (taking care of `12` of the `13`), `1` is left as a remainder. Probably the most common use of `%` is in determining whether a number is even or odd by looking at the result of a `%2` operation; `5%2` is `1`, indicating that `5` is odd, whereas `6%2` is `0`, indicating that `6` is even.

Another twist is that both the division and modulo operators in Eidos operate on `float` values – even if `integer` values are passed – and return `float` results. (For those who care, division is performed internally using the C++ division operator `/`, and modulo is performed using the C++ `fmod()` function). This policy was chosen because the definitions of integer division and modulo vary widely among programming languages and are contested and unclear (see Bantchev 2006, http://www.math.bas.bg/bantchev/articles/divmod.pdf). If you are sure that you want `integer` division or modulo, and understand the issues involved, Eidos provides `integerDiv()` and `integerMod()` for this purpose (see section 3.1). Besides side-stepping the vague definitions of the `integer` operator, this policy also avoids bugs involving the accidental use of `integer` division when `float` division was desired – a much more common occurrence than *vice versa*.

A third twist is that `+` and – can both act as "unary" operators, meaning that they are happy to take just a single operand. This is standard math notation, as in the expressions `–6+3` or `7∗–5`; but it can sometimes look a bit strange, as in the expression `5––6` (more easily read as `5 – –6`).

A fourth twist is that the `^` operator is right-associative, whereas all other binary Eidos operators are left-associative. For example, `2–3–4` is evaluated as `(2–3)–4`, not as `2–(3–4)`; this is left-associativity. However, `2^3^4` is evaluated as `2^(3^4)`, not `(2^3)^4`; this is right-associativity. Since this follows the standard associativity for these operators, in both mathematics and most other programming languages, the result should generally be intuitive, but if you have never explicitly thought about associativity before you might be taken by surprise. See section 2.3.6 for further discussion of associativity, as well as operator precedence, a related topic.

A fifth twist is that the arithmetic operators and functions in Eidos are guaranteed to handle overflows safely. The `float` type is safe because it uses IEEE-standard arithmetic, including the use of `INF` to indicate infinities and the use of `NAN` to represent not-a-number results, as in most languages. In Eidos, however, the `integer` type is also safe, unlike in C, C++, and many other languages. All operations on `integer` values in Eidos either (1) will always produce `float` results, as the `/` and `%` operators do; (2) will produce `float` results when needed to avoid overflow, as the `product()` and `sum()` functions do; or (3) will raise an error condition on an overflow, as the Eidos operators `+`, `–`, and `∗` do, as well as the `abs()` and `asInteger()` functions. This means that the `integer` type in Eidos can be used without fear that overflows might cause results to be incorrect.

The final twist is really a reminder: *everything is a vector*. These operators are designed to do something smart, when possible, with vectors of any length, not just with single-valued vectors as shown above. Here are a few examples, which will give you a sense of how it works:

```
> (1:10)+10
11 12 13 14 15 16 17 18 19 20
> (1:10)*5
5 10 15 20 25 30 35 40 45 50
> (1:10)*(10:1)
10 18 24 28 30 30 28 24 18 10
> (1:10)%2
1 0 1 0 1 0 1 0 1 0
```

In general, the operands of these arithmetic operators must either be the same length (in which case the elements in the operand vectors are paired off and the operation is performed between each pair), or one or the other vector must be of length `1` (in which case the operation is performed using that single value, paired with each value in the other operand vector). The examples above will be easier to understand than the previous sentence was.

2.3.2 Logical expressions: operator |, &, !

The `|`, `&`, and `!` operators act upon `logical` values. If they are given operands of other types, those operands will be "coerced" to `logical` values following the rule mentioned above: zero is `F`, non-zero is `T` (and for `string` operands, a `string` that is zero characters long – the empty string, `""` – is considered `F`, while all other `string` values are considered `T`).

As to what they do: `|` is the "or" operation, `&` is the "and" operation, and `!` is the "not" operation. As in common parlance, "or" is `T` if either of its operands is `T`, whereas "and" is `T` only if both of its operands are `T`. The "not" operator is unary (it takes only one operand), and it negates its operand; `T` becomes `F`, `F` becomes `T`. As with the arithmetic operators, these operators work with vector operands, too – either matching up values pairwise between the two operands, or applying a single value across a multivalued operand. Some examples:

```
> T & F
F
> T | F
T
> c(T, F) & F
F F
> c(T, F) | F
T F
> c(T, T, F, F) & c(T, F, T, F)
T F F F
> c(T, T, F, F) | c(T, F, T, F)
T T T F
```

Those familiar with programming might wish to know that the `|` and `&` operators do not "short-circuit" – they can't, because they are vector operators. If the `&` operator first sees an operand that evaluates to `F`, for example, it knows that it will produce `F` value(s) as a result; but it does not know what size result vector to make. If a later operand is a multivalued vector, the `&` operator will produce a result vector of matching length; if all later operands are also length `1`, however, `&` will produce a result vector of length `1`. To know this for sure (and to make sure that there are no illegal length mismatches between later operands), it must evaluate all of its operands; it cannot short-circuit. Similarly for the `|` operator.

These semantics match those in R, for its | and & operators, but they might seem a little strange to those used to C and other scalar-based languages. For those used to R, on the other hand, it should be noted here that Eidos does not support the && and || operators of R, for reasons of simplicity; it is safer to use the `any()` or `all()` functions described in section 3.5 to simplify multivalued `logical` vectors before using & or |. If this is gibberish to you, it is not important; the point here is only to prevent confusion among users accustomed to R.

2.3.3 Comparative expressions: operator ==, !=, <, <=, >, >=

These operators compare their left and right operand. The operators test for equality (==), inequality (!=), less-than (<), less-than-or-equality (<=), greater-than (>), and greater-than-or-equality (>=) relationships. As seen above with the arithmetic and logical operators, this can work in two different ways: if the operands are the same length, their elements are paired up and the comparison is done between each pair, whereas if the operands are not the same length then one operand must be a singleton (i.e., of length 1), and its value is compared against all of the values of the other operand. Regardless of the types of the operands, these operators all produce a `logical` result vector. If the operands are of different types, promotion will be used to coerce them to be the same type (i.e. `logical` will be coerced to `integer`, `integer` to `float`, and `float` to `string`, as previously discussed in the context of the `c()` function in section 2.2.3).

This is all pretty straightforward, so a few examples should suffice to make it clear:

```
> 5 == 5
T
> 5 == 6
F
> 5 == "5"
T
> 1:5 == 4
F F F T F
> 1:5 == 5:1
F F T F F
```

Note that this is often not what you want! You might not want the automatic type promotion that makes `5=="5"` evaluate as `T`, or the vectorized comparison that makes `1:5==4` evaluate as something other than simply `F`. You might really want to ask: are two values *identical?* With that as prelude, see the discussion at section 2.5.1 for an alternative to operator == and operator != that is a better fit in many situations.

2.3.4 String concatenation: operator +

The + operator was previously discussed as an arithmetic operator in section 2.3.1, but it can also act as a concatenation operator for `string` operands. Concatenation is pasting together; the + operator simply pastes its `string` operands together, end to end:

```
> "foo" + "bar"
"foobar"
```

In fact, this works with non-`string` operands too, as long as a `string` operand is adjacent; the interpretation of + as a concatenation operator is preferred by Eidos, and wins out over its arithmetic interpretation, as long as a `string` operand is present to suggest doing so. The other non-`string` operands will be coerced to `string`:

```
> 3 + " + " + 7 + " equals " + 10
"3 + 7 equals 10"
```

However, this does not work retroactively; if Eidos has already done arithmetic addition on some operands, it will not go back and perform concatenation instead:

```
> 3 + 7 + " equals " + 10
"10 equals 10"
```

To force concatenation, you can begin the expression with an empty string, "":

```
> "" + 3 + 7 + " equals " + 10
"37 equals 10"
```

Which is not a true statement; but it is a correct concatenation operation!

The concatenation operator also works with vectors, as usual:

```
> (99:97) + " bottles of beer on the wall..."
"99 bottles of beer on the wall..." "98 bottles of beer on the wall..."
"97 bottles of beer on the wall..."
```

Beginning with Eidos 2.2, string concatenation involving `NULL` concatenates the `string` value `"NULL"`, just as if `NULL` were a singleton `string` vector containing that value:

```
> "The result is " + NULL + "!"
"The result is NULL!"
```

2.3.5 The ternary conditional: operator `?else`

This operator is an advanced topic that assumes an understanding of many concepts not yet discussed; this section should therefore probably be skipped by beginning users of Eidos.

Eidos, like many languages, has an `if` statement that can be used to specify conditional execution of statements (see section 2.5.1), and an `if-else` construct can be used to provide an alternative code path (section 2.5.2). Sometimes, however, one wishes to have conditional execution of an expression, rather than an entire statement. The `if-else` construct is particularly inconvenient with assignments involving complex lvalues, such as:

```
if (condition)
    x[index].property = a;
else
    x[index].property = b;
```

It is desirable to provide a way for the user to specify that the choice of rvalue, `a` or `b`, should depend upon `condition` without having to duplicate the lvalue and the assignment. The R language provides this functionality by making `if-else` statements result in an rvalue, like an expression. The C language, on the other hand, provides a *ternary conditional* operator, `?:`, that can be used in expressions to much the same effect. Eidos straddles the gap with a ternary conditional operator, `?else`, that uses the `?` initiator of C, but the `else` token as a continuation as in R. In the syntax of Eidos, the above conditional assignment can be rewritten as:

```
x[index].property = condition ? a else b;
```

This will evaluate `condition` and result in `a` if `condition` is `T`, or `b` if `condition` is `F`. That result is then assigned into the lvalue. Note that, as in C, the precedence of the ternary conditional operator is very low, but higher than operator `=`, so that parentheses are often not needed to group statements of this type (see section 2.3.6). The `else` clause of the ternary conditional is required; there is no equivalent of an `if` statement without an `else`, since an rvalue must be produced.

Just as with `if-else` statements, only the selected subexpression, as determined by the condition, is evaluated; the other subexpression will not be evaluated, so any side effects it might have will not occur. For example, with the statement:

```
x = condition ? f1() else f2();
```

here `f1()` will be called if `condition` is `T`, `f2()` if `condition` is `F`; only the subexpression selected by the condition is evaluated, and so it is never the case that both `f1()` and `f2()` are called.

Ternary conditionals may be nested. Because the operator is right-associative (see section 2.3.6), an expression such as:

```
z = (a == b ? a else b ? c else d);
```

is grouped as:

```
z = (a == b ? a else (b ? c else d));
```

rather than

```
z = ((a == b ? a else b) ? c else d);
```

This is generally desirable, since it provides a flow similar to chaining of `if-else if-else` statements. In any case, parentheses may be used to change the order to evaluation as usual.

2.3.6 Nested expressions: precedence and the use of `()` for grouping

All of the discussion above involved simple expressions that allowed the standard precedence rules of mathematics to determine the order of operations; `1+2*3` is evaluated as `1+(2*3)` rather than `(1+2)*3` because the `*` operator is higher precedence than the `+` operator. For the record, here is the full precedence hierarchy for operators in Eidos (including a few operators that have not yet been discussed), from highest to lowest precedence:

`[]`, `()`, `.`	subscript, function call, and member access
`+`, `-`, `!`	unary plus, unary minus, logical (Boolean) negation *(right-associative)*
`^`	exponentiation *(right-associative)*
`:`	sequence construction
`*`, `/`, `%`	multiplication, division, and modulo
`+`, `-`	addition and subtraction
`<`, `>`, `<=`, `>=`	less-than, greater-than, less-than-or-equality, greater-than-or-equality
`==`, `!=`	equality and inequality
`&`	logical (Boolean) and
`|`	logical (Boolean) or
`?else`	ternary conditional *(right-associative)*
`=`	assignment

Operators at the same precedence level are generally evaluated in the order in which they are encountered. Put more technically, Eidos operators are generally left-associative; `3*5%2` evaluates as `(3*5)%2`, not as `3*(5%2)`. The only binary operator in Eidos that is an exception is the `^` operator, which (following standard convention) is right-associative; `2^3^4` is evaluated as `2^(3^4)`, not `(2^3)^4` (see section 2.3.1). The unary `+`, unary `-`, and `!` operators are also technically right-associative; this just implies that the operators must occur to the left of their operand (`-x`, not `x-`). The ternary conditional operation is also right-associative, as discussed in section 2.3.5.

In any case, parentheses can be used to modify the order of operations, just as in math:

```
> 1+2*3
```

```
7
> (1+2)*3
9
> ((1+2)*3)^4
6561
```

Note that this use of parentheses is distinct from the `()` operator, involved in making function calls and discussed in section 2.7 (and which we saw briefly in section 2.2.3 with `c()`).

2.4 Variables

2.4.1 Assignment: operator `=`

Thus far, we have only used expressions in an immediate sense: evaluating them and seeing their result print in Eidos's console. But the results of expressions can also be saved in variables. As in many languages, this is done with the = operator, often called the assignment operator. Here's an example of the use of a variable:

```
> x = 2 ^ 5
> x
32
> x + 20
52
> x
32
> x = x + 20
> x
52
> x == 52
T
```

Note that the expression `x + 20` did not change the value of `x`; a new assignment, `x = x + 20`, was necessary to achieve that.

As the example above illustrates, the assignment operator, `=`, is different from the equality comparison operator, `==`. In many languages, confusing the two can cause bugs that are hard to find; in C, for example, it is legal to write:

```
if (x=y) ...
```

In C, this would assign the value of `y` to `x`, and then the expression `x=y` would evaluate to the value that was assigned, and that value would be tested by the `if` statement. This can be useful as a way of writing extremely compact code; but it is also a very common source of bugs, especially for inexperienced programmers. In Eidos using assignment in this way is simply illegal; assignment is allowed only in the context of a statement like `x=y;` to prevent these issues. (This point is mostly of interest to experienced programmers, so if it is unclear, don't worry.)

Variable names are fairly unrestricted. They may begin with a letter (uppercase or lowercase) or an underscore, and subsequently may contain all of those characters, and numerical digits as well. So `x_23`, `fooBar`, and `MyVariable23` are all legal variable names (although not good ones – good variable names explain what the variable represents, such as `selection_coeff`). However, `4by4` would not be a legal variable name, since it begins with a digit.

Functions will be discussed in depth in section 2.7 and chapters 3 and 4, but here's a bit of foreshadowing: you can use the `ls()` function to *list* (thus the name of the function) all the variables that have been defined:

```
> ls()
E => (float) 2.71828
F => (logical) F
INF => (float) INF
NAN => (float) NAN
NULL => (NULL) NULL
PI => (float) 3.14159
T => (logical) T
x <-> (float) 52
```

(You might see some other variables listed as well, depending on your version of Eidos and the context in which you execute the statement.) Here `x` was defined by the code we just ran above; the rest of the variables listed are constants that are predefined by Eidos, as discussed in section 2.1. Note that variables you have defined are shown with a `<->` arrow; that indicates that they may be changed. Constants are shown with a `=>` arrow, simply indicating that they are constant. (Some constants are predefined in Eidos, but you can also define your own constants using the `defineConstant()` function, as will be seen in section 3.9.)

In the listing above, the type of each variable is also given. You might notice that there is a constant, `NULL`, of a type, also called `NULL`, that we have not yet discussed; that topic will be taken up in section 2.7.2.

2.4.2 Everything is a vector (a reminder)

Since it is so central to Eidos, it is worth a reminder here that every value in Eidos is a vector. Variables therefore contain vectors, too; they might have just a single value in them (or no values at all!), but they are vectors nonetheless. You can find out the number of values in a vector using the `size()` function (or `length()`, which is synonymous, for R users) – this is jumping ahead a bit, again, but it is simple enough:

```
> x = 17:391
> size(x)
375
```

2.5 Conditionals

2.5.1 The `if` statement and testing for equality

As in many languages, conditional execution is provided by the `if` statement. This statement is supplied with a `logical` condition; if the condition is `T`, the rest of the `if` statement is executed, whereas if the condition is `F`, the rest of the `if` statement is ignored. An example:

```
> if (2^2^2^2^2 > 10000) "exponentiation is da bomb!"
"exponentiation is da bomb!"
```

The only twist here, really, is that the condition must evaluate to a singleton, i.e. a vector of `size() == 1`. The `if` statement, in other words, is essentially a scalar operator, not a vector operator. If you have a multivalued `logical` vector, you can use the `any()` or `all()` functions to simplify it to a single `logical` value; see section 3.5. Alternatively, the `ifelse()` function provides a vector conditional operation, similar to that in R; see section 3.5.

As an example, suppose you have a variable `x` which ought to be equal to `3`, and a variable `y` which ought to contain two values, `7` and `8`. You might expect to be able to write:

```
> if (x == 3 & y == c(7,8)) "yes!"
ERROR (EidosInterpreter::Evaluate_If): condition for if statement has
size() != 1.
```

This is the first time we've seen a Eidos error in this manual; notice they print in red in EidosScribe. Their wording can be a bit arcane; apologies in advance for that. In this case, the error informs you that the size of the condition is not equal to `1` (and that that is a problem). The expression `y == c(7,8)` produces a `logical` vector with two values, the result of testing the first and second values respectively. The `&` operator thus produces a two-valued `logical` vector as its result, and `if` is not happy about that. To resolve this, you could use the `all()` function, as in:

```
if (x == 3 & all(y == c(7,8))) "yes!"
```

That makes it clear that you require *all* of the results of the `y == c(7,8)` comparison to be true, not just *any* of those results (for which you would use `any()`). Without that clarification, Eidos doesn't know what to do. (R has the same difficulty; it handles it by issuing a warning ("the condition has length > 1 and only the first element will be used") and testing only the first element of the `logical` condition vector – a behavior that is probably almost never what one wants.)

You might wonder why Eidos and R don't just take the obvious route of requiring all values to be true by default. The difficulty is that the semantics of the `&` operator do not cooperate. Suppose `x` is not a singleton, but instead has three values; your `if` statement would then become:

```
if (x == c(3,4,5) & y == c(7,8)) "yes!"
```

After the == operators have been evaluated, this boils down to a use of the `&` operator on two `logical` vectors, which might look like:

```
c(T,T,T) & c(T,T)
```

And that is not legal, because the operands to operator `&` differ in size and neither is a singleton; Eidos will give you an error ("`ERROR (EidosInterpreter::Evaluate_And): operands to the '&' operator are not compatible in size()`"). There is no good way to resolve that without breaking the semantics of the `&` operator in ways that would have deep and undesirable consequences in other areas. So it really is necessary to use `all()`, as in:

```
if (all(x == c(3,4,5)) & all(y == c(7,8))) "yes!"
```

However, there is a better solution! What this all points out, really, is that operator == is not a very good way to test whether one value is identical to another value, because it does an elementwise equality comparison and returns a vector with one `logical` value per element (see section 2.3.3). Eidos and R both provide a better way to test whether two values are identical: the `identical()` function. Since we haven't covered functions yet, this jumps ahead a bit, but its usage here is very simple. You could write the `if` statement above as:

```
if (identical(x, c(3,4,5)) & identical(y, c(7,8))) "yes!"
```

That will do precisely what you want. As an added bonus, it will also test that `x` and `y` have the correct type, whereas operator == will use automatic type promotion in its test; `7=="7"` is `T`, but `identical(7, "7")` is `F`. If you find yourself using operator == or operator `!=` in Eidos, you should always ask yourself, "Do I really want to perform an elementwise comparison with type promotion?", and if the answer is not a resounding `T`, consider using `identical()` instead.

It is also worth noting that the condition for `if` does not need to be a `logical` value; a value of a different type will be converted to `logical` by coercion if possible (see section 2.3.2).

2.5.2 The `if–else` statement

Often you want to perform an alternative action when the condition of an `if` statement is `F`; the `if–else` statement allows this. It is simplest to just show this with an example:

```
> if (2/2/2/2/2 > 10000) "division is da bomb!"; else "not so much."
"not so much."
```

Super simple, right?

2.5.3 A digression: the semicolon, ;

You might have noticed a subtle twist in the last example: the semicolon `;` at the end of the first part of the statement. A bit of subterfuge has gotten swept under the rug thus far, but it's time to come clean. Every statement in Eidos must end with a semicolon (except compound statements; see section 2.5.4). However, when you're working interactively in EidosScribe, EidosScribe will add a trailing semicolon to your statements if necessary, just to make your life simpler. So when you type:

```
> 1+1==2
```

what is really being evaluated behind the scenes is:

```
> 1+1==2;
```

When you're not working interactively, semicolons are required, and if you forget, you will get an error, like this:

```
> 1+1==2
ERROR (Parse): unexpected token 'EOF' in statement; expected ';'
```

EOF stands for End Of File; it's a standard way of referring to the end of an input buffer, in this case the line of input provided by the user for execution.

So now the reason for that semicolon's existence, in the example back in section 2.5.2, is obvious: it is required at the end of the `if` statement, as usual, but EidosScribe is not smart enough to add it for us automatically in this case, because the `if` statement is followed by an `else` clause.

The simplest and shortest possible statement in Eidos is the "null statement", which consists of nothing but a semicolon:

```
;
```

This is not terribly useful, since it does nothing.

2.5.4 Compound statements with { }

The other thing you might wonder about, regarding `if` statements, is: what if I want to perform more than one action in response to the condition being `T` or `F`? This, then, is an opportune moment to introduce the concept of compound statements. A compound statement is a series of statements (zero or more) enclosed by braces. An example is worth a thousand words:

```
> if (1+1==2)
{
   x = 1;
   x = x + 1;
   print(x);
}
else
{
   print("whoah, I'm confused");
}
2
```

Note that the input here is spread across multiple lines for clarity; all of this could be typed on a single line instead. If entered as multiple lines, it cannot be entered in EidosScribe's interactive

mode because the `if` statement would stand on its own and be evaluated as soon as it was completed; instead, the full text would need to be entered in the script area on the left, selected, and executed (see section 5 for more discussion of using EidosScribe). All of the blue lines are user input, whereas the final line in black, `2`, shows the output of the execution of the whole `if–else` statement; the `if` clause is executed, the calculations involving `x` are performed, and the final statement `x;` produces a result which is printed to the console as usual.

We are using the `print()` function here to print either the value of `x` or the string `"whoah, I'm confused"`. We haven't needed to use `print()` before, because when a single statement is executed in the Eidos interpreter the value of that statement is automatically printed. Compound statements, however, do not evaluate to a value (they evaluate to `void`, to be precise; see section 2.5.4), and so we need to explicitly call `print()` here. (Note that prior to version 2 of Eidos, compound statements evaluated to the value of the last statement they executed, as in R; that policy was changed in Eidos version 2 because it was found to be problematic in certain cases, due to the stronger type-checking in Eidos compared to R.) This is the first time we have seen the `print()` function; its meaning here is rather obvious, but it is covered in more detail in section 3.5.

You can use a compound statement in any context in which a single statement would be allowed. For example, compound statements are very commonly used with the looping constructs discussed in the next section.

2.6 Loops

Loops are used to repeat a statement (or a compound statement) more than once. Depending upon the task at hand, the best tool for the job might be a `while` loop, a `do–while` loop, or a `for` loop, as described in the following sections.

2.6.1 The `while` statement

A `while` loop repeats a statement as long as a given condition is true. The condition is tested before the first time that the statement is executed, so the statement will be executed zero or more times. Here is a code snippet to compute the first twenty numbers of the Fibonacci sequence:

```
> fib = c(1, 1);
while (size(fib) < 20)
{
   next_fib = fib[size(fib) - 1] + fib[size(fib) - 2];
   fib = c(fib, next_fib);
}
fib;
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

This snippet brings together many of the things we've seen in past sections, such as constructing vectors with the concatenation function `c()` (section 2.2.3), defining variables with the assignment operator `=` (section 2.4.1), getting the `size()` of a vector (section 2.4.2), and accessing values within a vector using the subset operator `[]` (section 2.2.4). Its use of a `while` loop is optimal, because it ensures that if the `fib` vector is already long enough to satisfy the length condition `size(fib) < 20`, no further values of `fib` will be computed. You could use this `while` loop to lengthen the `fib` vector on demand within a larger block of code that used the `fib` vector repeatedly.

2.6.2 The `do–while` statement

A `do–while` loop also repeats a statement as long as a given condition is true. However, in this case the condition is tested at the end of the loop, and thus the loop statement is always executed at least once. Here is a code snippet to compute a factorial:

```
> counter = 5;
factorial = 1;
do
{
   factorial = factorial * counter;
   counter = counter - 1;
}
while (counter > 0);
"The factorial of 5 is " + factorial;
"The factorial of 5 is 120"
```

This example brings in `string` concatenation using the `+` operator (section 2.3.4) in order to generate its output line. Note that this example could be rewritten using a `while` loop instead, but it might be a bit less intuitive in its operation since it would no longer embody the formal definition of the factorial as explicitly. Note also that computing a factorial could be done much more trivially (and efficiently) using the sequence operator `:` and the `product()` function (see section 3.1), but the code here is useful for the purpose of illustration.

2.6.3 The `for` statement

The third type of loop, the `for` loop, is used to loop through all of the elements in a vector. For each value in the given vector, a given variable is set to the value, and a given statement is then executed. For example, the following code computes squares by setting `element` to each value of `my_sequence`, one by one, and then executing the `print()` function for each value:

```
> my_sequence = 1:4;
for (element in my_sequence)
   print("The square of " + element + " is " + element^2);
"The square of 1 is 1"
"The square of 2 is 4"
"The square of 3 is 9"
"The square of 4 is 16"
```

Notice how operator `+` is used to construct the output.

This looping construct is called by various names in other languages, such as the “for each” statement (PHP), the “range-based for” (C++), “fast enumeration” (Objective-C), and so forth. It is different from the traditional `for` loop of C and related languages, which entails an initializer expression, a condition expression, and an increment/decrement expression. That type of `for` loop does not exist in Eidos (following R); the iterator `for` of R and Eidos is a more natural and efficient choice for vector-based languages.

2.6.4 The `next` statement

Sometimes you might wish to cut short the execution of a given iteration of a loop, skipping the rest of the work that would normally be done and proceeding directly to the next iteration. This is the function of the `next` statement. For illustration, here is a trivial modification of the previous example, changed to print only squares that are divisible by `5` – and to print their cubes as well:

```
> for (element in 1:20)
{
   square = element ^ 2;
   if (square % 5 != 0)
      next;
   cube = element ^ 3;
   print(element + " squared is " + square + ", cubed is " + cube);
}
"5 squared is 25, cubed is 125"
"10 squared is 100, cubed is 1000"
```

```
"15 squared is 225, cubed is 3375"
"20 squared is 400, cubed is 8000"
```

Notice how the vector used by the `for` loop is specified directly, using the sequence `1:20`; this is a very common pattern. This example also uses the modulo operator `%` (section 2.3.1) to determine divisibility.

The main point, however, is that the `next` statement causes the loop to skip most of its work whenever `square` is not divisible by `5`. The `next` statement can be used within `while` and `do–while` loops as well, and does exactly the same thing in those contexts.

2.6.5 The `break` statement

The final topic to cover regarding loops is the `break` statement. Often it is necessary to stop the execution of a loop altogether, not just to cut short the current iteration of the loop as `next` does. To achieve this – to break out of a loop completely – use the `break` statement. For example, here is a very primitive way of finding prime numbers in Eidos:

```
> for (number in 2:20)
{
   prime = T;
   for (divisor in 2:number^0.5)
      if (number % divisor == 0)
      {
         prime = F;
         break;
      }
   if (prime)
      print(number + " is prime!");
}
"3 is prime!"
"5 is prime!"
"7 is prime!"
"11 is prime!"
"13 is prime!"
"17 is prime!"
"19 is prime!"
```

There are a few things to note here. First of all, this example is more structurally complex than previous examples; we have an `if` statement with a compound statement, nested within a `for` statement, which is itself nested within the compound statement of another `for` statement. If it is not obvious what is going on here, take a little time to ponder it. Second, notice how the `^` operator (section 2.3.1) is used to calculate a square root; the `:` and `%` operators also play major roles, but we have already revisited them in previous loop examples. A `logical` "flag" variable, `prime`, is used to keep track of whether we have found a divisor; such flag variables are a very common and useful programming paradigm.

Most importantly, however, note how the `break` statement stops the execution of the inner `for` loop as soon as a divisor is found. This makes the loop *much* more efficient; on my machine, computing the primes up to 1,000,000 takes about 11.6 seconds with the `break` statement, versus 90.6 seconds without it – quite a difference! Skipping unnecessary work is good. Not that this code is a paragon of speed anyway; there are, of course, much faster ways to search for primes.

What's that you say? What's a faster way to search for primes? OK, since it also illustrates the use of `break`, but this time in a `do–while` loop, here's a faster way to search for primes:

```
// the Sieve of Eratosthenes in Eidos!
last = 1000000;
x = 2:last;
lim = last^0.5;
```

```
do {
   v = x[0];
   if (v > lim)
      break;
   print(v + " is prime!");
   x = x[x % v != 0];
} while (T);
for (v in x)
   print(v + " is prime!");
```

This Eidos algorithm takes about 0.47 seconds on my machine. It's based on a famous algorithm called the Sieve of Eratosthenes that was discovered more than 20 centuries ago. The first line of code is a comment that says precisely that, in fact. Comments in Eidos start with `//` and consume the remainder of the line; they do nothing except annotate your code for readability.

There are a few things to note here. One is that the `do–while` loop has a condition of `T`; it will thus loop forever unless it is terminated by a `break` statement. This is a common paradigm called an *infinite loop*, used when you want to terminate a loop based on a condition test in the middle of a loop, not at the beginning (as a `while` loop would do) or at the end (as a `do–while` loop would do). Inside the loop, the most recently found prime, `v`, is compared against a threshold value, `lim`; when that threshold is reached, all of the values that still remain in the vector `x` are guaranteed to be prime, so the loop is exited via `break`, and everything left in `x` is printed.

Another thing to note is that while Eidos is slow compared to compiled languages, it is still fairly quick. Timing tests indicate that it is often substantially faster than R, and often not markedly slower than Lua (which is nothing to sneeze at). As seen above, it can find every prime up to 1,000,000, and print them out as well, in much less than a second! It can do this, however, by virtue of using a good algorithm; using a less efficient algorithm to perform the same task took 11.6 seconds or even 90.6 seconds, as seen above. If your Eidos code is performing poorly, the first step should be to look at the big picture and think about whether there is a more efficient algorithm you could use to solve your problem.

2.6.6 The `return` statement

While we are on the subject of control flow, we can cover one more statement type: the `return` statement. This doesn't belong under the topic of loops, exactly, but we will nevertheless treat it here. The `return` statement returns a value from a block of code, as in other languages such as C and R. In one common case, when defining a user-defined function (see chapter 4), `return` is used to stop execution of the function and return a given value to the caller; that will be explored further in chapter 4 when we look at user-defined functions. Otherwise, a `return` is useful mostly when the Context within which you're using Eidos uses the returned value. When using Eidos in SLiM, for example, SLiM uses the value returned by Eidos scripts such as `fitness()` callbacks and `mateChoice()` callbacks, making `return` very useful in that Context (see SLiM's manual for details). Apart from such Context-dependent uses, `return` is mainly useful as a way to break out of nested loops regardless of the depth of nesting, as illustrated below.

The `return` statement is very simple: the keyword `return`, and then, optionally, an expression. When the `return` statement is executed, the expression is evaluated and its value is immediately returned as the value of the largest enclosing statement. The `return` statement therefore breaks out of all conditionals, loops, and compound statements, regardless of the depth of nesting. For example, one could write a (very dumb) search for factors of a number like this:

```
for (i in 1:10)
   for (j in 1:10)
      if (i*j == 21)  // the magic numbers?
         return 21 + " is " + i + " * " + j;
```

As soon as this code finds a combination of `i` and `j` that produce `21` when multiplied (as noted by the comment on the `if` statement), the `return` statement breaks out of both loops and returns a `string` describing the hit. A `break` statement, on the other hand, would only break out of the innermost loop; the outer loop, over `i`, would continue to execute. (A `break` statement also can't be given a value to return, so it would be much less convenient to use here for that reason as well.)

If the expression for the `return` statement is omitted, then for all practical purposes no value is returned. (Technically, the return is a special pseudo-value called `void`, representing the absence of any return value; `void` will be discussed in more detail in section 2.7.3.)

2.7 Functions

2.7.1 Calling functions: operator `()`

Functions are so useful that we have already used them in several examples. It is now time to discuss them more formally. A function is simply a block of code which has been given a name. Using that name, you can then cause the execution of that block of code whenever you wish. That is the first major purpose of functions: the *reuseability* of a useful chunk of code. A function can be supplied with the particular variables upon which it should act, called the function's "parameters" or "arguments"; you can execute a function with the sequence `5:15` as an argument in one place, and with the string `"foo"` as an argument in another. That is the second major purpose of functions: the *generalization* of a useful chunk of code to easily act on different inputs.

In Eidos, you may define your own functions (see chapter 4), or you may execute a *lambda* (i.e., a snippet of code represented as a `string` value) directly in the Eidos interpreter (see `executeLambda()`, section 3.9). However, a fairly large set of built-in functions are supplied for your use, and the hope is that they will suffice for most purposes. The built-in functions are covered in detail in section 3. For now, there are just a few generalities that should be mentioned.

First of all, functions are called using the `()` operator; in section 2.4.1, for example, we called the function named "ls" simply by writing `ls()`. The `ls()` function is an example of a function that can be called with no arguments; it will then print all of the constants and variables defined in the global namespace, as seen in section 2.4.1.

Second, function arguments go between the parentheses of the `()` operator, separated by commas. In section 2.4.2, for example, we used the `size()` function to get the size – the number of elements – of a vector named `x`, by passing `x` as an argument to `size()`. That was written as `size(x)`. The `size()` function requires exactly one argument, coincidentally also named `x`; writing `size()` without an argument will thus cause an error:

```
> size()
ERROR (EidosInterpreter::_ProcessArgumentList): missing required
argument x.
```

Third, some functions can take a variable number of arguments. For example, in section 2.2.3 we saw the `c()` function, which will take any number of arguments and stick them all together to produce a single vector as its result. Most functions expect an exact number of arguments; many functions, in fact, are even fussier than that, requiring each parameter to be of a particular type, a particular size, or both. But some, such as `c()`, are more flexible. In section 3 the argument requirements for every built-in function will be specified.

Fourth, many functions provide a return value. In other words, a function call like `c(5,6)` can evaluate to a particular value, just as an expression like `5+6` evaluates to a particular value. The result from a function call can be used in an expression or assigned to a variable, as you might expect. Here is an example:

```
> x = c(1, 3, 5, 7, 9) + 0.5
> x
1.5 3.5 5.5 7.5 9.5
> size(x)
5
```

The `c()` function produces a vector; that vector is then added to `0.5` with the `+` operator, and the result of that is assigned to `x` with the `=` operator. The number of elements in `x` is then counted with the `size()` function, which again produces a return value: `5`. The return type for all of the built-in functions will be specified in section 3.

2.7.2 The `NULL` type

In section 2.4.1, you saw a bit of foreshadowing regarding another variable type in Eidos: `NULL`. The time has come to delve into that topic a little more, because `NULL` is important in using functions. This section will be fairly esoteric, however, and beginning Eidos users may wish to skip it until later. That said… `NULL` has two uses, in fact: as a return value, and as a function argument value.

As a return value, `NULL` is often used to indicate that a function happened to have nothing useful to return, or perhaps that a non-fatal error was encountered that prevented a value from being returned; in this context, `NULL` is sort of a shrug of the shoulders. Often such a `NULL` return is a result of passing `NULL` in as an argument; garbage in, garbage out, as they say. A trivial example:

```
> max(NULL)
NULL
```

The `max()` function returns the maximum element within the vector it is passed, but given `NULL`, it simply returns `NULL`. More interestingly, the `readFile()` function will return `NULL` if an error occurs that prevents the file-read operation from completing. The calling code could then detect that `NULL` return and act accordingly – it might try to read from a different path, print an error message, or terminate execution with `stop()`, or it might just ignore the problem, if reading the file was optional anyway (such as an optional configuration file to modify the default behavior of a script).

The other use of `NULL`, as mentioned above, is as an argument to a function. Passing `NULL` is occasionally a way of signaling that you don't want to supply a value for an argument, or that you want a default behavior from the function rather than telling it more specifically what to do. None of the functions you have seen thus far use `NULL` in this way, but you will see examples of it in section 2.7.5 and chapter 3.

`NULL` cannot be an element of a vector of some other type; it cannot be used to mark missing or unknown values, for example. Instead, `NULL` is its own type of vector in Eidos, always of zero length. (There is also no `NA` value in Eidos like the one in R, while we're on the topic of marking missing values. Not having to worry about missing values makes Eidos substantially simpler and faster, and Eidos – unlike R – is not designed to be used for doing statistical analysis, so marking missing values is not expected to be important. Eidos does support `NAN` – Not A Number – values in `float` vectors, however, which could conceivably be used to mark missing values if necessary.)

The basic philosophy of how Eidos handles `NULL` values in expressions and computations is that `NULL` in such situations represents a non-fatal error or an unknown value. If using the `NULL` value in some meaningful way could lead to potentially misleading or incorrect results, Eidos will generate a fatal error. The idea is to give Eidos code an opportunity to detect a `NULL`, and thus to catch and handle the non-fatal error; but if the code does not handle the `NULL`, using the `NULL` in further operations should result in a fatal error before the functioning of the code is seriously compromised. `NULL` values are thus a sort of third rail; there's a good reason they exist, but you

have to be very careful around them. They are a bit like zero-valued pointers in C (`NULL`), C++ (`nullptr`), Objective-C (`nil`), and similar languages; they are widely used, but if you ever use one the wrong way it is an immediate and fatal error.

Documenting exactly how Eidos handles `NULL` in every conceivable situation would be difficult. If you need to know what happens in a particular case, you should try it out in EidosScribe; as Socrates liked to say, "there's nothing like asking". A few broad guidelines can be stated, though (making reference to some topics that have not yet been covered, so don't worry if some terms here are unfamiliar):

- Functions and methods may return `NULL` to indicate a non-fatal exceptional condition even if their call signature does not state `NULL` as a potential return type; call signatures indicate only the return type under normal, non-exceptional circumstances. If an exceptional return of `NULL` is a possibility for a given function or method, that should be mentioned in its documentation. Functions and methods may also be explicitly declared as possibly returning `NULL`; this indicates that in those cases `NULL` is a standard, normal return value rather than an indication of an exceptional condition (again, the documentation for the function or method should provide further detail).

- Method calls on object vectors that contain multiple elements might result in `NULL` being returned by some elements and non-`NULL` values being returned by other elements. In this case, the `NULL` returns will be silently dropped from the aggregated return value from the method call. This is probably the most dangerous aspect of the way Eidos handles `NULL`. If the method returns one value per element, you can check for this condition by comparing the `size()` of the returned value to the `size()` of the object upon which the method was called. If you need to detect and respond to `NULL` returns from each element in an object, you should iterate over the elements of the object with a `for` loop and call the method on each element individually. Because of this, it is probably wise for Context designers to design methods so that they return `NULL` only in cases where dropping the `NULL` values is likely to be harmless.

- `NULL` may not be passed as a parameter value to a function or method unless that is specifically allowed by the call signature of the function/method. Eidos designates this as a fatal error in order to limit the unpredictable consequences of propagating `NULL` values forward.

- `NULL` may never be the value of a property of an object. Assigning `NULL` into a property is always illegal, and a property will never give `NULL` as its value, even to indicate an exceptional condition. Since properties are intended to represent lightweight attributes, they should not be generating exceptional conditions anyway.

- `NULL` may not be used with any of the arithmetic operators (`+`, `-`, `*`, `/`, `%`, `^`), nor with any of the logical operators (`&`, `|`, `!`), nor with the range operator (`:`), nor may it be used as the condition in an `if`, `while`, or `do` loop, nor may it be used as the value over which a `for` loop iterates. In all these cases, using `NULL` is a fatal error. Beginning in Eidos 2.2, it may, however, be used with the string concatenation operator (`+`), where it is treated exactly as if it were a singleton `string` vector containing the string element `"NULL"` since that behavior is useful for output.

- `NULL` may not be used with the comparison operators (`==`, `!=`, `<`, `>`, `<=`, `>=`). To test whether a given value is `NULL`, therefore, *you cannot use the equality operator,* `==`; the comparison `x == NULL` will always cause a fatal error. Instead, you must use the `isNULL()` function.

- It is legal to subscript `NULL`; `NULL[...]` always produces `NULL`. `NULL` may not be used as a subscript, however; `x[NULL]` is always an error (except `NULL[NULL]`, which produces `NULL`).
- You may assign `NULL` into a variable (`x = NULL`) and you may return `NULL` from your code (`return NULL`). Passing `NULL` values around in your own code is harmless; it is the attempt to actually use a `NULL` value that is often fatal.
- `NULL` can be passed to `print()` (it prints as "`NULL`") and to `cat()` (it emits nothing).
- `NULL` may be assembled into a vector with other values using `c()`, but the `NULL` values will be dropped from the returned vector. For example, the result of `c(7, NULL, 8)` is a vector containing only `7` and `8`; the `NULL` is silently dropped. This is the same behavior described above for assembling the results of vectorized method calls into a single result vector.
- `NULL` cannot be turned into a value of another type by automatic type promotion. If you attempt to explicitly coerce `NULL` to another type using the `as...()` family of functions, it is dropped since `NULL` is zero-length, resulting in a zero-length vector. For example, the result of `asInteger(NULL)` is `integer(0)`, Eidos's way of printing a zero-length vector of type `integer`. Beginning in Eidos 2.2, an exception to this rule is `asString(NULL)`, which returns the string `"NULL"` since that behavior is useful for output.

These policies are somewhat stricter than the policies regarding `NULL` in R. This reflects several factors. First, R generally takes quite a permissive attitude toward runtime errors, and is famous (some would say infamous) for bending over backwards to find a possible interpretation for questionable code rather than generating an error. Eidos is much stricter, in order to minimize the risk of code invisibly producing an incorrect result; maximal robustness, rather than maximal flexibility, is the primary design goal of Eidos. Second, Eidos does not have the `NA` value of R, representing a missing value, so many cases in which R would produce an `NA` have to produce a `NULL` in Eidos. This means that the policies in Eidos regarding `NULL` have to be correspondingly stricter, because `NULL` is more common than in R, and might represent cases that the user would expect, from experience with R, to be handled in an intelligent manner as a missing value; the risk of undetected failure is therefore high, and a strict policy is needed to minimize that risk.

2.7.3 The `void` type

In sections 2.5.4 and 2.6.6, we saw hints of another obscure Eidos type: `void`. Continuing on in the vein of the previous section's discussion of `NULL`, let's now get all our cards on the table by discussing `void` as well. This section, like the previous one, will be quite esoteric, and beginning Eidos users may wish to skip it.

The `void` type is much simpler than `NULL`; arguably it isn't really even a type, in fact, since there isn't really a "value" to have a type in the first place. Instead, `void` is used as a way of declaring that no value exists. We will see that many functions return `void`, which indicates that they do not return a value at all. Many language constructs also evaluate to `void`, such as compound statements, while and do-while and for loops, and so forth; all that this really means is that these constructs do not generate a value. (An exception is `if` statements, which *can* evaluate to a non-`void` value; we actually saw this in section 2.5.1, where we did not need to call `print()` for the result of the `if` statement to be printed in the console.)

For example, the `print()` function returns `void`. If we call it, it prints what it is told to print, but nothing else is emitted to the console besides that, because it provides no return value:

```
> print("foo")
"foo"
```

Here `"foo"` is *not* the return value from `print()`; it is merely output that `print()` sent to the console. There is no return value from `print()` for the Eidos interpreter to print; its return is `void`.

Indeed, if we attempt to assign the result of `print()` into a variable, we get an error:

```
> x = print("foo")
"foo"
ERROR (EidosInterpreter::_AssignRValueToLValue): void may never be
assigned.
```

A variable may never have a value of `void`, because `void` is not really a value; so trying to assign `void` into `x` results in an error. Similarly, `void` cannot be passed as an argument to any function, and cannot be manipulated by any operator. You can't even test whether a returned value is `void`, in fact; that is also an error. This makes `void` sound more dangerous than it is, though. In fact, the rule for dealing with it is quite simple: if a function or method is declared as returning `void`, just don't attempt to use that return value for any purpose whatsoever. If you follow that rule, you will be fine.

You may be wondering what the point of all this really is. The point is type-safety. If a function is declared as having a return type of `void`, it is an error for it to return anything other than `void`; conversely, if a function is declared as having some non-`void` return type, it is an error for it to return `void`. This allows Eidos to easily catch certain common errors, such as forgetting to include a `return` statement where one was needed. It also catches errors that result from a belief that a function returns a value when it actually doesn't; the attempt to use the `void` return will trigger an error. In general, it allows Eidos to catch bugs at runtime – a good thing.

2.7.4 The `functionSignature()` function, and introducing function signatures

Section 3 will cover every built-in function in detail. Before we get to that, however, it is worth exploring Eidos's support for a language feature called *introspection*: it has ways for you to examine the internals of the language itself. Of particular note here is a function called `functionSignature()`. Calling this function results in quite a bit of output, beginning like this (with a vertical ellipsis to indicate that only some of the output is shown here):

```
> functionSignature()
(numeric)abs(numeric x)
(float)acos(numeric x)
(logical$)all(logical x)
(logical$)any(logical x)
    ⋮
```

Each line shows the *function signature* of a different built-in (or user-defined) function. The first line, for example, shows the signature of `abs()`, the absolute value function:

```
(numeric)abs(numeric x)
```

This signature indicates that `abs()` takes a `numeric` argument (meaning either an `integer` or a `float`), and it also returns a `numeric` value. In fact, if you pass it an `integer` it will return an `integer` back, and if you pass it a `float` it will return a `float` back; but the function signature is not specific enough to show that.

The next function, `acos()`, takes a `numeric` value but always returns a `float`:

```
(float)acos(numeric x)
```

This is the arc cosine function. The signature for `acos()` is the function signature of most of the math functions in Eidos, in fact. These math functions will take an `integer` as an argument for

convenience, but their return value is `float` because the mathematical operation they perform can produce a `float` even from `integer` input; `acos(0)` is approximately `1.5708`, for example.

The next signature is for the `all()` function, which was discussed a little in section 2.5.1:

```
(logical$)all(logical x)
```

This signature indicates that `all()` takes a logical vector and returns a logical vector, but the `$` symbol in the `logical$` return value guarantees that `all()`'s return value will be a *singleton* – a vector containing exactly one value. This makes sense, since `all()` determines whether or not *all* of the values in the vector passed in are T; if *all* of them are, it returns `T`, otherwise it returns `F`. It is thus one way of summarizing a multivalued vector down to a singleton – as is the next function, `any()`.

Let's look at a few other interesting signatures to see some of the other notation conventions that they use. The signature for the function `c()` looks like:

```
(*)c(...)
```

The ellipsis (`...`) indicates that `c()` takes any number of arguments, while the asterisk (`*`) indicates that `c()` can return any type. In fact, the type that `c()` returns will depend on what types of arguments you pass to it, because it concatenates its arguments together with type promotion, as we saw in section 2.2.3.

Next let's look at the signature for the `ls()` function, which we saw in section 2.4.1:

```
(void)ls(void)
```

The notation `void` here indicates the lack of a value, as discussed in detail in section 2.7.3; in other words, `ls()` takes no parameters, and it returns no value.

The signature for the `asFloat()` function has a symbol we haven't seen yet:

```
(float)asFloat(+ x)
```

The `+` symbol here, shown as the type for the parameter named `x`, indicates that `x` may be any type of vector except an `object` vector (we will cover the `object` type in section 2.8). So the `*` symbol that we saw as the return type of `c()` indicates *any* type; the `+` symbol indicates any type *except* `object`. The `asFloat()` function converts the parameter it is passed to be of type `float`; but while `logical`, `integer`, and `string` vectors (and even `NULL`) can all be converted to `float`, `object` vectors cannot.

Finally, let's look at the signature for `sum()`:

```
(numeric$)sum(lif x)
```

The parameter `x` here has the curious type designation `lif`. This represents the fact that `x` may be any of three types: `logical` (`l`), `integer` (`i`), or `float` (`f`). Similarly, type `NULL` can be abbreviated as `N`, string as `S`, and object as `o` (and void as `v`, in fact, but that is almost never seen). The `numeric` type designation seen previously is just a convenient euphemism for `if` (which would otherwise be a bit confusing since it is also the word "if"), whereas `+` represents `Nlifs` and `*` represents `Nlifso`.

If you look through the output of `functionSignature()`, you will see some function signatures using other notation that we have not yet discussed. That will be the topic of the next section.

2.7.5 Optional arguments and default values

In the output from `functionSignature()` that we started examining in the previous section, look and the function signature for `functionSignature()` itself:

```
(void)functionSignature([Ns$ functionName = NULL])
```

First of all, `functionSignature()` takes one parameter, named `functionName`, of type `Ns$`, which the previous section explained means that `functionName` must either be `NULL` or a singleton `string` vector. However, we called it before without any parameter, simply by calling `functionSignature()`. Doesn't that contradict `functionSignature()`'s signature? The answer is no, and the reason is that the `functionName` parameter is enclosed in brackets, `[]` (not to be confused with the use of brackets as the subset operator, as discussed in section 2.2.4). These brackets indicate that `functionName` is an *optional argument*; it may be omitted by the caller. If it is omitted, furthermore, the value after the `=` indicates the default value for the argument – `NULL`, in this case. So calling `functionSignature()` is precisely identical in meaning to calling `functionSignature(NULL)`; one is just a shorthand for the other.

So what happens if we override `functionSignature()`'s default value, and pass it a `string` for `functionName`? Calling `functionSignature("runif")`, for example, gives us this:

```
(float)runif(integer$ n, [numeric min = 0], [numeric max = 1])
```

So we can print the signature for a single function of our choice, rather than the full list that `functionSignature()` prints by default. The `runif()` function shown here draws `n` values from a uniform distribution (where `n` is a singleton `integer`) and returns them as a `float` vector; that's the first half of the signature. The second half indicates that `runif()` can also take two optional arguments, `min` and `max`, which are both `numeric`; `min` has a default value of `0`, `max` has a default value of `1`. So `runif(5)` will return `5` draws from the uniform distribution spanning `0` to `1`, using the default values of `min` and `max`. But `runif(5, -1)` will draw from the distribution spanning `-1` to `1` (using the default value for `max`, but specifying a value for `min`), and `runif(5, 10.5, 100)` will draw from the distribution spanning `10.5` to `100`, overriding both default values.

2.7.6 Function signatures summarized

The previous two sections introduced function signatures to illustrate how Eidos represents functions, and to explain the various constraints and guarantees they embody when you use them. In summary, a function signature states the arguments and the return type of a function, following a template like:

```
(return_type)function_name(argument_list)
```

The return type and the argument list use various symbols:

`void`	no return value, or (for the argument list) no arguments
`integer`	an `integer` (see section 2.1.1)
`float`	a `float` (see section 2.1.2)
`logical`	a `logical` (see section 2.1.3)
`string`	a `string` (see section 2.1.4)
`object`	an `object` (see section 2.8), optionally with a class specifier (see section 4.3)
`numeric`	either an `integer` or `float`
`+`	any type except `object` (but not `void`)
`*`	any type, including `object` (but not `void`)
`$`	follows a type to indicate a *singleton* – a vector containing exactly one value
`[]`	encloses an *optional argument* – an argument which may be omitted
`=`	precedes a *default value* for an optional argument
`...`	indicates that any number of arguments may be supplied, from zero on up

Occasionally, a function will allow an argument or a return value to be of more than one type. For example, in SLiM it is common for functions to take an argument that is either an `integer` (identifying a SLiM object by ID) or an `object` (passing the object directly). In such cases, as explained in section 2.7.4, Eidos represents the type using single letters to designate each supported type:

`v`	`void`
`N`	`NULL` type
`l`	`logical` type
`i`	`integer` type
`f`	`float` type
`s`	`string` type
`o`	`object` type, optionally followed by a class specifier (see section 4.3)

If you write your own user-defined functions, you will write a function signature to declare the function (see chapter 4); otherwise, it's just useful to understand them so that you can use `functionSignature()` to remind yourself about the built-in functions you're using. When you put the insertion point inside a function call in EidosScribe or SLiMgui (SLiM's graphical modeling environment), the function signature will be shown in the status bar at the bottom of the window, which is a useful way to quickly remind yourself about the function's call syntax. Signatures will also come up in the context of *methods* – discussed in section 2.8, in which the `object` type is introduced.

2.7.7 Named arguments in function calls

We have mentioned the names of function arguments in passing; for example, the names of the arguments to `runif()` are `n`, `min`, and `max`, as seen in `runif()`'s signature:

```
(float)runif(integer$ n, [numeric min = 0], [numeric max = 1])
```

These argument names can actually be used when calling functions. For example, calling `runif(5, 3, 7)` is rather cryptic; unless you remember the arguments that `runif()` takes, you might need to go back and look at `runif()`'s signature, or at its documentation, to remember what meaning it will ascribe to the `5`, the `3`, and the `7`. But instead, if you wish, you can call:

```
runif(n=5, min=3, max=7);
```

The meaning of that call is much clearer. That is therefore the first use of *named arguments*: to make the meaning of a call more explicit by labeling its arguments.

A second use of named arguments is to skip over intervening optional arguments that you don't want to bother specifying. Suppose, for example, that you want to get five draws from a uniform distribution spanning `0` to `10`. You could call `runif(5, 0, 10)`, but isn't it tedious having to specify the min of `0` even though that is `min`'s default already? Well, perhaps not that tedious; but there are calls with more parameters than that, where specifying a value for the very last parameter would require you to give values for all of the intervening optional parameters as well, and at a certain point that really does get tedious. Instead, you can simply call `runif(5, max=10)`. The first parameter, `5`, is matched up with the first argument in the signature, `n`, as usual. But then the second parameter is named, `max=10`, and that tells Eidos to scan forward looking for `max`, using default values for everything in between.

You can use names for both required and optional arguments. However, once you use a named argument in a call, all the rest of the arguments for that call must also be named; you cannot call `runif(n=5, 3, 7)`, for example. Required arguments cannot be skipped over; you cannot call `runif(max=10)` since the argument `n` is not optional (and thus has no default value). And finally,

you must give arguments in the same order that they are shown in the signature; you cannot call `runif(min=3, max=7, n=5)` since the argument `n` is out of order.

The use of named arguments slows down the function dispatch code inside Eidos just a little bit, so using named arguments is not recommended inside very tight loops calling out to very fast functions, where the function dispatch overhead itself is a major part of the total runtime of the code. This difference is small, however, and will be noticed only in the most extreme scenarios.

2.8 Objects

2.8.1 The `object` type

In addition to `logical`, `integer`, `float`, `string`, and `NULL`, there is one more type in Eidos left to discuss: `object`. An `object` is a vector that contains elements; it is a container, a bag of stuff. In this way, it is similar to Eidos's other types; a `float` in Eidos is a vector containing floating-point elements, whereas an `object` is a vector containing `object`-elements (often just called "elements" in general). An `object` can also embody *behavior*: it has operations that it can perform using the elements it contains. The `object` type in Eidos is thus similar to objects in other languages such as Java, C++, or R – except much more limited. In Eidos you cannot define your own classes of `object` type; you work only with the predefined `object` types supplied by SLiM or whatever other Context you might be using Eidos within. These predefined `object` types generally contain Context-dependent elements related to the task performed by the Context; in SLiM, the elements are things such as mutations, genomic elements, and mutation types (described in SLiM's documentation).

The behaviors of objects in Eidos manifest in two ways: objects can have *properties* (also called instance variables or member variables in some languages) that can be read from and written to, and they can have *methods* (also sometimes called member functions). The properties of an `object` in Eidos are determined by the type of element the `object` contains; an Eidos `object` will always contain only one type of element (just as a `float` cannot contain `string`-elements, for example).

Instances of particular `object` classes – particular kinds of objects – are obtained via built-in functions and/or global constants and variables. For example, in SLiM there is a global constant called `sim` that represents the current simulation as an instance of the `SLiMSim` class. In the interests of keeping free of the assumption that Eidos is being used with SLiM, we will discuss objects here using a hypothetical example: a *house* `object` type. This idea will be fleshed out in the next section, on element access.

2.8.2 Element access: operator `[]` and sharing semantics

If an `object` is a vector of elements, how do you access those elements from Eidos? The answer is obvious, really: using the `[]` operator, just as you can access the `float`-elements in a `float` vector, or the `string`-elements in a `string` vector (section 2.2.4). Remember that `[]` used on a `float` vector does *not* give you a raw floating-point value; the individual elements inside Eidos types are never directly accessible. Instead, everything in Eidos is a vector (section 2.2.1), and even single values are always wrapped inside a vector. The same is true for `object` vectors and their underlying elements; you cannot get an `object`-element on its own.

So this subsection is not really saying anything new; it is just saying that the `object` type works very much like the other built-in types. Don't get confused by that. As an example, let's develop the idea introduced above (section 2.8.1) of a house `object` type. The `object` is a vector that contains elements; each element would be a house, in this example, so you could think of the whole vector as representing a neighborhood (or any other list of houses). If you had a house `object` variable called `neigh` (for "neighborhood"), it might contain ten houses comprising some

neighborhood; `myHome = neigh[0]` would assign a new house `object`, containing just the first house from `neigh`, into a new variable named `myHome`.

Those of you with programming experience might have just asked yourselves the question: what exactly is meant by the claim that `myHome` contains "just the first house from `neigh`"? There are two possibilities. One is that an element is only ever contained by one particular `object`, so when that element is put into a new object, it gets copied. This is called "copy semantics" – and it is *not* how Eidos works. The other possibility is that an element can be contained by more than one `object`, and never gets copied unless a copy is explicitly requested somehow. This is called "sharing semantics" – and it is how objects in Eidos work. Skip this comment if it confuses you, but: if you are familiar with the concept of "pointers" from languages such as C, you could think of Eidos `object` variables as using pointers to refer to the `object`-elements they contain – and that is, in fact, how Eidos is implemented internally.

It might be worth mentioning the reason for this policy choice, since it is somewhat unusual (at least without the explicit use of pointers, as in C and Objective-C, or references, as in C++). The reason is that Eidos is not about creating new objects, new classes, etc.; it is about manipulating the existing objects defined by the Context within which Eidos is being used. In SLiM, for example, the SLiM simulation defines the genomes, mutations, genomic element types, etc., that Eidos manipulates. If Eidos made copies of those elements as a side effect of operations like subscripting and assignment into variables, the newly created elements would not be recognized by the simulation, and changing or manipulating them would make no difference to the simulation – a bizarre and pointless outcome. Instead, it makes sense for Eidos to always act on the objects that presently exist within its Context; thus, always following sharing semantics is the logical policy. When you wish to create a new `object`-element, such as a new mutation in SLiM, you generally do so by sending a request, via a function call or method call, to the Context.

For those with some programming experience: this means that there is no `new` operator in Eidos as there is in C++, no `malloc()` as in C, no `+alloc` as in Objective-C. Similarly, there is no `delete` operator, no `free()`, no `–dealloc`; just as you never create a new `object`-element directly, you also never dispose of one. Finally, it means that there are no memory management issues in Eidos; deciding when to dispose of an `object`-element is not your problem, it is the Context's problem. In SLiM, for example, a mutation might be disposed of when it is no longer referenced by any genome in the simulation. You might wonder: what happens to the mutation `object`-element in Eidos when that happens? Since your Eidos scripts are never executing at the moment, in-between generations, when SLiM cleans up mutation objects, and since you cannot define an Eidos variable that lives across that boundary, either, that is a question which has no answer. By design, it is impossible for a reference in Eidos to ever be "stale" – to refer to an `object`-element that has been disposed of – assuming the Context has been designed correctly. In fact, Eidos does not even need to garbage-collect; the whole topic of memory management is enclosed by a big Somebody Else's Problem field (hat-tip to Douglas Adams). The way Eidos works may seem strange to those with programming experience, but the advantages should now be clear!

So, returning from philosophy to practicality: `neigh[0]` produces a new house `object`, a vector containing exactly one house. That house is *the same house* as the one in `neigh`. This means that if something is done to change the house in `myHome`, the corresponding house in `neigh` will also change – because they are, in fact, *the same house*. This may seem like a strange thing to emphasize, but it is in fact crucial, as will be seen in the next section.

2.8.3 Properties: operator `.`

What can you do with an `object`? In section 2.8.1 it was stated that objects encapsulate behaviors as well as elements. One type of behavior is called a *property*. A property is a simple attribute of each element in an `object`. For example, following the house `object` example above,

one property of a house might be its address. That would be an example of a read-only property (sometimes called a "member constant"); you can ask a house what its address is, but you can't easily change the address of a house. Properties can be read using the member-access operator, written as `.` (a period). The name of a particular property can be used with `.` to get that property's value. For example, we could (hypothetically) access the addresses of the houses in `neigh`:

```
> neigh.address
"100 Main St." "101 Main St." "102 Main St." ...
```

Notice that since `neigh` is a vector containing several house elements, the result is a vector containing several `string` elements; operations on `object` are vectorized just as they are for all other types in Eidos (see sections 2.2.1 and 2.4.2).

Houses might also have a `color` property, the color they are painted. Since houses can be repainted in a new color, this would be a read-write property (sometimes called a "member variable"). You can use the member-access operator to both read and write properties. For example, using our variable `myHome` that contains just one house (see section 2.8.2), we could do:

```
> myHome.color
"red"
> myHome.color = "blue"
> myHome.color
"blue"
```

We've just repainted our house – and the corresponding house in `neigh` will now also be blue, since they are the same house! This is exactly the behavior that is needed for controlling an external simulation. In your scripts, you may want to play around with `object` vectors that contain mutations, for example; you want to be able to take subsets of those vectors, set properties on the mutations (e.g., change their selection coefficients), and so forth – operating the whole time on the actual mutations in your simulation, not on copies that would fail to affect your simulation run. This is the reason that sharing semantics were chosen as a foundation of Eidos: you will almost always want to be referring to objects that already exist in your simulation, and that you want to collect and manipulate *in situ*.

2.8.4 Multiplexed assignment through properties: operator = revisited

In the previous section, we saw that you can change a property of an element by assigning to a property using the `.` operator, but we did that only for a house `object` containing a single house element. What happens if we have a house `object` with a bunch of house elements in it?

The answer to that is "multiplexed assignment". To see what that means, let's take a step back and ask the same question about simple `integer` vectors; when we saw the assignment operator, `=`, in section 2.4.1, this question got conveniently glossed over. Suppose we have an `integer` vector:

```
> x = 1:10
> x
1 2 3 4 5 6 7 8 9 10
```

We saw in section 2.4.1 that we could replace the value of `x` by assignment, with a statement like `x = "foo"`, but let's try something more interesting instead:

```
> x[3] = 100
> x
1 2 3 100 5 6 7 8 9 10
```

That assignment replaced the element at index `3` with a new element, `100`. So the = operator can be used to replace individual elements, not just to redefine entire variables. It can also be used to replace a larger subset of elements:

```
> x[x % 2 == 0] = 0
> x
1 0 3 0 5 0 7 0 9 0
```

This is "multiplexed assignment": a single value, `0`, has been assigned into multiple elements (in this case, elements that currently contain an even value, as determined by the `logical` vector produced by `x % 2 == 0`).

Assignment can also be used to assign a different value to each element selected by the subset operator:

```
> x[x == 0] = c(-1, -3, -5, -7, -9)
> x
1 -1 3 -3 5 -5 7 -7 9 -9
```

So assignment can either assign the same value to each selected element, or it can assign corresponding values from the right-hand side to the left-hand side of the assignment. If the number of values provided does not fit either of these cases, then it is an error:

```
> x[x < 0] = c(3, 8)
ERROR (_AssignRValueToLValue): assignment to a subscript requires an
rvalue that is a singleton (multiplex assignment) or that has a .size()
matching the .size of the lvalue.
```

That error message refers to the "rvalue" and the "lvalue" of the assignment; the "rvalue" is the value on the right-hand side, and the "lvalue" is the value on the left-hand side.

To return to our original question about assignments involving properties, the answer is that that works in exactly the same way. Given our hypothetical house `object` named `neigh` that contains several house elements, with each house having a property named `color`, we could write:

```
> neigh.color
"blue" "red" "red" "red" "red" ...
> neigh.color = "green"
> neigh.color
"green" "green" "green" "green" "green" ...
> neigh.color = c("red", "blue", "white", "purple", "celadon", ...)
> neigh.color
"red" "blue" "white" "purple" "celadon" ...
```

Furthermore, the `[]` operator can be mixed into this as well:

```
> neigh.color[2:3] = "turquoise"
> neigh.color
"red" "blue" "turquoise" "turquoise" "celadon" ...
> neigh[c(1, 3)].color = "mustard"
> neigh
"red" "mustard" "turquoise" "mustard" "celadon" ...
```

Note that `neigh.color[indices]` is essentially the same as `neigh[indices].color`. They are conceptually distinct; the first form selects particular indices from a vector of `color` properties derived from all of the elements of `x`, while the second form selects particular elements from `x` and then gets the `color` property of the selected elements. In practice, however, the same values are

referred to, and so they are formally equivalent. (This formal equivalence is guaranteed by Eidos, in fact, as a consequence of sharing semantics, combined with a guarantee that read-write properties – although not read-only properties – must have exactly one value per element. This means that there is a one-to-one correspondence between read-write properties and their values, and so it makes no difference in which order `.` and `[]` are performed.)

All of this machinery (modeled on how assignment works in R, incidentally) results in a lot of power contained within the assignment operator. For example, we could institute a neighborhood code that all houses must be beige – except that the guy with the celadon house refuses to repaint:

```
> neigh[neigh.color != "celadon"].color = "beige"
> neigh
"beige" "beige" "beige" "beige" "celadon" ...
```

In the Context of a SLiM simulation, this could be useful for changing the selection coefficient of all neutral mutations to be slightly deleterious, for example, while leaving all other mutations untouched; that operation could be achieved in a single line, with no `for` loops and no `if` statements, using an operation very similar to the above example.

2.8.5 Comparison with `object`: operator ==, !=, <, <=, >, >= revisited

In this discussion of the `object` type we've revisited various operators, such as `[]` and `=`, and seen how they work with `object` operands. There is one more category of operators that we need to revisit: the comparative operators `==`, `!=`, `<`, `<=`, `>`, and `>=` (all previously seen in section 2.3.3). What does it mean to test whether a mutation `object`, for example, is less than or greater than a genomic element `object`? What, in general, does it mean to compare `object` values to each other, and to other types?

Eidos's answer to this question is: not much. Using the relative comparative operators `<`, `<=`, `>`, and `>=` with an `object` operand is simply illegal; no meaning is attached to such comparisons at all:

```
> neigh[2] < neigh[4]
ERROR (Evaluate_Lt): the '<' operator cannot be used with type object.
```

Using the equality operators `==` and `!=` with a mixture of `object` and non-`object` operands is also illegal:

```
> myHome == 5.5
ERROR: operand type object cannot be converted to type float.
```

You may, however, use the equality operators to compare one `object` to another `object`. Two `object`-elements are equal if and only if they are *exactly the same* `object`-element – not different `object`-elements with exactly the same values:

```
> myHome == neigh[0]
T
> myHome == neigh[1]
F
```

(That last comparison would be `F` even if `neigh[1]` had the same address and color as `myHome`, since they are different `object`-elements.) Just as with other operand types (as described in section 2.3.3), this comparison can compare `object`-elements one-to-one, one-to-many, or many-to-one. For example, we can find out which house in `neigh` is our home:

```
> neigh == myHome
T F F F F ...
```

Each element in `neigh` is compared to `myHome`; only the first element of `neigh` is the same `object`-element as the `object`-element in `myHome`, so only the first `logical` value produced is `T`.

2.8.6 Methods: operator `.` and the `methodSignature()` method

Section 2.8.3 introduced one type of `object` behavior, properties. Properties generally represent simple stored values; getting the value of a property might involve a little bit of computation behind the scenes, but conceptually a property is a trait or an attribute of the `object` itself, like the color of a house. The other type of `object` behavior in Eidos is the *method*. Methods are very much like functions; they are chunks of code that you can call to perform tasks. However, each type of `object` has its own particular methods – unlike functions, which are defined globally. Methods are more heavyweight than properties; they might involve quite a lot of computation, they might create a completely new `object` as their result, and they might even modify the `object` upon which they are called. Not all methods are heavyweight in this sort of way, however; anything that one might want an `object` to do, but that does not feel like a simple property of the `object`, can be a method. Methods can also take arguments, just like functions, and they can return whole vectors as their result, unlike (read-write) properties, which must refer to singleton values so that multiplexed assignment can work (see section 2.8.3). Methods are therefore much more powerful than properties.

Eidos does not allow you to define your own methods. There is a set of pre-defined methods for each `object` class, and the hope is that they will do everything that you need done (if not, please let us know). This section will not go into any detail about the pre-defined methods for the `object` classes provided by SLiM; see the SLiM manual for details on that. Instead, here we will continue with the hypothetical neighborhood `object` type introduced above.

Methods are called using the member-access operator, `.`, with a syntax that looks a lot like accessing a property, but combined with the function call operator, `()`. For example, suppose our house element had a method named `setZip()`. This method would find the zip code from within the address of the house, and would change it to a new zip code passed to the method, while keeping the rest of the house's address the same. In code, we could do something like this:

```
> neigh[3].address
"33 N. Wiltshire St., Cambridge, CA 32588"
> neigh[3].setZip(14850)
> neigh[3].address
"33 N. Wiltshire St., Cambridge, CA 14850"
```

Notice that the method is allowed to change `address` even though `address` is a read-only property; methods are allowed more latitude than properties are, and can change the state of an `object` in whatever ways are allowed by the underlying code in which they are implemented.

Naturally, method calls are also vector operations, so we could change the zip code of our whole neighborhood in a single statement:

```
> neigh.setZip(54321)
```

Or if we had a `getZip()` method that extracted the current zip code from the address, we could change the zip codes only for houses with an old zip code that needed updating:

```
> neigh[neigh.getZip() == 65432].setZip(23456)
```

The possibilities are endless, obviously – limited only by the facilities provided by the Context within which Eidos is being used.

In section 2.7.4 we encountered the `functionSignature()` function, which printed out all of the functions known to Eidos in the form of function signatures. There is a parallel method, the

`methodSignature()` method, for printing out all of the methods defined for a particular object in the form of method signatures. For example, we could do a little introspection on our neighborhood `object`, `neigh`:

```
> neigh.methodSignature()
- (integer$)getZip(void)
+ (void)methodSignature([string$])
+ (void)propertySignature([string$])
- (void)setZip(integer$)
- (void)str(void)
```

This output tells us that there are five methods defined on our house element (and thus on our neighborhood `object`): the `getZip()` and `setZip()` methods that we just discussed, the `methodSignature()` method that we are discussing now, and two more, `propertySignature()` and `str()`, that we will discuss momentarily.

Notice that `getZip()` is declared as returning `integer$`, a singleton `integer`. This might seem a bit surprising, since above we called `neigh.getZip()` to get a whole vector containing a zip code for each house in the neighborhood; that sure doesn't look like a singleton! The explanation is that when an Eidos method is declared as returning a singleton, that means that a single value will be returned *per element*. The overall return value from the method call will therefore contain exactly the same number of elements as the target `object` upon which the method was called; there will be a one-to-one correspondence between the elements in the target and in the return value. When a method is declared as *not* having a singleton return value, that means that each element in the target might return any number of values in response; all of the returned values from all of the elements in the target will be concatenated together to form the final return value. In this case, there may not be a one-to-one correspondence between elements in the target and in the return. The semantics here are identical to those for properties, as a matter of fact; the `address` and `color` properties of our hypothetical house object are singleton properties, meaning that the properties have exactly one value per element in the target `object` (not one value total per `object`).

There is an important difference between properties and methods in a related aspect of their treatment of singletons, however. If you assign a multi-valued vector into a property, the assignment will be multiplexed out into the elements of the target `object`, as we saw in section 2.8.4; a one-to-one correspondence between values in the target and in the vector assigned to the target is used. If you pass a multi-valued vector to a method, however, no multiplexing occurs; each element in the target of the method call receives the full multi-valued vector as a parameter. When `setZip()` is declared above as taking `integer$`, therefore, that means that you may only pass a singleton to the method, and that singleton value will be used in the method call for each element in the target. For example, if you call `setZip(14850)` on `neigh`, every house in `neigh` will be given the same zip code, `14850`. There is no way to use a single call to `setZip()` to assign a different zip code individually to each house in `neigh`, the way you could with multiplexed assignment if the zip code were a property; you may not pass a vector of zip codes to `setZip()`, and even if you could, each house in `neigh` would then receive a `setZip()` call with the entire vector of zip codes, and would not know which zip code it was supposed to adopt (which is why the method signature for `setZip()` does not allow that in the first place). This is a fundamental difference between properties and methods that should be understood; properties use multiplexed assignment, whereas methods pass the full argument list to every element in the target.

Another thing you might notice is that these method signatures begin with a leading + or –, unlike the function signatures we saw in section 2.7.4. This notation is borrowed from Objective-C; it denotes whether a method is a *class method* (with a +) or an *instance method* (with a –). In some languages this is a very strong distinction, between a *class* – representing the abstract concept of a neighborhood, for example – and an *instance* – representing a specific neighborhood

with particular houses in it. In Eidos this distinction is less important, because classes are not very visible in Eidos; there is no way to get a class `object`, there is no visible inheritance hierarchy among classes, and so forth. However, the difference does manifest with methods in one significant way: a class method is called just a single time, even when it is called on an `object` that contains several elements, because the method is being called on the class, which is shared by all the elements. Since `methodSignature()` is a class method, this happened with the call above, in fact, but you may not have noticed! If `methodSignature()` had been called on each element – each house inside `neigh` – then the list of method signatures would have been printed once for each element, as `methodSignature()` was called on each element. Instead, `methodSignature()` was called just once, on the class. In other words, `methodSignature()` does not ask elements "what methods do you respond to?", but instead asks the class "what methods would all instances of you respond to?" If that explanation is clear as mud, here's another way to look at it: instance methods multicast out to every `object` element in the `object` they are sent to, whereas class methods "singlecast" to the class to allow it to handle the method on behalf of all of the `object` elements. In other languages, this might be represented by a "static method" or a "static member function" that takes a vector (or an array) of objects as a parameter and performs some operation on the whole vector; in Eidos, the same syntax is used as for an instance method dispatch.

Note that `methodSignature()` takes an optional singleton `string` argument; as with `functionSignature()`, this allows us to get the method signature for a specific method:

```
> neigh.methodSignature("propertySignature")
+ (void)propertySignature([string$])
```

So what is the `propertySignature()` method, then? It is actually an `object` introspection method as well, like `methodSignature()`, but it shows us the *properties* that are defined by a given `object` class. In a sense, it is very similar to the `ls()` function (see section 2.4.1); but the `ls()` function lists globally defined constants and variables, whereas the `propertySignature()` method shows us the properties defined by an object. For example, we could call it on our neighborhood vector, `neigh`:

```
> neigh.propertySignature()
address => (string$)
color <-> (string$)
```

This output tells us that two properties are defined for `neigh` (or for the neighborhood class, really, since `propertySignature()` is a class method). One is a read-only property (as shown by the `=>` arrow) named `address`; that property has type `string` and is a singleton, meaning that a given house element has exactly one address. The other is a read-write property (as shown by the `<->` arrow) named `color`; that property also has type `string` and is also a singleton, since a given house element has exactly one color. All of this should be quite reminiscent of the `ls()` function that we saw in section 2.4.1.

There was one more method shown above in the output from `neigh.methodSignature()`: the `str()` method. Let's try calling it on `neigh` to see what it does:

```
> neigh.str()
House:
   address => (string$) "100 Main St." "101 Main St." ... (10 values)
   color <-> (string$) "beige" "beige" ... (10 values)
```

So it is similar to `propertySignature()`, but it shows us some additional information about the `object`: what class it is (`House`), how many values each property has (`10`, in this case), and what the first couple of values look like. The name "str" is short for "structure"; the idea is that this method gives you a sense of the internal structure of an `object`.

Incidentally, `methodSignature()`, `propertySignature()`, and `str()` are defined for all `object` classes in Eidos; they are in no way particular to the hypothetical neighborhood class we've been exploring. They are formally documented in section 3.10.

2.9 Matrices and arrays

2.9.1 Defining matrices and arrays

Thus far, our mantra has been "everything is a vector". We are not going to deviate from that now, but we're going to add a new layer onto it: sometimes, a vector is *also* a matrix or an array. The way that matrices and arrays are handled in Eidos is very parallel to how they are handled in R; minor differences will be highlighted below, but experienced R users will find few surprises (but see the discussion of "dropping" in section 2.9.4, which is one major difference). Note that matrices and arrays are an advanced topic; new users might wish to skip this whole section.

Conceptually, a matrix is a vector with a two-dimensional structure imposed upon it, such as:

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

The matrix above could be constructed in Eidos using the `matrix()` function:

```
> x = matrix(1:6, nrow=2)
> x
     [,0] [,1] [,2]
[0,]    1    3    5
[1,]    2    4    6
```

We specify to `matrix()` that we want two rows, and it infers from the length of the data vector provided that three columns will thus be needed (one may specify the number of columns instead, or both, or neither). It fills the values in the matrix by column, by default, so this produces the same vector as above; supplying `byrow=T` to `matrix()` would fill values by row instead.

Note that `x` really is still a vector:

```
> x[2]
3
> size(x)
6
> c(x)
1 2 3 4 5 6
```

The matrix-ness of it is just a veneer; normal subscripting with `[]` can still refer to the values in the vector, `size()` still returns the number of elements, and the `c()` function will strip away the matrix veneer to reveal the underlying vector. In R, this veneer is made quite explicit in that a matrix is simply a vector that has had an attribute named `$dim` attached to it; in Eidos the internal design is much the same, but the dimension attribute is not directly accessible in code.

Similarly, an array is a vector with a two-dimensional *or greater* structure imposed upon it, here is a depiction of a three-dimensional one, with two "planes" each of which is a 2×3 matrix:

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 9 & 11 \\ 8 & 10 & 12 \end{bmatrix}$$

This array could be constructed in Eidos using the `array()` function:

```
> y = array(1:12, c(2,3,2))
> y
, , 0

     [,0] [,1] [,2]
[0,]    1    3    5
[1,]    2    4    6

, , 1

     [,0] [,1] [,2]
[0,]    7    9   11
[1,]    8   10   12
```

This function takes a data vector first, as `matrix()` did, and then takes a vector of dimension sizes. The first entry in that vector is the number of rows; here, `2`. The second is the number of columns; here, `3`. Further entries give the size of further, higher-order dimensions; here we specify just one, of size `2`, meaning that the array is a size `2` array of matrices, making three dimensions in all. Any number of higher-order dimensions can be used, although it gets difficult to visualize.

Again, this array is still really a vector at heart; the array-ness is just a veneer:

```
> y[7]
8
> size(y)
12
> c(y)
1 2 3 4 5 6 7 8 9 10 11 12
```

Since arrays can be two-dimensional or greater, a matrix is really a special case of an array, but to be clear we will usually use "matrix" to refer specifically to the two-dimensional case, and "matrix or array" when we want to be more general.

2.9.2 Matrix and array attributes

As we saw in the previous section, matrices and arrays are just vectors with a dimension attribute. When working with them, it is useful to be able to distinguish them from plain vectors, get their dimensions, and so forth. Several functions are provided for this purpose.

First of all, `dim()` can be used to get the dimensions of a matrix:

```
> dim(x)
2 3
```

For vectors that are not matrices or arrays, `dim()` will return `NULL`, so it can also be used as a way to distinguish matrices and arrays from plain vectors.

Second, the number of rows or columns of a matrix can be queried separately using the `nrow()` and `ncol()` functions:

```
> nrow(x)
2
> ncol(x)
3
```

These functions also work with arrays, but the size of higher-order array dimensions is not accessible with `nrow()` and `ncol()`; `dim()` should be used for that:

```
> nrow(y)
2
> ncol(y)
3
> dim(y)
2 3 2
```

As with `dim()`, `nrow()` and `ncol()` will return `NULL` for plain vectors. R provides special functions named `NROW()` and `NCOL()` that work in the same way as `nrow()` and `ncol()`, but that additionally treat plain vectors as one-column matrices; Eidos does not provide these functions, but it would be trivial to define them as user-defined functions (see chapter 4).

2.9.3 Using Eidos operators and built-in functions with matrices and arrays

The fact that matrices and arrays are really just vectors is very important, because in many cases Eidos (like R) will treat them as vectors, simply ignoring their dimension attribute. In other cases, however, the dimension attribute is important, and the behavior of Eidos is different with matrices and arrays than it would be with the corresponding plain vectors. This section will attempt to clarify and summarize this behavior.

First of all, Eidos has many binary operators that perform an operation on two operands that involves some sort of combination or comparison of the operands; examples would include the operators `+`, `-`, `*`, `/`, `%`, `^`, `&`, `|`, `==`, `!=`, `<`, `<=`, `>`, and `>=`. The behavior of all of these with matrix or array operands is the same: if both operands are matrices/arrays, the operands must have exactly the same dimensions, and the operation is then performed elementwise on the operands, and results in a matrix of the same dimensions. For example:

```
> x = matrix(1:6, nrow=2)
> y = matrix(rep(1:2, 3), nrow=2)
> x
     [,0] [,1] [,2]
[0,]    1    3    5
[1,]    2    4    6
> y
     [,0] [,1] [,2]
[0,]    1    1    1
[1,]    2    2    2
> x + y
     [,0] [,1] [,2]
[0,]    2    4    6
[1,]    4    6    8
```

Each element of x + y is the sum of the corresponding elements of x and y. If x and y did not have the same dimensions, an error would result; they would be said to be "non-conformable" (which does not mean "not the same size", but rather, "of sizes that are incompatible with each other for the operation requested").

These binary operators can also handle one singleton operand and one non-singleton matrix or array operand. In this case, the operation is performed between the singleton operand and every individual element of the matrix/array operand, generating a result that is a matrix/array with the same dimensionality as the non-singleton operand. For example (using `x` as defined above):

```
> x + 1
     [,0] [,1] [,2]
```

```
[0,]    2    4    6
[1,]    3    5    7
```

The result would be the same with `1 + x` rather than `x + 1`.

Second, there are unary operators that perform some transformation upon a single operand; examples include unary – and +, as well as the `!` operator. With matrix/array operands, these operators produce a result with the same dimensions as their operand. For example:

```
> z = matrix(c(T,F,T,F,F), nrow=1)
> z
     [,0] [,1] [,2] [,3] [,4]
[0,]    T    F    T    F    F
> !z
     [,0] [,1] [,2] [,3] [,4]
[0,]    F    T    F    T    T
```

Here `z` is a "row matrix" (a matrix comprised of a single row), and the `!` operator preserves that structure. This example also illustrates that matrices and arrays in Eidos may be of any type; here we have a `logical` matrix.

There are just a few operators left. The range operator, `:`, does not allow matrices/arrays as operands at all; note that that is a much stricter policy than R. The assignment operator, `=`, preserves the dimensions of the value, as we have already seen in the examples above.

The `.` operator is a bit more complicated. When used to call a method, it ignores the dimensions of the target object and always returns a plain vector. This is necessary for consistency, because methods are allowed to return `NULL` to indicate a non-fatal error condition, which means the length of the result of a method call, even if declared to return singleton, is not necessarily the same as the length of the target object. When used to access a non-singleton property, the `.` operator will similarly ignore the target's dimensions and return a plain vector, because again the length of the result may not match the length of the target. When used to access a singleton property, however, the `.` operator will produce a result with the same dimensionality as the target object, just as the unary operators do, since it is possible to do so in that case (since the length of the result is guaranteed to equal the length of the target), and since that behavior is useful. In the other cases, if you wish to re-impose a particular dimensionality on a vector you can always use `matrix()` or `array()` to do so.

Finally, there is the subset operator, `[]`, which we have left for last because it is the most complicated. In fact, it is such a complex topic that we will push it out to the next subsection.

Except for the deferral of operator `[]`, that completes our discussion of the Eidos operators with matrices and arrays. Next, there are also the built-in functions to be considered.

First of all, there are functions which can be thought of as unary operators, in a sense; they perform some elementwise transformation upon a single operand, their one argument. Such functions include mathematical operations like `abs()` and `ceil()` and `cos()` and `sqrt()`, as well as tests like `isFinite()`, type transformations like `asFloat()`, and string operations like `nchar()`. In general, these functions will accept a matrix or array operand, and they will return a result with the same dimensions; each element is transformed individually.

Second, there are functions that in some way summarize their one argument down to a single value, such as mathematical operations like `sum()` and `product()`, logical operations like `all()` and `any()`, type tests like `isFloat()`, and summary statistics functions like `min()` and `mean()`. In general, these functions will ignore the dimensions of their operand, treat it as a plain vector, and return a plain vector result.

Third, there are functions that are like binary operators, taking two operands and performing an operation that in some way combines or compares them. The behavior of these is more case-by-case. The `atan2()` function has the same conformability requirements and behavior as the binary operators discussed above, and the same is basically true of `pmax()` and `pmin()`. The set logic functions like `setUnion()` and `setDifference()`, on the other hand, ignore dimensions and always return a plain vector result. The `identical()` function accepts matrices and arrays, and will return `T` only if the dimensions of its arguments are exactly the same (in addition to all of the elements being the same, as usual).

Fourth, there are many built-in functions that don't fall into these categories and that simply ignore the dimensions of their arguments in all cases. Examples include the distribution functions like `dnorm()` and `runif()` and `sample()`, functions that reorder or concatenate their argument like `c()` and `rep()` and `sort()`, and functions that subset or match their argument like `match()`, `unique()`, and `which()`. All of these functions ignore the dimensionality or their argument(s), treat them as plain vectors, and return the same result as they would return otherwise.

Fifth, there are a few outliers that have some special behavior. Some of these are functions specifically for working with matrices and arrays, such as `cbind()` and `rbind()` for combining matrices by column or row, `matrixMult()` for matrix multiplication, `t()` for matrix transposition, and `apply()` for iterating over the margins of matrices and arrays. Others are simply unusual functions, like `ifelse()` and `sapply()`. Some higher-level output functions, such as `str()` and `print()`, treat matrices and arrays in a special way in order to produce more useful output, but lower-level output functions such as `cat()` and `catn()` ignore the dimension of their arguments.

For any edge cases not discussed above, you might consult the reference documentation in chapter 3; however, the documentation on each function will not exhaustively discuss how matrices and arrays are treated, because that is just too tedious. Instead, it will generally just call attention to unusual cases and exceptions from these rules. If the reference documentation is insufficient, you might try an experiment to answer your question – really that is often the best way to get an answer.

Finally, there are the Eidos language keywords to be considered, such as `if`, `for`, `while`, and `return`. The answer there is simple: they all just ignore the dimensions of their operands, not modifying those dimensions nor using them for any purpose.

Broadly, these rules are very similar to those in R, although there are some exceptions. As a rule, R is more forgiving than Eidos about such things, and tries to make sense out of mismatches by doing things like promoting vectors to matrices or repeating the values in a vector to fill out its length, in order to achieve conformability and make a given operation happen. Eidos will, in many cases, produce an error instead, as a matter of design philosophy, since jumping through such hoops will often produce unexpected behavior that leads to bugs; an error message is often preferable in real-world usage, especially for relatively inexperienced programmers.

2.9.4 Subsets of matrices and arrays using operator []

In the previous section we saw how Eidos treats matrix and array values with all of its operators, built-in functions, and language keywords, with one exception: subsetting with the `[]` operator. We will cover that topic now.

First of all, subsetting can be done on a matrix or array by treating it as a plain vector, as we have seen above. In that case, the elements of the matrix or array are addressed with a single zero-based index that counts upward, in column order, through the dimension hierarchy, just as the values are laid out by `array()`. Subsetting with a `logical` vector also works with matrices and arrays. The vector-based behavior of subsetting is thus unchanged with matrices and arrays.

In addition, however, with matrices and arrays it is possible to subset by each dimension independently. This is easier to illustrate than describe:

```
> x = matrix(1:6, nrow=2)
> x
     [,0] [,1] [,2]
[0,]    1    3    5
[1,]    2    4    6
> x[1,2]
     [,0]
[0,]    6
```

Here the subset operator selects the value from row `1` and column `2` of `x` (keeping in mind that indices in Eidos are zero-based), which is `6`. The dimensionality of `x` is preserved by the subset operation, so since `x` is a matrix, this slice of `x` is a matrix as well, just with fewer rows and columns. And perhaps this is obvious, but note that the way `x` was printed above made it easy to predict what `x[1,2]` would be; just read across the row labeled `[1,]` to the column labeled `[,2]`. That is, of course, why matrices are printed in that manner.

If a given dimension is omitted entirely, all indices in that dimension are selected:

```
> x[0,]
     [,0] [,1] [,2]
[0,]    1    3    5
> x[,1]
     [,0]
[0,]    3
[1,]    4
```

Here we select row `0` of `x` across all columns, and then column `1` of `x` across all rows. Again the results are matrices, preserving the rank (the number of dimensions) of `x`. The indices selected for each dimension may be in any order, and may repeat; the subset operator will select whatever values are requested:

```
> x[c(0,0), c(2,0)]
     [,0] [,1]
[0,]    5    1
[1,]    5    1
```

Here row `0` is selected twice, and columns `2` and `0` are selected, generating the result matrix shown, which repeats row `0` twice but only including the values from column `2` and `0` (in that order, as specified).

The indices to select from a given dimension may also be specified with a `logical` vector, just as we could do with the simpler vector-based subsetting discussed in section 2.2.4. For example:

```
> x[1:0, c(T,F,T)]
     [,0] [,1]
[0,]    2    6
[1,]    1    5
```

This selects rows `1` and `0` (in that order) and columns `0` and `2`, producing the result shown. When a `logical` vector is used to select indices for a given dimension, the length of the `logical` vector must be equal to the size of the dimension; each `logical` value is then taken as including (`T`) or excluding (`F`) the corresponding index of the dimension.

So to summarize, indices for a given dimension may be selected in any of three ways: (1) by omitting the specification entirely, as `x[0,]` selects all columns within row `0`, (2) by specifying `integer` indices (or `float` indices, which are rounded down) to select indices in the order specified, or (3) by specifying a `logical` vector that selects the indices corresponding to `T` values.

Specifying `NULL` for a dimension is exactly the same as omitting any specifier for that dimension; `x[0,NULL]` is thus the same as `x[0,]`.

All of this works in exactly the same way for higher-rank arrays. There is no need to belabor that point, so let's look at just one example with an array `y`:

```
> y = array(1:12, c(2,3,2))
> y
, , 0

     [,0] [,1] [,2]
[0,]    1    3    5
[1,]    2    4    6

, , 1

     [,0] [,1] [,2]
[0,]    7    9   11
[1,]    8   10   12

> y[1, c(F,T,T), ]
, , 0

     [,0] [,1]
[0,]    4    6

, , 1

     [,0] [,1]
[0,]   10   12
```

This shows all of the types of index selection described above: row `1` is elected using its index, columns `1` and `2` are selected with a `logical` vector, and both planes of the array are selected since the third specifier was omitted entirely. This extends to any number of dimensions.

Much of this is exactly as it is in R, but those who are familiar with R may have noticed one striking difference between the behavior of Eidos and that of R: R drops unneeded dimensions when subsetting, by default, whereas Eidos does not. We saw this in many of the examples above, such as this example using the matrix `x` that we defined above:

```
> x[1,2]
     [,0]
[0,]    6
```

As mentioned above, Eidos preserves the dimensionality of `x`, returning a matrix from the subset operation even though the result is a singleton. R provides that option as well, with an optional "drop" flag; in R that would be written as `x[1,2,drop=F]`, and would produce the result above. Eidos has essentially the opposite policy: subsetting does *not* drop dimensions, and if you want to drop them you must call a function named `drop()` explicitly to do so. For example:

```
> drop(x[1,2])
6
```

The value `6` is now "unwrapped" from the redundant dimensions it was enclosed in, and is returned as a plain vector. Similarly:

```
> drop(x[0,])
1 3 5
```

Since there was only one row in the result, that redundant dimension is dropped and the result is again a plain vector. Importantly, however, only dimensions of size `1` are dropped; others are preserved, so the result of `drop()` is not always a plain vector, it is just a matrix or array of the *minimal* dimensions needed to preserve the structure of the data. (Of course, if you do want to remove all dimension structure completely, `c()` will do that as mentioned previously.) The array-based example above with `y` provides a good example of a more complex `drop()` operation:

```
> drop(y[1, c(F,T,T), ])
     [,0] [,1]
[0,]    4   10
[1,]    6   12
```

Compare this to the same example above without the `drop()` call to see what `drop()` has done here. Since the result of the subset operation had only one row, that redundant dimension was dropped; but it had two columns and two planes, so those non-redundant dimensions were preserved. Since the row dimension was removed, the columns become rows and the planes become columns – everybody slides down one! – producing the result above.

As you can see, the behavior of `drop()` can be complex and counterintuitive. It can also be unpredictable; calling `drop()` on the result of subsetting a three-dimensional array like `y` could result in a plain vector, a matrix, or an array, depending upon how many indices were selected from each dimension. R's behavior, of automatically dropping unless requested not to do so, therefore often results in unexpected and unpredictable behavior – if not outright bugs. I have been bitten by that "feature" of R more than once, so when designing Eidos I decided that this would be a rare case in which I would deliberately break with R's precedent. If you are used to R's behavior, this shift may take a little adjustment.

This completes our discussion of matrices and arrays. Note, however, that there are several functions such as `cbind()`, `rbind()`, `matrixMult()`, and `t()` that we have not introduced here, but that are specifically designed for working with matrices; if you will be using matrices you may wish to consult their documentation in chapter 3.7.

2.10 Comments

2.10.1 Single-line comments with `//`

Section 2.6.5 briefly introduced the concept of comments in Eidos. For completeness, we ought to discuss them in a little more detail.

Technically, comments are actually a type of whitespace; comments in Eidos code are completely ignored and have no effect whatsoever on the results of the execution of code, just like other kinds of whitespace in most respects. Single-line comments begin with `//` and then may consist of any text whatsoever, up to the end of the current line of code. A comment may occur by itself on a line, or it may follow other Eidos code. So for example, you could write:

```
1 + 1 == 2;    // this is true
```

Comments are never required in Eidos, but using them to annotate your code is a very good idea, both so that you remember what your intentions were when you come back to the code later, and so that others who might need to understand or maintain your code have a helping hand.

2.10.2 Block comments with `/* */`

It is possible to comment out whole blocks of script, instead of just single lines. This can be useful for writing longer comments that describe a section of code in more detail. In Eidos (as in C and C++), such block comments can be written with a beginning `/*` and a terminating `*/`:

```
/*
   This computes the factorial x!, which is
   the product of all values from 1 to x, for
   any positive integer x.
*/
x_factorial = product(1:x);
```

A nice feature of Eidos is that block comments nest properly, making it possible to use them to comment out stretches of code that already contain block comments. For example, if the code above was no longer needed, but you didn't want to delete it entirely because you might need it again later, you could use a block comment to disable it:

```
/*  ******* NOT NEEDED ************
/*
   This computes the factorial x!, which is
   the product of all values from 1 to x, for
   any positive integer x.
*/
x_factorial = product(1:x);
*/
```

The outer block comment is not terminated by the first `*/` because Eidos recognizes that that belongs to the inner block comment; the outer block comment continues until the second `*/` is encountered.

3. Built-in functions and methods

The built-in functions presented here are intended to provide broad, general-purpose support for most common tasks. If you find that you need a reasonably common function that has not been provided, such as a math function, please let us know. You can also define your own functions (see chapter 4), which is an easy way to extend the base functionality of Eidos. Finally, you might want to try out the `executeLambda()` function described in section 3.9, which provides a cheap substitute for user-defined functions.

Functions are listed here by category (math, summary statistics, vector construction, value inspection/manipulation, value typing/coercion, matrix and array, filesystem access, color manipulation, and other miscellaneous functions). Within each category functions are listed alphabetically. For each function, the function signature is shown (see section 2.7.4), and then a brief description is given.

It might be worth noting here that the basic philosophy of Eidos is that of a *functional language*: operations in Eidos (other than assignment) generally do not modify existing values, but instead generate new values. If you wish a vector `x` to become sorted, you execute `x = sort(x)` instead of simply `sort(x)`; the `sort()` function does not modify `x`, but rather produces a new value which must be assigned back into `x` if you want the new value to replace the old. Eidos follows R in adopting this philosophy because it limits unexpected side effects. The `object`-elements in Eidos violate this philosophy; when you set a property or call a method on an `object`-element, you are changing the `object`-element itself (for the reasons discussed in section 2.8.2). The built-in types and functions of Eidos, however, follow this philosophy.

3.1 Math functions

The math functions in Eidos are pattered closely upon those in C++, and are typically implemented by calling a C++ function of the same name, as described below when applicable.

`(numeric)abs(numeric x)`
Returns the **absolute value** of `x`. If `x` is `integer`, the C++ function `llabs()` is used and an `integer` vector is returned; if `x` is `float`, the C++ function `fabs()` is used and a `float` vector is returned.

`(float)acos(numeric x)`
Returns the **arc cosine** of `x` using the C++ function `acos()`.

`(float)asin(numeric x)`
Returns the **arc sine** of `x` using the C++ function `asin()`.

`(float)atan(numeric x)`
Returns the **arc tangent** of `x` using the C++ function `atan()`.

`(float)atan2(numeric x, numeric y)`
Returns the **arc tangent** of `y/x` using the C++ function `atan2()`, which uses the signs of both x and y to determine the correct quadrant for the result.

`(float)ceil(float x)`
Returns the **ceiling** of `x`: the smallest integral value greater than or equal to `x`. Note that the return value is `float` even though integral values are guaranteed, because values could be outside of the range representable by `integer`.

`(float)cos(numeric x)`
Returns the **cosine** of `x` using the C++ function `cos()`.

`(numeric)cumProduct(numeric x)`

Returns the **cumulative product** of `x`: a vector of equal length as `x`, in which the element at index `i` is equal to the product of the elements of `x` across the range `0:i`. The return type will match the type of `x`. If `x` is of type `integer`, but all of the values of the cumulative product vector cannot be represented in that type, an error condition will result.

`(numeric)cumSum(numeric x)`

Returns the **cumulative sum** of `x`: a vector of equal length as `x`, in which the element at index `i` is equal to the sum of the elements of `x` across the range `0:i`. The return type will match the type of `x`. If `x` is of type `integer`, but all of the values of the cumulative sum vector cannot be represented in that type, an error condition will result.

`(float)exp(numeric x)`

Returns the **base-e exponential** of `x`, e^x, using the C++ function `exp()`. This may be somewhat faster than `E^x` for large vectors.

`(float)floor(float x)`

Returns the **floor** of `x`: the largest integral value less than or equal to `x`. Note that the return value is `float` even though integral values are guaranteed, because values could be outside of the range representable by `integer`.

`(integer)integerDiv(integer x, integer y)`

Returns the result of **integer division** of `x` by `y`. The `/` operator in Eidos always produces a `float` result; if you want an `integer` result you may use this function instead. If any value of `y` is `0`, an error will result. The parameters `x` and `y` must either be of equal length, or one of the two must be a singleton. The precise behavior of `integer` division, in terms of how rounding and negative values are handled, may be platform dependent; it will be whatever the C++ behavior of `integer` division is on the given platform. Eidos does not guarantee any particular behavior, so use this function with caution.

`(integer)integerMod(integer x, integer y)`

Returns the result of **integer modulo** of `x` by `y`. The `%` operator in Eidos always produces a `float` result; if you want an `integer` result you may use this function instead. If any value of `y` is `0`, an error will result. The parameters `x` and `y` must either be of equal length, or one of the two must be a singleton. The precise behavior of `integer` modulo, in terms of how rounding and negative values are handled, may be platform dependent; it will be whatever the C++ behavior of `integer` modulo is on the given platform. Eidos does not guarantee any particular behavior, so use this function with caution.

`(logical)isFinite(float x)`

Returns the **finiteness** of `x`: `T` if `x` is not `INF` or `NAN`, `F` if `x` is `INF` or `NAN`. `INF` and `NAN` are defined only for type `float`, so x is required to be a `float`. Note that `isFinite()` is not the opposite of `isInfinite()`, because `NAN` is considered to be neither finite nor infinite.

`(logical)isInfinite(float x)`

Returns the **infiniteness** of `x`: `T` if `x` is `INF`, `F` otherwise. `INF` is defined only for type `float`, so x is required to be a `float`. Note that `isInfinite()` is not the opposite of `isFinite()`, because `NAN` is considered to be neither finite nor infinite.

`(logical)isNAN(float x)`

Returns the **undefinedness** of `x`: `T` if `x` is not `NAN`, `F` if `x` is `NAN`. `NAN` is defined only for type `float`, so x is required to be a `float`.

`(float)log(numeric x)`

Returns the **base-e logarithm** of `x` using the C++ function `log()`.

`(float)log10(numeric x)`

Returns the **base-10 logarithm** of `x` using the C++ function `log10()`.

`(float)log2(numeric x)`

Returns the **base-2 logarithm** of `x` using the C++ function `log2()`.

`(numeric$)product(numeric x)`

Returns the **product** of `x`: the result of multiplying all of the elements of `x` together. If `x` is `float`, the result will be `float`. If `x` is `integer`, things are a bit more complex; the result will be `integer` if it can fit into the `integer` type without overflow issues (including during intermediate stages of the computation), otherwise it will be `float`.

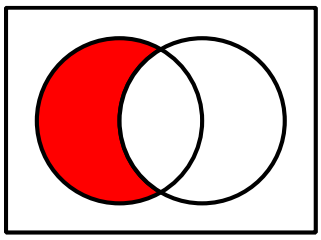
`(float)round(float x)`

Returns the **round** of `x`: the integral value nearest to `x`, rounding half-way cases away from `0` (different from the rounding policy of R, which rounds halfway cases toward the nearest even number). Note that the return value is `float` even though integral values are guaranteed, because values could be outside of the range representable by `integer`.

`(*)setDifference(* x, * y)`

Returns the **set-theoretic (asymmetric) difference** of `x` and `y`, denoted `x` ∖ `y`: a vector containing all elements that are in `x` but are not in `y`. Duplicate elements will be stripped out, in the same manner as the `unique()` function. The order of elements in the returned vector is arbitrary and should not be relied upon. The returned vector will be of the same type as `x` and `y`, and `x` and `y` must be of the same type.

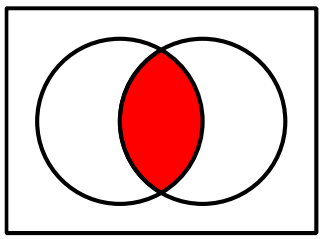
The operation performed by this function can be represented graphically using a Venn diagram, where the left circle is `x` and the right circle is `y`:



`(*)setIntersection(* x, * y)`

Returns the **set-theoretic intersection** of `x` and `y`, denoted `x` ∩ `y`: a vector containing all elements that are in both `x` and `y` (but not in *only* `x` or `y`). Duplicate elements will be stripped out, in the same manner as the `unique()` function. The order of elements in the returned vector is arbitrary and should not be relied upon. The returned vector will be of the same type as `x` and `y`, and `x` and `y` must be of the same type.

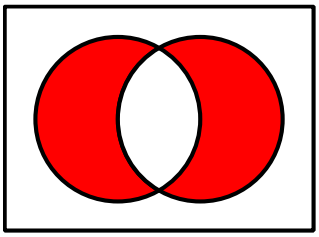
The operation performed by this function can be represented graphically using a Venn diagram, where the left circle is `x` and the right circle is `y`:



`(*)setSymmetricDifference(* x, * y)`

Returns the **set-theoretic symmetric difference** of `x` and `y`, denoted `x` Δ `y`: a vector containing all elements that are in `x` or `y`, but not in both. Duplicate elements will be stripped out, in the same manner as the `unique()` function. The order of elements in the returned vector is arbitrary and should not be relied upon. The returned vector will be of the same type as `x` and `y`, and `x` and `y` must be of the same type.

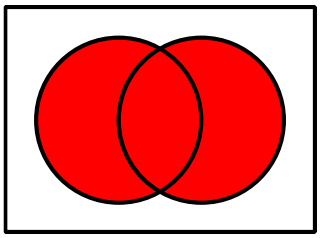
The operation performed by this function can be represented graphically using a Venn diagram, where the left circle is `x` and the right circle is `y`:



`(*)setUnion(* x, * y)`

Returns the **set-theoretic union** of `x` and `y`, denoted `x` $\cup$ `y`: a vector containing all elements that are in `x` and/or `y`. Duplicate elements will be stripped out, in the same manner as the `unique()` function. This function is therefore roughly equivalent to `unique(c(x, y))`, but this function will probably be faster. The order of elements in the returned vector is arbitrary and should not be relied upon. The returned vector will be of the same type as `x` and `y`, and `x` and `y` must be of the same type.

The operation performed by this function can be represented graphically using a Venn diagram, where the left circle is `x` and the right circle is `y`:



`(float)sin(numeric x)`

Returns the **sine** of `x` using the C++ function `sin()`.

`(float)sqrt(numeric x)`

Returns the **square root** of `x` using the C++ function `sqrt()`. This may be somewhat faster than `x^0.5` for large vectors.

`(numeric$)sum(lif x)`

Returns the **sum** of `x`: the result of adding all of the elements of `x` together. The unusual parameter type signature `lif` indicates that `x` can be `logical`, `integer`, or `float`. If `x` is `float`, the result will be `float`. If `x` is `logical`, the result will be `integer` (the number of `T` values in `x`, since the `integer` values of `T` and `F` are `1` and `0` respectively). If `x` is `integer`, things are a bit more complex; in this case, the result will be `integer` if it can fit into the `integer` type without overflow issues (including during intermediate stages of the computation), otherwise it will be `float`. Note that floating-point roundoff issues can cause this function to return inexact results when `x` is `float` type; this is rarely an issue, but see the `sumExact()` function for an alternative.

`(float$)sumExact(float x)`

Returns the **exact sum** of `x`: the exact result of adding all of the elements of `x` together. Unlike the `sum()` function, `sumExact()` accepts only type `float`, since the `sum()` function is already exact for other types. When summing floating-point values – particularly values that vary across many orders of magnitude – the precision limits of floating-point numbers can lead to roundoff errors that cause the `sum()` function to return an inexact result. This function does additional work to ensure that the final result is exact within the possible limits of the `float` type; some roundoff may still inevitably occur, in other words, but a more exact result could not be represented with a value of type `float`. The disadvantage of using this function instead of `sum()` is that it is much slower – about 35 times slower, according to one test on Mac OS X 10.2.5, but that will vary across operating systems and hardware. This function is rarely truly needed, but apart from the performance consequences there is no disadvantage to using it.

`(float)tan(numeric x)`

Returns the **tangent** of `x` using the C++ function `tan()`.

`(float)trunc(float x)`

Returns the **truncation** of `x`: the integral value nearest to, but no larger in magnitude than, `x`. Note that the return value is `float` even though integral values are guaranteed, because values could be outside of the range representable by `integer`.

3.2 Statistics functions

`(float$)cor(numeric x, numeric y)`

Returns the **sample Pearson's correlation coefficient** between `x` and `y`, usually denoted *r*. The sizes of `x` and `y` must be identical. If `x` and `y` have a size of `0` or `1`, the return value will be `NULL`. At present it is illegal to call `cor()` with a matrix or array argument, because the desired behavior in that case has not yet been implemented.

`(float$)cov(numeric x, numeric y)`

Returns the **corrected sample covariance** between `x` and `y`. The sizes of `x` and `y` must be identical. If `x` and `y` have a size of `0` or `1`, the return value will be `NULL`. At present it is illegal to call `cov()` with a matrix or array argument, because the desired behavior in that case has not yet been implemented.

`(+$)max(+ x, ...)`

Returns the **maximum** of `x` and the other arguments supplied: the single greatest value contained by all of them. All of the arguments must be the same type as `x`, and the return type will match that of `x`. If all of the arguments have a size of `0`, the return value will be `NULL`; note that this means that `max(x, max(y))` may produce an error, if `max(y)` is `NULL`, in cases where `max(x, y)` does not.

`(float$)mean(lif x)`

Returns the **arithmetic mean** of `x`: the sum of `x` divided by the number of values in `x`. If `x` has a size of `0`, the return value will be `NULL`. The unusual parameter type signature `lif` indicates that `x` can be `logical`, `integer`, or `float`; if `x` is `logical`, it is coerced to `integer` internally (with `F` being `0` and `T` being `1`, as always), allowing `mean()` to calculate the average truth value of a `logical` vector.

`(+$)min(+ x, ...)`

Returns the **minimum** of `x` and the other arguments supplied: the single smallest value contained by all of them. All of the arguments must be the same type as `x`, and the return type will match that of `x`. If all of the arguments have a size of `0`, the return value will be `NULL`; note that this means that `min(x, min(y))` may produce an error, if `min(y)` is `NULL`, in cases where `min(x, y)` does not.

`(+)pmax(+ x, + y)`

Returns the **parallel maximum** of `x` and `y`: the element-wise maximum for each corresponding pair of elements in `x` and `y`. The type of `x` and `y` must match, and the returned value will have the same type. In one usage pattern the size of `x` and `y` match, in which case the returned value will have the same size. In the other usage pattern either `x` and `y` is a singleton, in which case the returned value will match the size of the non-singleton argument, and pairs of elements for comparison will be formed between the singleton's element and each of the elements in the non-singleton.

`(+)pmin(+ x, + y)`

Returns the **parallel minimum** of `x` and `y`: the element-wise minimum for each corresponding pair of elements in `x` and `y`. The type of `x` and `y` must match, and the returned value will have the same type. In one usage pattern the size of `x` and `y` match, in which case the returned value will have the same size. In the other usage pattern either `x` and `y` is a singleton, in which case the returned value will match the size of the non-singleton argument, and pairs of elements for comparison will be formed between the singleton's element and each of the elements in the non-singleton.

`(numeric)range(numeric x, ...)`

Returns the **range** of `x` and the other arguments supplied: a vector of length `2` composed of the minimum and maximum values contained by all of them, at indices `0` and `1` respectively. All of the arguments must be the same type as `x`, and the return type will match that of `x`. If all of the arguments have a size of `0`, the return value will be `NULL`; note that this means that `range(x, range(y))` may produce an error, if `range(y)` is `NULL`, in cases where `range(x, y)` does not.

`(float$)sd(numeric x)`

Returns the **corrected sample standard deviation** of `x`. If `x` has a size of `0` or `1`, the return value will be `NULL`.

`(float$)ttest(float x, [Nf y = NULL], [Nf$ mu = NULL])`

Returns the ***p*-value resulting from running a *t*-test** with the supplied data. Two types of *t*-tests can be performed. If `x` and `y` are supplied (i.e., `y` is non-`NULL`), a two-sample unpaired two-sided Welch's *t*-test is conducted using the samples in `x` and `y`, each of which must contain at least two elements. The null hypothesis for this test is that the two samples are drawn from populations with the same mean. Other options, such as pooled-variance *t*-tests, paired *t*-tests, and one-sided *t*-tests, are not presently available. If `x` and `mu` are supplied (i.e., `mu` is non-`NULL`), a one-sample *t*-test is conducted in which the null hypothesis is that the sample is drawn from a population with mean `mu`.

Note that the results from this function are substantially different from those produced by R. The Eidos `ttest()` function uses uncorrected sample statistics, which means they will be biased for small sample sizes, whereas R probably uses corrected, unbiased sample statistics. This is an Eidos bug, and might be fixed if anyone complains. If large sample sizes are used, however, the bias is likely to be small, and uncorrected statistics are simpler and faster to compute.

`(float$)var(numeric x)`

Returns the **corrected sample variance** of `x`. If `x` has a size of `0` or `1`, the return value will be `NULL`. This is the square of the standard deviation calculated by `sd()`. At present it is illegal to call `var()` with a matrix or array argument, because the desired behavior in that case has not yet been implemented.

3.3 Distribution drawing and density functions

These functions for drawing from various distributions, and for calculating probability density function values for various distributions, are all based upon algorithms in the GNU Scientific Library (GSL). The distributions supported are those which are presently expected to be useful for users of Eidos; adding further distributions to Eidos on request is trivial as long as they are supported by the GSL, so feel free to make requests.

Draws are obtained using the standard Eidos random number generator, which might be shared with the Context; generating draws may therefore change the state of the Context. (In SLiM, for example, generating draws will alter the future of your simulation, since the random number sequence used by SLiM will be changed as a side effect.)

`(float)dmvnorm(float x, numeric mu, numeric sigma)`

Returns a vector of **probability densities for a *k*-dimensional multivariate normal distribution** with a length *k* mean vector `mu` and a *k* × *k* variance-covariance matrix `sigma`. The `mu` and `sigma` parameters are used for all densities. The quantile values, `x`, should be supplied as a matrix with one row per vector of quantile values and *k* columns (one column per dimension); for convenience, a single quantile may be supplied as a vector rather than a matrix with just one row. The number of dimensions *k* must be at least two; for *k*=1, use `dnorm()`.

Cholesky decomposition of the variance-covariance matrix `sigma` is involved as an internal step, and this requires that `sigma` be positive-definite; if it is not, an error will result. When more than one density is needed, it is much more efficient to call `dmvnorm()` once to generate all of the densities, since the Cholesky decomposition of `sigma` can then be done just once.

`(float)dnorm(float x, [numeric mean = 0], [numeric sd = 1])`

Returns a vector of **probability densities for a normal distribution** at quantiles `x` with mean `mean` and standard deviation `sd`. The `mean` and `sd` parameters may either be singletons, specifying a single value to be used for all of the densities, or they may be vectors of the same length as `x`, specifying a value for each density computation.

`(float)pnorm(float q, [numeric mean = 0], [numeric sd = 1])`

Returns a vector of **cumulative distribution function values for a normal distribution** at quantiles `q` with mean `mean` and standard deviation `sd`. The `mean` and `sd` parameters may either be singletons, specifying a single value to be used for all of the quantiles, or they may be vectors of the same length as `q`, specifying a value for each quantile.

`(float)rbeta(integer$ n, numeric alpha, numeric beta)`

Returns a vector of `n` **random draws from a beta distribution** with parameters `alpha` and `beta`. The `alpha` and `beta` parameters may either be singletons, specifying a single value to be used for all of the draws, or they may be vectors of length `n`, specifying a value for each draw. Draws are made from a beta distribution with probability density $P(s \mid \alpha,\beta) = [\Gamma(\alpha+\beta)/\Gamma(\alpha)\Gamma(\beta)]s^{\alpha-1}(1-s)^{\beta-1}$, where α is `alpha` and β is `beta`. Both parameters must be greater than `0`. The values drawn are in the interval [0, 1].

`(integer)rbinom(integer$ n, integer size, float prob)`

Returns a vector of `n` **random draws from a binomial distribution** with a number of trials specified by `size` and a probability of success specified by `prob`. The `size` and `prob` parameters may either be singletons, specifying a single value to be used for all of the draws, or they may be vectors of length `n`, specifying a value for each draw.

`(float)rcauchy(integer$ n, [numeric location = 0], [numeric scale = 1])`

Returns a vector of `n` **random draws from a Cauchy distribution** with location `location` and scale `scale`. The `location` and `scale` parameters may either be singletons, specifying a single value to be used for all of the draws, or they may be vectors of length `n`, specifying a value for each draw.

`(integer)rdunif(integer$ n, [integer min = 0], [integer max = 1])`

Returns a vector of `n` **random draws from a discrete uniform distribution** from `min` to `max`, inclusive. The `min` and `max` parameters may either be singletons, specifying a single value to be used for all of the draws, or they may be vectors of length `n`, specifying a value for each draw. See `runif()` for draws from a continuous uniform distribution.

`(float)rexp(integer$ n, [numeric mu = 1])`

Returns a vector of `n` **random draws from an exponential distribution** with mean `mu` (i.e. rate `1/mu`). The `mu` parameter may either be a singleton, specifying a single value to be used for all of the draws, or it may be a vector of length `n`, specifying a value for each draw.

`(float)rgamma(integer$ n, numeric mean, numeric shape)`

Returns a vector of `n` **random draws from a gamma distribution** with mean `mean` and shape parameter `shape`. The `mean` and `shape` parameters may either be singletons, specifying a single value to be used for all of the draws, or they may be vectors of length `n`, specifying a value for each draw. Draws are made from a gamma distribution with probability density $P(s \mid \alpha,\beta) = [\Gamma(\alpha)\beta^{\alpha}]^{-1}s^{\alpha-1}\exp(-s/\beta)$, where α is the shape parameter `shape`, and the mean of the distribution given by `mean` is equal to $\alpha\beta$. Values of `mean` less than zero are allowed, and are equivalent (in principle) to the negation of a draw from a gamma distribution with the same `shape` parameter and the negation of the `mean` parameter.

`(integer)rgeom(integer$ n, float p)`

Returns a vector of `n` **random draws from a geometric distribution** with parameter `p`. The `p` parameter may either be a singleton, specifying a single value to be used for all of the draws, or it may be a vector of length `n`, specifying a value for each draw. Eidos follows R in using the geometric distribution with support on the set {0, 1, 2, …}, where the drawn value indicates the number of failures prior to success. There is an alternative definition, based upon the number of trial required to get one success, so beware.

`(float)rlnorm(integer$ n, [numeric meanlog = 0], [numeric sdlog = 1])`

Returns a vector of `n` **random draws from a lognormal distribution** with mean `meanlog` and standard deviation `sdlog`, specified on the log scale. The `meanlog` and `sdlog` parameters may either be singletons, specifying a single value to be used for all of the draws, or they may be vectors of length `n`, specifying a value for each draw.

`(float)rmvnorm(integer$ n, numeric mu, numeric sigma)`

Returns a matrix of `n` **random draws from a *k*-dimensional multivariate normal distribution** with a length k mean vector `mu` and a $k \times k$ variance-covariance matrix `sigma`. The `mu` and `sigma` parameters are used for all `n` draws. The draws are returned as a matrix with `n` rows (one row per draw) and k columns (one column per dimension). The number of dimensions k must be at least two; for k=1, use `rnorm()`.

Cholesky decomposition of the variance-covariance matrix `sigma` is involved as an internal step, and this requires that `sigma` be positive-definite; if it is not, an error will result. When more than one draw is needed, it is much more efficient to call `rmvnorm()` once to generate all of the draws, since the Cholesky decomposition of `sigma` can then be done just once.

`(float)rnorm(integer$ n, [numeric mean = 0], [numeric sd = 1])`

Returns a vector of `n` **random draws from a normal distribution** with mean `mean` and standard deviation `sd`. The `mean` and `sd` parameters may either be singletons, specifying a single value to be used for all of the draws, or they may be vectors of length `n`, specifying a value for each draw.

`(integer)rpois(integer$ n, numeric lambda)`

Returns a vector of `n` **random draws from a Poisson distribution** with parameter `lambda` (not to be confused with the language concept of a "lambda"; `lambda` here is just the name of a parameter, because the symbol typically used for the parameter of a Poisson distribution is the Greek letter λ). The `lambda` parameter may either be a singleton, specifying a single value to be used for all of the draws, or it may be a vector of length `n`, specifying a value for each draw.

`(float)runif(integer$ n, [numeric min = 0], [numeric max = 1])`

Returns a vector of `n` **random draws from a continuous uniform distribution** from `min` to `max`, inclusive. The `min` and `max` parameters may either be singletons, specifying a single value to be used

for all of the draws, or they may be vectors of length `n`, specifying a value for each draw. See `rdunif()` for draws from a discrete uniform distribution.

`(float)rweibull(integer$ n, numeric lambda, numeric k)`

Returns a vector of `n` **random draws from a Weibull distribution** with scale parameter `lambda` and shape parameter `k`, both greater than zero. The `lambda` and `k` parameters may either be singletons, specifying a single value to be used for all of the draws, or they may be vectors of length `n`, specifying a value for each draw. Draws are made from a Weibull distribution with probability distribution $P(s \mid \lambda,k) = (k / \lambda^k)\, s^{k-1} \exp(-(s/\lambda)^k)$.

3.4 Vector construction functions

`(*)c(...)`

Returns the **concatenation** of all of its parameters into a single vector, stripped of all matrix/array dimensions (see `rbind()` and `cbind()` for concatenation that does not strip this information). The parameters will be promoted to the highest type represented among them, and that type will be the return type. `NULL` values are ignored; they have no effect on the result.

`(float)float(integer$ length)`

Returns a **new `float` vector** of the length specified by `length`, filled with `0.0` values. This can be useful for pre-allocating a vector which you then fill with values by subscripting.

```
(integer)integer(integer$ length, [integer$ fill1 = 0], [integer$ fill2 = 1],
    [Ni fill2Indices = NULL])
```

Returns a **new `integer` vector** of the length specified by `length`, filled with `0` values by default. This can be useful for pre-allocating a vector which you then fill with values by subscripting.

If a value is supplied for `fill1`, the new vector will be filled with that value instead of the default of `0`. Additionally, if a non-`NULL` vector is supplied for `fill2Indices`, the indices specified by `fill2Indices` will be filled with the value provided by `fill2`. For example, given the default values of `0` and `1` for `fill1` and `fill2`, the returned vector will contain `1` at all positions specified by `fill2Indices`, and will contain `0` at all other positions.

`(logical)logical(integer$ length)`

Returns a **new `logical` vector** of the length specified by `length`, filled with `F` values. This can be useful for pre-allocating a vector which you then fill with values by subscripting.

`(object<undefined>)object(void)`

Returns a **new empty `object` vector**. Unlike `float()`, `integer()`, `logical()`, and `string()`, a length cannot be specified and the new vector contains no elements. This is because there is no default value for the object type. Adding to such a vector is typically done with `c()`. Note that the return value is of type `object<undefined>`; this method creates an `object` vector that does not know what element type it contains. Such `object` vectors may be mixed freely with other `object` vectors in `c()` and similar contexts; the result of such mixing will take its `object`-element type from the `object` vector with a defined `object`-element type (if any).

`(*)rep(* x, integer$ count)`

Returns the **repetition** of `x`: the entirety of `x` is repeated `count` times. The return type matches the type of `x`.

`(*)repEach(* x, integer count)`

Returns the **repetition of elements** of `x`: each element of `x` is repeated. If `count` is a singleton, it specifies the number of times that each element of `x` will be repeated. Otherwise, the length of `count` must be equal to the length of `x`; in this case, each element of `x` is repeated a number of times specified by the corresponding value of `count`.

`(*)sample(* x, integer$ size, [logical$ replace = F], [Nif weights = NULL])`

Returns a vector of `size` containing a **sample from the elements of `x`**. If `replace` is `T`, sampling is conducted with replacement (the same element may be drawn more than once); if it is `F` (the default), then sampling is done without replacement. A vector of weights may be supplied in the optional parameter `weights`; if not `NULL`, it must be equal in size to `x`, all weights must be non-negative, and the sum of the weights must be greater than `0`. If `weights` is `NULL` (the default), then equal weights are used for all elements of `x`. An error occurs if `sample()` runs out of viable elements from which to draw; most notably, if sampling is done without replacement then `size` must be at most equal to the size of `x`, but if weights of zero are supplied then the restriction on `size` will be even more stringent. The draws are obtained from the standard Eidos random number generator, which might be shared with the Context.

`(numeric)seq(numeric$ from, numeric$ to, [Nif$ by = NULL], [Ni$ length = NULL])`

Returns a **sequence**, starting at `from` and proceeding in the direction of `to` until the next value in the sequence would fall beyond `to`. If the optional parameters `by` and `length` are both `NULL` (the default), the sequence steps by values of `1` or `-1` (as needed to proceed in the direction of `to`). A different step value may be supplied with `by`, but must have the necessary sign. Alternatively, a sequence length may be supplied in `length`, in which case the step magnitude will be chosen to produce a sequence of the requested length (with the necessary sign, as before). If `from` and `to` are both `integer` then the return type will be `integer` when possible (but this may not be possible, depending upon values supplied for `by` or `length`), otherwise it will be `float`.

`(integer)seqAlong(* x)`

Returns an **index sequence along `x`**, from `0` to `size(x) - 1`, with a step of `1`. This is a convenience function for easily obtaining a set of indices to address or iterate through a vector. Any matrix/array dimension information is ignored; the index sequence is suitable for indexing into `x` as a vector.

`(integer)seqLen(integer$ length)`

Returns an **index sequence of `length`**, from `0` to `length - 1`, with a step of `1`; if `length` is `0` the sequence will be zero-length. This is a convenience function for easily obtaining a set of indices to address or iterate through a vector. Note that when `length` is `0`, using the sequence operator with `0:(length-1)` will produce `0 -1`, and calling `seq(a, b, length=length)` will raise an error, but `seqLen(length)` will return `integer(0)`, making `seqLen()` particularly useful for generating a sequence of a given length that might be zero.

`(string)string(integer$ length)`

Returns a **new `string` vector** of the length specified by `length`, filled with `""` values. This can be useful for pre-allocating a vector which you then fill with values by subscripting.

3.5 Value inspection & manipulation functions

`(logical$)all(logical x, ...)`

Returns `T` if **all values are `T`** in `x` and in any other arguments supplied; if any value is `F`, returns `F`. All arguments must be of `logical` type. If all arguments are zero-length, `T` is returned.

`(logical$)any(logical x, ...)`

Returns `T` if **any value is `T`** in `x` or in any other arguments supplied; if all values are `F`, returns `F`. All arguments must be of `logical` type. If all arguments are zero-length, `F` is returned.

`(void)cat(* x, [string$ sep = " "])`

Concatenates output to Eidos's output stream, joined together by `sep`. The value `x` that is output may be of any type. A newline is not appended to the output, unlike the behavior of `print()`; if a trailing newline is desired, you can use `"\n"` (or use the `catn()` function). Also unlike `print()`, `cat()` tends to emit very literal output; `print(logical(0))` will emit "`logical(0)`", for example – showing a

semantic interpretation of the value – whereas `cat(logical(0))` will emit nothing at all, since there are no elements in the value (it is zero-length). Similarly, `print(NULL)` will emit "`NULL`", but `cat(NULL)` will emit nothing.

`(void)catn([* x = ""], [string$ sep = " "])`

Concatenates output (with a trailing newline) to Eidos's output stream, joined together by `sep`. The behavior of `catn()` is identical to that of `cat()`, except that a final newline character is appended to the output for convenience. For `catn()` a default value of `""` is supplied for `x`, to allow a simple `catn()` call with no parameters to emit a newline.

`(string)format(string$ format, numeric x)`

Returns a vector of **formatted strings** generated from `x`, based upon the formatting string `format`. The `format` parameter may be any `string` value, but must contain exactly one escape sequence beginning with the `%` character. This escape sequence specifies how to format a single value from the vector `x`. The returned vector contains one `string` value for each element of `x`; each `string` value is identical to the string supplied in `format`, except with a formatted version of the corresponding value from `x` substituted in place of the escape sequence.

The syntax for `format` is a subset of the standard C/C++ `printf()`-style format strings (available in many places online, such as http://en.cppreference.com/w/c/io/fprintf). The escape sequence used to format each value of x is composed of several elements:

- A `%` character at the beginning, initiating the escape sequence (if an actual `%` character is desired, rather than an escape sequence, `%%` may be used)
- Optional flags that modify the style of formatting:
  - `-` : The value is left-justified with the field (as opposed to the default of right-justification).
  - `+` : The sign of the value is always prepended, even if the value is positive (as opposed to the default of appending the sign only if the value is negative).
  - *space* : The value is prepended by a space when a sign is not prepended. This is ignored if the `+` flag is present, since values are then always prepended by a sign.
  - `#` : An alternative format is used. For `%o`, at least one leading zero is always produced. For `%x` and `%X`, `0x` or `0X` (respectively) is prepended if the value is nonzero. For `%f`, `%F`, `%e`, `%E`, `%g`, and `%G`, a decimal point is forced even if no zeros follow.
  - `0` : The value is left-justified with the field (as opposed to the default of right-justification)
- An optional minimum field width, specified as an integer value. Fields will be padded out to this minimum width. Padding will be done with space characters by default (or with zeros, if the `0` flag is used), on the left by default (or on the right, if the `-` flag is used).
- An optional precision, given as an integer value preceded by a `.` character. If no integer value follows the `.` character, a precision of zero will be used. For integer values of `x` (formatted with `%d`, `%i`, `%o`, `%x`, or `%X`) the precision specifies the minimum number of digits that will appear (with extra zeros on the left if necessary), with a default precision of `1`. For float values of `x` formatted with `%f`, `%F`, `%e`, `%E`, `%g`, or `%G`, the precision specifies the minimum number of digits that will appear to the right of the decimal point (with extra zeros on the right if necessary), with a default precision of `6`.
- A format specifier. For `integer` values, this may be `%d` or `%i` (producing base-10 output; there is no difference between the two), `%o` (producing base-8 or octal output), `%x` (producing base-16 hexadecimal output using lowercase letters), or `%X` (producing base-16 hexadecimal output using uppercase letters). For `float` values, this may be `%f` or `%F` to produce decimal notation (of the form `[-]ddd.ddd`; there is no difference between the two), `%e` or `%E` to produce scientific notation (of the form `[-]d.ddde±dd` or `[-]d.dddE±dd`, respectively), or `%g` or `%G` to produce either decimal notation or scientific notation (using the formatting of `%f` / `%e` or `%F` / `%E`, respectively) on a per-value basis, depending upon the range of the value.

Note that relative to the standard C/C++ `printf()`-style behavior, there are a few differences: (1) only a single escape sequence may be present in the format string, (2) the use of `*` to defer field width and precision values to a passed parameter is not supported, (3) only `integer` and `float` values of `x` are supported, (4) only the `%d`, `%i`, `%o`, `%x`, `%X`, `%f`, `%F`, `%e`, `%E`, `%g`, and `%G` format specifiers are supported, and (5) no length modifiers may be supplied, since Eidos does not support different sizes of the `integer` and `float` types. Note also that the Eidos conventions of emitting `INF` and `NAN` for infinities and Not-A-Number values respectively is not honored by this function; the strings generated for such values are platform-dependent, following the implementation definition of the C++ compiler used to build Eidos, since `format()` calls through to `snprintf()` to assemble the final string values.

For example, `format("A number: %+7.2f", c(-4.1, 15.375, 8))` will produce a vector with three elements: `"A number:  -4.10" "A number:  +15.38" "A number:  +8.00"`. The precision of `.2` results in two digits after the decimal point, the minimum field width of `7` results in padding of the values on the left (with spaces) to a minimum of seven characters, the flag `+` causes a sign to be shown on positive values as well as negative values, and the format specifier `f` leads to the `float` values of `x` being formatted in base-10 decimal. One `string` value is produced in the result vector for each value in the parameter `x`. These values could then be merged into a single string with `paste()`, for example, or printed with `print()` or `cat()`.

`(logical$)identical(* x, * y)`

Returns a `logical` value indicating **whether two values are identical**. If `x` and `y` have exactly the same type and size, and all of their corresponding elements are exactly the same, and (for matrices and arrays) their dimensions are identical, this will return `T`, otherwise it will return `F`. The test here is for *exact* equality; an `integer` value of `1` is not considered identical to a `float` value of `1.0`, for example. Elements in `object` values must be literally the same element, not simply identical in all of their properties. Type promotion is never done. For testing whether two values are the same, this is generally preferable to the use of operator == or operator `!=`; see the discussion at section 2.5.1. Note that `identical(NULL,NULL)` is `T`.

`(*)ifelse(logical test, * trueValues, * falseValues)`

Returns the result of a **vector conditional** operation: a vector composed of values from `trueValues`, for indices where `test` is `T`, and values from `falseValues`, for indices where `test` is `F`. The lengths of `trueValues` and `falseValues` must either be equal to `1` or to the length of `test`; however, `trueValues` and `falseValues` don't need to be the same length as each other. Furthermore, the type of `trueValues` and `falseValues` must be the same (including, if they are `object` type, their element type). The return will be of the same length as `test`, and of the same type as `trueValues` and `falseValues`. Each element of the return vector will be taken from the corresponding element of `trueValues` if the corresponding element of `test` is `T`, or from the corresponding element of `falseValues` if the corresponding element of `test` is `F`; if the vector from which the value is to be taken (i.e., `trueValues` or `falseValues`) has a length of `1`, that single value is used repeatedly, recycling the vector. If `test`, `trueValues`, and/or `falseValues` are matrices or arrays, that will be ignored by `ifelse()` *except* that the result will be of the same dimensionality as `test`.

This is quite similar to a function in R of the same name; note, however, that Eidos evaluates all arguments to functions calls immediately, so `trueValues` and `falseValues` will be evaluated fully regardless of the values in `test`, unlike in R. Value expressions without side effects are therefore recommended.

`(integer$)length(* x)`

Returns the **size** (e.g., length) of `x`: the number of elements contained in `x`. Note that `length()` is a synonym for `size()`.

`(integer)match(* x, * table)`

Returns a vector of the **positions of (first) matches** of `x` in `table`. Type promotion is not performed; `x` and `table` must be of the same type. For each element of `x`, the corresponding element in the result will give the position of the first match for that element of `x` in `table`; if the element has no match in

`table`, the element in the result vector will be `-1`. The result is therefore a vector of the same length as `x`. If a `logical` result is desired, with `T` indicating that a match was found for the corresponding element of `x`, use `(match(x, table) >= 0)`.

`(integer)nchar(string x)`

Returns a vector of the **number of characters** in the string-elements of `x`.

`(integer)order(+ x, [logical$ ascending = T])`

Returns a **vector of sorting indices** for `x`: a new `integer` vector of the same length as `x`, containing the indices into `x` that would sort `x`. In other words, `x[order(x)]==sort(x)`. This can be useful for more complex sorting problems, such as sorting several vectors in parallel by a sort order determined by one of the vectors. If the optional `logical` parameter `ascending` is `T` (the default), then the sorted order will be ascending; if it is `F`, the sorted order will be descending. The ordering is determined according to the same logic as the `<` and `>` operators in Eidos. To easily sort vectors in a single step, use `sort()` or `sortBy()`, for non-`object` and `object` vectors respectively.

`(string$)paste(* x, [string$ sep = " "])`

Returns a **joined string** composed from the `string` representations of the elements of `x`, joined together by `sep`. Although this function is based upon the R function of the same name, note that it is much simpler and less powerful; in particular, only the elements of a single vector may be joined, rather than the var-args functionality of the R `paste()`. The string representation used by `paste()` is the same as that emitted by `cat()`.

`(string$)paste0(* x)`

Returns a **joined string** composed from the `string` representations of the elements of `x`, joined together with no separator. This function is identical to `paste()`, except that no separator is used.

`(void)print(* x)`

Prints output to Eidos's output stream. The value `x` that is output may be of any type. A newline is appended to the output. See `cat()` for a discussion of the differences between `print()` and `cat()`.

`(*)rev(* x)`

Returns the **reverse** of `x`: a new vector with the same elements as `x`, but in the opposite order.

`(integer$)size(* x)`

Returns the **size** of `x`: the number of elements contained in `x`. Note that `length()` is a synonym for `size()`.

`(+)sort(+ x, [logical$ ascending = T])`

Returns a **sorted copy** of `x`: a new vector with the same elements as `x`, but in sorted order. If the optional `logical` parameter `ascending` is `T` (the default), then the sorted order will be ascending; if it is `F`, the sorted order will be descending. The ordering is determined according to the same logic as the `<` and `>` operators in Eidos. To sort an `object` vector, use `sortBy()`. To obtain indices for sorting, use `order()`.

`(object)sortBy(object x, string$ property, [logical$ ascending = T])`

Returns a **sorted copy** of `x`: a new vector with the same elements as `x`, but in sorted order. If the optional `logical` parameter `ascending` is `T` (the default), then the sorted order will be ascending; if it is `F`, the sorted order will be descending. The ordering is determined according to the same logic as the `<` and `>` operators in Eidos. The `property` argument gives the name of the property within the elements of `x` according to which sorting should be done. This must be a simple property name; it cannot be a property path. For example, to sort a `Mutation` vector by the selection coefficients of the mutations, you would simply pass `"selectionCoeff"`, including the quotes, for `property`. To sort a non-`object` vector, use `sort()`. To obtain indices for sorting, use `order()`.

`(void)str(* x)`

Prints the structure of x: a summary of its type and the values it contains. If x is an `object`, note that `str()` produces different results from the `str()` method of x; the `str()` function prints the external structure of x (the fact that it is an object, and the number and type of its elements), whereas the `str()` method prints the internal structure of x (the external structure of all the properties contained by x).

`(string)strsplit(string$ x, [string$ sep = " "])`

Returns **substrings** of x that were separated by the separator string `sep`. Every substring defined by an occurrence of the separator is included, and thus zero-length substrings may be returned. For example, `strsplit(".foo..bar.", ".")` returns a string vector containing `""`, `"foo"`, `""`, `"bar"`, `""`. In that example, the empty string between `"foo"` and `"bar"` in the returned vector is present because there were two periods between `foo` and `bar` in the input string – the empty string is the substring between those two separators. If `sep` is `""`, a vector of single characters will be returned, resulting from splitting x at every position. Note that `paste()` performs the inverse operation of `strsplit()`.

`(string)substr(string x, integer first, [Ni last = NULL])`

Returns **substrings** extracted from the elements of `x,` spanning character position `first` to character position `last` (inclusive). Character positions are numbered from `0` to `nchar(x)-1`. Positions that fall outside of that range are legal; a substring range that encompasses no characters will produce an empty string. If `first` is greater than `last`, an empty string will also result. If `last` is NULL (the default), then the substring will extend to the end of the string. The parameters `first` and `last` may either be singletons, specifying a single value to be used for all of the substrings, or they may be vectors of the same length as x, specifying a value for each substring.

`(*)unique(* x, [logical$ preserveOrder = T])`

Returns the **unique values** in x. In other words, for each value k in x that occurs at least once, the vector returned will contain k exactly once. If `preserveOrder` is `T` (the default), the order of values in x is preserved, taking the first instance of each value; this is relatively slow, with $O(n^2)$ performance. If `preserveOrder` if `F` instead, the order of values in x is not preserved, and no particular ordering should be relied upon; this is relatively fast, with $O(n \log n)$ performance. This performance difference will only matter for large vectors, however; for most applications the default behavior can be retained whether the order of the result matters or not.

`(integer)which(logical x)`

Returns the **indices of `T` values** in x. In other words, if an index k in x is `T`, then the vector returned will contain k; if index k in x is `F`, the vector returned will omit k. One way to look at this is that it converts from a `logical` subsetting vector to an `integer` (index-based) subsetting vector, without changing which subset positions would be selected.

`(integer$)whichMax(+ x)`

Returns the **index of the (first) maximum value** in x. In other words, if k is equal to the maximum value in x, then the vector returned will contain the index of the first occurrence of k in x. If the maximum value is unique, the result is the same as (but more efficient than) the expression `which(x==max(x))`, which returns the indices of *all* of the occurrences of the maximum value in x.

`(integer$)whichMin(+ x)`

Returns the **index of the (first) minimum value** in x. In other words, if k is equal to the minimum value in x, then the vector returned will contain the index of the first occurrence of k in x. If the minimum value is unique, the result is the same as (but more efficient than) the expression `which(x==min(x))`, which returns the indices of *all* of the occurrences of the minimum value in x.

3.6 Value type testing and coercion functions

`(float)asFloat(+ x)`
Returns the **conversion to `float`** of `x`. If `x` is `string` and cannot be converted to `float`, Eidos will throw an error.

`(integer)asInteger(+ x)`
Returns the **conversion to `integer`** of `x`. If `x` is of type `string` or `float` and cannot be converted to `integer`, Eidos will throw an error.

`(logical)asLogical(+ x)`
Returns the **conversion to `logical`** of `x`. Recall that in Eidos the empty `string` `""` is considered `F`, and all other `string` values are considered `T`. Converting `INF` or `-INF` to `logical` yields `T` (since those values are not equal to zero); converting `NAN` to `logical` throws an error.

`(string)asString(+ x)`
Returns the **conversion to `string`** of `x`. Note that `asString(NULL)` returns `"NULL"` even though `NULL` is zero-length.

`(string$)elementType(* x)`
Returns the **element type** of `x`, as a `string`. For the non-`object` types, the element type is the same as the type: `"NULL"`, `"logical"`, `"integer"`, `"float"`, or `"string"`. For `object` type, however, `elementType()` returns the name of the type of element contained by the object, such as `"SLiMSim"` or `"Mutation"` in the Context of SLiM. Contrast this with `type()`.

`(logical$)isFloat(* x)`
Returns `T` if `x` **is `float` type**, `F` otherwise.

`(logical$)isInteger(* x)`
Returns `T` if `x` **is `integer` type**, `F` otherwise.

`(logical$)isLogical(* x)`
Returns `T` if `x` **is `logical` type**, `F` otherwise.

`(logical$)isNULL(* x)`
Returns `T` if `x` **is `NULL` type**, `F` otherwise.

`(logical$)isObject(* x)`
Returns `T` if `x` **is `object` type**, `F` otherwise.

`(logical$)isString(* x)`
Returns `T` if `x` **is `string` type**, `F` otherwise.

`(string$)type(* x)`
Returns the **type** of `x`, as a `string`: `"NULL"`, `"logical"`, `"integer"`, `"float"`, `"string"`, or `"object"`. Contrast this with `elementType()`.

3.7 Matrix and array functions

`(*)apply(* x, integer margin, string$ lambdaSource)`
Prior to Eidos 1.6 / SLiM 2.6, `sapply()` was named `apply()`, and this function did not yet exist
Applies a block of Eidos code to margins of `x`. This function is essentially an extension of `sapply()` for use with matrices and arrays; it is recommended that you fully understand `sapply()` before tackling this function. As with `sapply()`, the lambda specified by `lambdaSource` will be executed for subsets of `x`, and the results will be concatenated together with type-promotion in the style of `c()` to

produce a result. Unlike `sapply()`, however, the subsets of `x` used might be rows, columns, or higher-dimensional slices of `x`, rather than just single elements, depending upon the value of `margin`. For `apply()`, `x` must be a matrix or array (see section 2.9). The `apply()` function in Eidos is patterned directly after the `apply()` function in R, and should behave identically, except that dimension indices in Eidos are zero-based whereas in R they are one-based.

The `margin` parameter gives the indices of dimensions of `x` that will be iterated over when assembling values to supply to lambdaSource. If `x` is a matrix it has two dimensions: rows, of dimension index `0`, and columns, of dimension index `1`. These are the indices of the dimension sizes returned by `dim()`; `dim(x)[0]` gives the number of rows of `x`, and `dim(x)[1]` gives the number of columns. These dimension indices are also apparent when subsetting `x`; a subset index in position `0`, such as `x[m,]`, gives row `m` of `x`, whereas a subset index in position `1`, such as `x[,n]`, gives column `n` of `x`. In the same manner, supplying `0` for `margin` specifies that subsets of `x` from `x[0,]` to `x[m,]` should be "passed" to `lambdaSource`, through the `applyValue` "parameter"; dimension `0` is iterated over, whereas dimension `1` is taken in aggregate since it is not included in `margin`. The final effect of this is that whole rows of `x` are passed to `lambdaSource` through `applyValue`. Similarly, `margin=1` would specify that subsets of `x` from `x[,0]` to `x[,n]` should be passed to `lambdaSource`, resulting in whole columns being passed. Specifying `margin=c(0,1)` would indicate that dimensions `0` and `1` should both be iterated over (dimension `0` more rapidly), so for a matrix each each individual value of `x` would be passed to `lambdaSource`. Specifying `margin=c(1,0)` would similarly iterate over both dimensions, but dimension `1` more rapidly; the traversal order would therefore be different, and the dimensionality of the result would also differ (see below). For higher-dimensional arrays dimension indices beyond `1` exist, and so `margin=c(0,1)` or `margin=c(1,0)` would provide slices of `x` to `lambdaSource`, each slice having a specific row and column index. Slices are generated by subsetting in the same way as operator `[]`, but additionally, redundant dimensions are dropped as by `drop()`.

The return value from `apply()` is built up from the type-promoted concatenated results, as if by the `c()` function, from the iterated execution of `lambdaSource`; the only question is what dimensional structure is imposed upon that vector of values. If the results from `lambdaSource` are not of a consistent length, or are of length zero, then the concatenated results are returned as a plain vector. If all results are of length `n > 1`, the return value is an array of dimensions `c(n, dim(x)[margin]);` in other words, each `n`-vector provides the lowest dimension of the result, and the sizes of the marginal dimensions are imposed upon the data above that. If all results are of length `n == 1`, then if a single margin was specified the result is a vector (of length equal to the size of that marginal dimension), or if more than one margin was specified the result is an array of dimension `dim(x)[margin]`; in other words, the sizes of the marginal dimensions are imposed upon the data. Since `apply()` iterates over the marginal dimensions in the same manner, these structures follows the structure of the data.

The above explanation may not be entirely clear, so let's look at an example. If `x` is a matrix with two rows and three columns, such as defined by `x = matrix(1:6, nrow=2);`, then executing `apply(x, 0, "sum(applyValue);");` would cause each row of `x` to be supplied to the lambda through `applyValue`, and the values in each row would thus be summed to produce `9 12` as a result. The call `apply(x, 1, "sum(applyValue);");` would instead sum columns of `x`, producing `3 7 11` as a result. Now consider using `range()` rather than `sum()` in the lambda, thus producing two values for each row or column. The call `apply(x, 0, "range(applyValue);");` produces a result of `matrix(c(1,5,2,6), nrow=2)`, with the range of the first row of `x`, 1–5, in the first column of the result, and the range of the second row of `x`, 2–6, in the second column. Although visualization becomes more difficult, these same patterns extend to higher dimensions and arbitrary margins of `x`.

`(*)array(* data, integer dim)`

Creates a new array from the data specified by `data`, with the dimension sizes specified by `dim`. The first dimension size in `dim` is the number of rows, and the second is the number of columns; further entries specify the sizes of higher-order dimensions. As many dimensions may be specified as desired, but with a minimum of two dimensions. An array with two dimensions is a matrix (by definition); note

that `matrix()` may provide a more convenient way to make a new matrix. Each dimension must be of size `1` or greater; `0`-size dimensions are not allowed.

The elements of `data` are used to populate the new array; the size of `data` must therefore be equal to the size of the new array, which is the product of all the values in `dim`. The new array will be filled in dimension order: one element in each row until a column is filled, then on to the next column in the same manner until all columns are filled, and then onward into the higher-order dimensions in the same manner.

`(*)cbind(...)`

Combines vectors or matrices by column to produce a single matrix. The parameters must be vectors (which are interpreted by `cbind()` as if they were one-column matrices) or matrices. They must be of the same type, of the same class if they are of type `object`, and have the same number of rows. If these conditions are met, the result is a single matrix with the parameters joined together, left to right. Parameters may instead be `NULL`, in which case they are ignored; or if all parameters are `NULL`, the result is `NULL`. A sequence of vectors, matrices, and `NULL`s may thus be concatenated with the `NULL` values removed, analogous to `c()`. Calling `cbind(x)` is an easy way to create a one-column matrix from a vector.

To combine vectors or matrices by row instead, see `rbind()`.

`(integer)dim(* x)`

Returns the dimensions of matrix or array `x`. The first dimension value is the number of rows, the second is the number of columns, and further values indicate the sizes of higher-order dimensions, identically to how dimensions are supplied to `array()`. `NULL` is returned if `x` is not a matrix or array.

`(*)drop(* x)`

Returns the result of dropping redundant dimensions from matrix or array `x`. Redundant dimensions are those with a size of exactly 1. Non-redundant dimensions are retained. If only one non-redundant dimension is present, the result is a vector; if more than one non-redundant dimension is present, the result will be a matrix or array. If `x` is not a matrix or array, it is returned unmodified.

`(*)matrix(* data, [Ni$ nrow = NULL], [Ni$ ncol = NULL], [logical$ byrow = F])`

Creates a new matrix from the data specified by `data`. By default this creates a one-column matrix. If non-`NULL` values are supplied for `nrow` and/or `ncol`, a matrix will be made with the requested number of rows and/or columns if possible; if the length of `data` is not compatible with the requested dimensions, an error will result. By default, values from data will populate the matrix by columns, filling each column sequentially before moving on to the next column; if `byrow` is `T` the matrix will be populated by rows instead.

`(numeric)matrixMult(numeric x, numeric y)`

Returns the result of matrix multiplication of `x` with `y`. In Eidos (as in R), with two matrices `A` and `B` the simple product `A * B` multiplies the corresponding elements of the matrices; in other words, if `X` is the result of `A * B`, then $X_{ij} = A_{ij} * B_{ij}$. This is parallel to the definition of other operators; A + B adds the corresponding elements of the matrices ($X_{ij} = A_{ij} + B_{ij}$), etc. In R, true matrix multiplication is achieved with a special operator, `%*%`; in Eidos, the `matrixMult()` function is used instead.

Both `x` and `y` must be matrices, and must be conformable according to the standard definition of matrix multiplication (i.e., if `x` is an $n \times m$ matrix then `y` must be a $m \times p$ matrix, and the result will be a $n \times p$ matrix). Vectors will not be promoted to matrices by this function, even if such promotion would lead to a conformable matrix.

`(integer$)ncol(* x)`

Returns the number of columns in matrix or array `x`. For vector `x`, `ncol()` returns `NULL`; `size()` should be used. An equivalent of R's `NCOL()` function, which treats vectors as `1`-column matrices, is not provided but would be trivial to implement as a user-defined function.

`(integer$)nrow(* x)`

Returns the number of rows in matrix or array `x`. For vector `x`, `nrow()` returns `NULL`; `size()` should be used. An equivalent of R's `NROW()` function, which treats vectors as `1`-column matrices, is not provided but would be trivial to implement as a user-defined function.

`(*)rbind(...)`

Combines vectors or matrices by row to produce a single matrix. The parameters must be vectors (which are interpreted by `rbind()` as if they were one-row matrices) or matrices. They must be of the same type, of the same class if they are of type `object`, and have the same number of columns. If these conditions are met, the result is a single matrix with the parameters joined together, top to bottom. Parameters may instead be `NULL`, in which case they are ignored; or if all parameters are `NULL`, the result is `NULL`. A sequence of vectors, matrices, and `NULL`s may thus be concatenated with the `NULL` values removed, analogous to `c()`. Calling `rbind(x)` is an easy way to create a one-row matrix from a vector.

To combine vectors or matrices by column instead, see `cbind()`.

`(*)t(* x)`

Returns the transpose of `x`, which must be a matrix. This is the matrix reflected across its diagonal; or alternatively, the matrix with its columns written out instead as rows in the same order.

3.8 Filesystem access functions

`(logical$)createDirectory(string$ path)`

Creates a new filesystem directory at the path specified by `path` and returns a `logical` value indicating if the creation succeeded (`T`) or failed (`F`). If the path already exists, `createDirectory()` will do nothing to the filesystem, will emit a warning, and will return `T` to indicate success if the existing path is a directory, or `F` to indicate failure if the existing path is not a directory.

`(logical$)deleteFile(string$ filePath)`

Deletes the file specified by `filePath` and returns a `logical` value indicating if the deletion succeeded (`T`) or failed (`F`).

`(logical$)fileExists(string$ filePath)`

Checks the existence of the file specified by `filePath` and returns a `logical` value indicating if it exists (`T`) or does not exist (`F`). This also works for directories.

`(string)filesAtPath(string$ path, [logical$ fullPaths = F])`

Returns a `string` vector containing the **names of all files in a directory** specified by `path`. If the optional parameter `fullPaths` is `T`, full filesystem paths are returned for each file; if `fullPaths` is `F` (the default), then only the filenames relative to the specified directory are returned. This list includes directories (i.e. subfolders), including the `"."` and `".."` directories on Un*x systems. The list also includes invisible files, such as those that begin with a `"."` on Un*x systems. This function does not descend recursively into subdirectories. If an error occurs during the read, `NULL` will be returned.

`(string$)getwd(void)`

Gets the current filesystem working directory. The filesystem working directory is the directory which will be used as a base path for relative filesystem paths. For example, if the working directory is `"~/Desktop"` (the `Desktop` subdirectory within the current user's home directory, as represented by `~`), then the filename `"foo.txt"` would correspond to the filesystem path `"~/Desktop/foo.txt"`, and the relative path `"bar/baz/"` would correspond to the filesystem path `"~/Desktop/bar/baz/"`.

Note that the path returned may not be identical to the path previously set with `setwd()`, if for example symbolic links are involved; but it ought to refer to the same actual directory in the filesystem.

The initial working directory is – as is generally the case on Un*x – simply the directory given to the running Eidos process by its parent process (the operating system, a shell, a job scheduler, a debugger, or whatever the case may be). If you launch Eidos (or SLiM) from the command line in a Un*x shell, it is typically the current directory in that shell. Before relative filesystem paths are used, you may therefore wish check what the initial working directory is on your platform, with `getwd()`, if you are not sure. Alternatively, you can simply use `setwd()` to set the working directory to a known path.

`(string)readFile(string$ filePath)`

Reads in the contents of a file specified by `filePath` and returns a `string` vector containing the lines (separated by `\n` and `\r` characters) of the file. Reading files other than text files is not presently supported. If an error occurs during the read, `NULL` will be returned.

`(string$)setwd(string$ path)`

Sets the current filesystem working directory. The filesystem working directory is the directory which will be used as a base path for relative filesystem paths (see `getwd()` for further discussion). An error will result if the working directory cannot be set to the given path.

The current working directory prior to the change will be returned as an invisible `string` value; the value returned is identical to the value that would have been returned by `getwd()`, apart from its invisibility.

See `getwd()` for discussion regarding the initial working directory, before it is set with `setwd()`.

`(logical$)writeFile(string$ filePath, string contents, [logical$ append = F])`

Writes or appends to a file specified by `filePath` with contents specified by `contents`, a `string` vector of lines. If `append` is `T`, the write will be appended to the existing file (if any) at `filePath`; if it is `F` (the default), then the write will replace an existing file at that path. If the write is successful, `T` will be returned; if not, `F` will be returned.

Note that newline characters will be added at the ends of the lines in `contents`. If you do not wish to have newlines added, you should use `paste()` to assemble the elements of `contents` together into a singleton `string`.

`(string$)writeTempFile(string$ prefix, string$ suffix, string contents)`

Writes to a unique temporary file with contents specified by `contents`, a `string` vector of lines. The filename used will begin with `prefix` and end with `suffix`, and will contain six random characters in between; for example, if `prefix` is `"plot1_"` and `suffix` is `".pdf"`, the generated filename might look like `"plot1_r5Mq0t.pdf"`. It is legal for `prefix`, `suffix`, or both to be the empty string, `""`, but supplying a file extension is usually advisable at minimum. The file will be created inside the `/tmp/` directory of the system, which is provided by Un*x systems as a standard location for temporary files; the `/tmp/` directory should not be specified as part of prefix (nor should any other directory information). The filename generated is guaranteed not to already exist in `/tmp/`. The file is created with Un*x permissions `0600`, allowing reading and writing only by the user for security. If the write is successful, the full path to the temporary file will be returned; if not, `""` will be returned.

Note that newline characters will be added at the ends of the lines in `contents`. If you do not wish to have newlines added, you should use `paste()` to assemble the elements of `contents` together into a singleton `string`.

3.9 Color manipulation functions

`(string)cmColors(integer$ n)`

Generate colors in a "cyan-magenta" color palette. The number of colors desired is passed in `n`, and the returned vector will contain `n` color strings. The color sequence begins with cyan, passes through white, and then ramps to magenta. See `rainbow()`, `heatColors()`, and `terrainColors()` for other color palettes.

`(float)color2rgb(string color)`

Converts a color string to RGB. The color string specified in `color` may be either a named color (see chapter 6) or a color in hexadecimal format such as `"#007FC0"`. The equivalent RGB color is returned as a `float` vector of length three (red, green, blue). Returned RGB values will be in the interval [0, 1].

This function can also be called with a non-singleton vector of color strings in `color`. In this case, the returned `float` value will be a matrix of RGB values, with three columns (red, green, blue) and one row per element of `color`.

`(string)heatColors(integer$ n)`

Generate colors in a "heat map" color palette. The number of colors desired is passed in `n`, and the returned vector will contain `n` color strings. The color sequence begins with red, passes through orange to yellow, and then fades up to white. See `rainbow()`, `cmColors()`, and `terrainColors()` for other color palettes.

`(float)hsv2rgb(float hsv)`

Converts an HSV color to RGB. The HSV color is specified in `hsv` as a `float` vector of length three (hue, saturation, value), and the equivalent RGB color is returned as a `float` vector of length three (red, green, blue). HSV values will be clamped to the interval [0, 1], and returned RGB values will also be in the interval [0, 1].

This function can also be called with a matrix of HSV values, with three columns (hue, saturation, value). In this case, the returned `float` value will be a matrix of RGB values, with three columns (red, green, blue) and one row per row of `hsv`.

```
(string)rainbow(integer$ n, [float$ s = 1], [float$ v = 1], [float$ start = 0],
    [Nf$ end = NULL], [logical$ ccw = T])
```

Generate colors in a "rainbow" color palette. The number of colors desired is passed in `n`, and the returned vector will contain `n` color strings. Parameters `s` and `v` control the saturation and value of the rainbow colors generated. The color sequence begins with the hue `start`, and ramps to the hue `end`, in a counter-clockwise direction around the standard HSV color wheel if `ccw` is `T` (the default, following R), otherwise in a clockwise direction. If `end` is `NULL` (the default), a value of `(n-1)/n` is used, producing a complete rainbow around the color wheel when `start` is also the default value of `0`. See `cmColors()`, `heatColors()`, and `terrainColors()` for other color palettes.

`(string)rgb2color(float rgb)`

Converts an RGB color to a color string. The RGB color is specified in `rgb` as a `float` vector of length three (red, green, blue). The equivalent color string is returned as singleton `string` specifying the color in the format `"#RRGGBB"`, such as `"#007FC0"`. RGB values will be clamped to the interval [0, 1].

This function can also be called with a matrix of RGB values, with three columns (red, green, blue). In this case, the returned `string` value will be a vector of color strings, with one element per row of `rgb`.

`(float)rgb2hsv(float rgb)`

Converts an RGB color to HSV. The RGB color is specified in `rgb` as a `float` vector of length three (red, green, blue), and the equivalent HSV color is returned as a `float` vector of length three (hue, saturation, value). RGB values will be clamped to the interval [0, 1], and returned HSV values will also be in the interval [0, 1].

This function can also be called with a matrix of RGB values, with three columns (red, green, blue). In this case, the returned `float` value will be a matrix of HSV values, with three columns (hue, saturation, value) and one row per row of `rgb`.

`(string)terrainColors(integer$ n)`

Generate colors in a "terrain" color palette. The number of colors desired is passed in `n`, and the returned vector will contain `n` color strings. The color sequence begins with forest green, passes

through dark orange, and then ramps to off-white. See `rainbow()`, `cmColors()`, and `heatColors()` for other color palettes.

3.10 Miscellaneous functions

`(void)beep([Ns$ soundName = NULL])`

Plays a sound or beeps. On Mac OS X in a GUI environment (i.e., in EidosScribe or SLiMgui), the optional parameter `soundName` can be the name of a sound file to play; in other cases (if `soundName` is `NULL`, or at the command line, or on platforms other than OS X) `soundName` is ignored and a standard system beep is played.

When `soundName` is not `NULL`, a sound file in a supported format (such as `.aiff` or `.mp3`) is searched for sequentially in four standard locations, in this order: `~/Library/Sounds`, `/Library/Sounds`, `/Network/Library/Sounds`, and finally `/System/Library/Sounds`. Standard OS X sounds located in `/System/Library/Sounds` include `"Basso"`, `"Blow"`, `"Bottle"`, `"Frog"`, `"Funk"`, `"Glass"`, `"Hero"`, `"Morse"`, `"Ping"`, `"Pop"`, `"Purr"`, `"Sosumi"`, `"Submarine"`, and `"Tink"`. Do not include the file extension, such as `.aiff` or `.mp3`, in `soundName`.

CAUTION: When not running in EidosScribe or SLiMgui, it is often the case that the only simple means available to play a beep is to send a `BEL` character (ASCII 7) to the standard output. Unfortunately, when this is the case, it means that (1) no beep will be audible if output is being redirected into a file, and (2) a control character, `^G`, will occur in the output at the point when the beep was requested. It is therefore recommended that `beep()` be used only when doing interactive work in a terminal shell (or in a GUI, on OS X), not when producing output files. However, this issue is platform-specific; on some platforms `beep()` may result in a beep, and no emitted `^G`, even when output is redirected. When a `^G` must be emitted to the standard output to generate the beep, a warning message will also be emitted to make any associated problems easier to diagnose.

`(void)citation(void)`

Prints citation information for Eidos to Eidos's output stream.

`(float$)clock([string$ type = "cpu"])`

Returns the value of a **system clock**. If `type` is `"cpu"`, this returns the current value of the CPU usage clock. This is the amount of CPU time used by the current process, in seconds; it is unrelated to the current time of day (for that, see the `time()` function). This is useful mainly for determining how much processor time a given section of code takes; `clock()` can be called before and after a block of code, and the end clock minus the start clock gives the elapsed CPU time consumed in the execution of the block of code. See also the `timed` parameter of `executeLambda()`, which automates this procedure. Note that if multiple cores are utilized by the process, the CPU usage clock will be the sum of the CPU usage across all cores, and may therefore run faster than the wall clock.

If `type` is `"mono"`, this returns the value of the system's monotonic clock. This represents user-perceived ("wall clock") elapsed time from some arbitrary timebase (which will not change during the execution of the program), but it will not jump if the time zone or the wall clock time are changed for the system. This clock is useful for measuring user-perceived elapsed time, as described above, and may provide a more useful metric for performance than CPU time if multiple cores are being utilized.

`(string$)date(void)`

Returns a **standard date string** for the current date in the local time of the executing machine. The format is `%d-%m-%Y` (day in two digits, then month in two digits, then year in four digits, zero-padded and separated by dashes) regardless of the localization of the executing machine, for predictability and consistency.

`(void)defineConstant(string$ symbol, + value)`

Defines a new constant with the name `symbol` and the value specified by `value`. The value may be any non-object value at all. The name cannot previously be defined in any way (i.e., as either a

variable or a constant). The defined constant acts identically to intrinsic Eidos constants such as T, NAN, and PI, and will remain defined for as long as the Eidos context lives even if it is defined inside a block being executed by executeLambda(), apply(), sapply(), or a Context-defined script block.

(vNlifso)doCall(string$ functionName, ...)

Returns the results from a **call to a specified function**. The function named by the parameter functionName is called, and the remaining parameters to doCall() are forwarded on to that function verbatim. This can be useful for calling one of a set of similar functions, such as sin(), cos(), etc., to perform a math function determined at runtime, or one of the as...() family of functions to convert to a type determined at runtime. Note that named arguments and default arguments, beyond the functionName argument, are not supported by doCall(); all arguments to the target function must be specified explicitly, without names.

(vNlifso)executeLambda(string$ lambdaSource, [ls$ timed = F])

Executes a block of Eidos code defined by lambdaSource. Eidos allows you to execute *lambdas*: blocks of Eidos code which can be called directly within the same scope as the caller. Eidos lambdas do not take arguments; for this reason, they are not first-class functions. (Since they share the scope of the caller, however, you may effectively pass values in and out of a lambda using global variables.) The string argument lambdaSource may contain one or many Eidos statements as a single string value. Lambdas are represented, to the caller, only as the source code string lambdaSource; the executable code is not made available programmatically. If an error occurs during the tokenization, parsing, or execution of the lambda, that error is raised as usual; executing code inside a lambda does not provide any additional protection against exceptions raised. The return value produced by the code in the lambda is returned by executeLambda(). If the optional parameter timed is T, the total (CPU clock) execution time for the lambda will be printed after the lambda has completed (see clock()); if it is F (the default), no timing information will be printed. The timed parameter may also be "cpu" or "mono" to specifically request timing with the CPU clock (which will count the usage across all cores, and may thus run faster than wall clock time if multiple cores are being utilized) or the monotonic clock (which will correspond, more or less, to elapsed wall clock time regardless of multithreading); see the documentation for clock() for further discussion of these timing options.

The current implementation of executeLambda() caches a tokenized and parsed version of lambdaSource, so calling executeLambda() repeatedly on a single source string is much more efficient than calling executeLambda() with a newly constructed string each time. If you can use a string literal for lambdaSource, or reuse a constructed source string stored in a variable, that will improve performance considerably.

(logical)exists(string symbol)

Returns a logical vector indicating **whether symbols exist**. If a symbol has been defined as an intrinsic Eidos constant like T, INF, and PI, or as a Context-defined constant like sim in SLiM, or as a user-defined constant using defineConstant(), or as a variable by assignment, this function returns T. Otherwise, the symbol has not been defined, and exists() returns F. This is commonly used to check whether a user-defined constant already exists, with the intention of defining the constant if it has not already been defined. A vector of symbols may be passed, producing a vector of corresponding results.

(void)functionSignature([Ns$ functionName = NULL])

Prints function signatures for all functions (if functionName is NULL, the default), or for the function named by functionName, to Eidos's output stream. See section 2.7.4 for more information.

(integer$)getSeed(void)

Returns the **random number seed**. This is the last seed value set using setSeed(); if setSeed() has not been called, it will be a seed value chosen based on the process-id and the current time when Eidos was initialized, unless the Context has set a different seed value.

`(void)license(void)`

Prints Eidos's license terms to Eidos's output stream.

`(void)ls(void)`

Prints all currently defined variables to Eidos's output stream. See section 2.4.1 for more information.

`(void)rm([Ns variableNames = NULL], [logical$ removeConstants = F])`

Removes global variables from the Eidos namespace; in other words, it causes the variables to become undefined. Variables are specified by their `string` name in the `variableNames` parameter. If the optional `variableNames` parameter is `NULL` (the default), *all* variables will be removed (be careful!). If the optional parameter `removeConstants` is `F` (the default), then attempting to remove a constant is an error; if `removeConstants` is `T`, constants defined with `defineConstant()` may be removed, but attempting to remove intrinsic Eidos constants is still an error.

`(*)sapply(* x, string$ lambdaSource, [string$ simplify = "vector"])`

Named `apply()` prior to Eidos 1.6 / SLiM 2.6

Applies a block of Eidos code to the elements of x. This function is sort of a hybrid between `c()` and `executeLambda()`; it might be useful to consult the documentation for both of those functions to better understand what `sapply()` does. For each element in `x`, the lambda defined by `lambdaSource` will be called. For the duration of that callout, a variable named `applyValue` will be defined to have as its value the element of `x` currently being processed. The expectation is that the lambda will use `applyValue` in some way, and will return either `NULL` or a new value (which need not be a singleton, and need not be of the same type as `x`). The return value of `sapply()` is generated by concatenating together all of the individual vectors returned by the lambda, in exactly the same manner as the `c()` function (including the possibility of type promotion).

Since this function can be hard to understand at first, here is an example:

```
sapply(1:10, "if (applyValue % 2) applyValue ^ 2;");
```

This produces the output `1 9 25 49 81`. The `sapply()` operation begins with the vector `1:10`. For each element of that vector, the lambda is called and `applyValue` is defined with the element value. In this respect, `sapply()` is actually very much like a `for` loop. If `applyValue` is even (as evaluated by the modulo operator, `%`), the condition of the `if` statement is `F` and so `NULL` is implicitly returned by the lambda (since the `if` has no `else` clause). If `applyValue` is odd, on the other hand, the lambda returns its square (as calculated by the exponential operator, `^`). Just as with the `c()` function, `NULL` values are dropped during concatenation, so the final result contains only the squares of the odd values.

This example illustrates that the lambda can "drop" values by returning `NULL`, so `sapply()` can be used to select particular elements of a vector that satisfy some condition, much like the subscript operator, `[]`. The example also illustrates that input and result types do not have to match; the vector passed in is `integer`, whereas the result vector is `float`.

Beginning in Eidos 1.6, a new optional parameter named `simplify` allows the result of `sapply()` to be a matrix or array in certain cases, better organizing the elements of the result. If the `simplify` parameter is `"vector"`, the concatenated result value is returned as a plain vector in all cases; this is the default behavior, for backward compatibility. Two other possible values for `simplify` are presently supported. If `simplify` is `"matrix"`, the concatenated result value will be turned into a matrix with one column for each non-`NULL` value returned by the lambda, as if the values were joined together with `cbind()`, as long as all of the lambda's return values are either (a) `NULL` or (b) the same length as the other non-`NULL` values returned. If `simplify` is `"match"`, the concatenated result value will be turned into a vector, matrix, or array that exactly matches the dimensions as `x`, with a one-to-one correspondence between `x` and the elements of the return value just like a unary operator, as long as all of the lambda's return values are singletons (with no `NULL` values). Both `"matrix"` and `"match"` will raise an error if their preconditions are not met, to avoid unexpected behavior, so care should be taken that the preconditions are always met when these options are used.

As with `executeLambda()`, all defined variables are accessible within the lambda, and changes made to variables inside the lambda will persist beyond the end of the `sapply()` call; the lambda is executing in the same scope as the rest of your code.

The `sapply()` function can seem daunting at first, but it is an essential tool in the Eidos toolbox. It combines the iteration of a `for` loop, the ability to select elements like operator `[]`, and the ability to assemble results of mixed type together into a single vector like `c()`, all with the power of arbitrary Eidos code execution like `executeLambda()`. It is relatively fast, compared to other ways of achieving similar results such as a `for` loop that accumulates results with `c()`. Like `executeLambda()`, `sapply()` is most efficient if it is called multiple times with a single `string` script variable, rather than with a newly constructed `string` for `lambdaSource` each time.

Prior to Eidos 1.6 (SLiM 2.6), `sapply()` was instead named `apply()`; it was renamed to `sapply()` in order to more closely match the naming of functions in R. This renaming allowed a new `apply()` function to be added to Eidos that operates on the margins of matrices and arrays, similar to the `apply()` function of R (see `apply()`, above).

`(void)setSeed(integer$ seed)`

Set the random number seed. Future random numbers will be based upon the seed value set, and the random number sequence generated from a particular seed value is guaranteed to be reproducible. The last seed set can be recovered with the `getSeed()` function.

`(void)source(string$ filePath)`

Executes the contents of an Eidos source file found at the filesystem path `filePath`. This is essentially shorthand for calling `readFile()`, joining the read lines with newlines to form a single string using `paste()`, and then passing that string to `executeLambda()`. The source file must consist of complete Eidos statements. Regardless of what the last executed source line evaluates to, `source()` has no return value.

`(void)stop([Ns$ message = NULL])`

Stops execution of Eidos (and of the Context, such as the running SLiM simulation, if applicable), in the event of an error. If the optional `message` parameter is not `NULL`, it will be printed to Eidos's output stream prior to stopping.

`(logical$)suppressWarnings(logical$ suppress)`

Turns suppression of warning messages on or off. The `suppress` flag indicates whether suppression of warnings should be enabled (`T`) or disabled (`F`). The previous warning-suppression value is returned by `suppressWarnings()`, making it easy to suppress warnings from a given call and then return to the previous suppression state afterwards. It is recommended that warnings be suppressed only around short blocks of code (not all the time), so that unexpected but perhaps important warnings are not missed. And of course warnings are generally emitted for good reasons; before deciding to disregard a given warning, make sure that you understand exactly why it is being issued, and are certain that it does not represent a serious problem.

```
(string)system(string$ command, [string args = ""], [string input = ""],
    [logical$ stderr = F], [logical$ wait = T])
```

Runs a Un*x command in a `/bin/sh` shell with optional arguments and input, and returns the result as a vector of output lines. The `args` parameter may contain a vector of arguments to `command`; they will be passed directly to the shell without any quoting, so applying the appropriate quoting as needed by `/bin/sh` is the caller's responsibility. The arguments are appended to `command`, separated by spaces, and the result is passed to the shell as a single command string, so arguments may simply be given as part of `command` instead, if preferred. By default no input is supplied to `command`; if `input` is non-empty, however, it will be written to a temporary file (one line per `string` element) and the standard input of `command` will be redirected to that temporary file (using standard `/bin/sh` redirection with `<`, appended to the command string passed to the shell). By default, output sent to standard error will not be captured (and thus may end up in the output of the SLiM process, or may be

lost); if `stderr` is `T`, however, the standard error stream will be redirected into standard out (using standard `/bin/sh` redirection with `2>&1`, appended to the command string passed to the shell).

Arbitrary command strings involving multiple commands, pipes, redirection, etc., may be used with `system()`, but may be incompatible with the way that `args`, `input`, and `stderr` are handled by this function, so in this case supplying the whole command string in `command` may be the simplest course. You may redirect standard error into standard output yourself in `command` with `2>&1`. Supplying input to a complex command line can often be facilitated by the use of parentheses to create a subshell; for example,

```
system("(wc -l | sed 's/ //g')", input=c('foo', 'bar', 'baz'));
```

will supply the input lines to `wc` courtesy of the subshell started for the `()` operator. If this strategy doesn't work for the command line you want to execute, you can always write a temporary file yourself using `writeFile()` and redirect that file to standard input in `command` with `<`.

If `wait` is `T` (the default), `system()` will wait for the command to finish, and return the output generated as a `string` vector, as described above. If `wait` is `F`, `system()` will instead append `" &"` to the end of the command line to request that it be run in the background, and it will not collect and return the output from the command; instead it will return `string(0)` immediately. If the output from the command is needed, it could be redirected to a file, and that file could be checked periodically in Eidos for some indication that the command had completed; if output is not redirected to a file, it may appear in SLiM's output stream. If the final command line executed by `system()` ends in `" &"`, the behavior of `system()` should be just as if `wait=T` had been supplied, but it is recommended to use `wait=T` instead to ensure that the command line is correctly assembled.

`(string$)time(void)`

Returns a **standard time string** for the current time in the local time of the executing machine. The format is `%H:%M:%S` (hour in two digits, then minute in two digits, then seconds in two digits, zero-padded and separated by dashes) regardless of the localization of the executing machine, for predictability and consistency. The 24-hour clock time is used (i.e., no AM/PM).

`(float$)usage([logical$ peak = F])`

Returns the **current (or peak) memory usage**. This is the amount of memory used by the current process, in MB (megabytes); multiply by `1024*1024` to get the usage in bytes. If `peak` is `F` (the default), the current memory usage is returned; if `peak` is `T`, the maximum memory usage over the history of the process is returned. Memory usage is a surprisingly complex topic; the particular metric reported by `usage()` is the resident set size, or RSS, which includes memory usage from shared libraries, but does not include memory that is swapped out or has never been used. For most purposes, RSS is a useful metric of memory usage from a practical perspective. On some platforms (AIX, BSD, Solaris) the memory usage reported may be zero, but it should be correct on both Mac OS X and Linux platforms.

This function can be useful for documenting the memory usage of long runs as they are in progress; in SLiM, it could also be used to trigger tree-sequence simplification with a call to `treeSeqSimplify()`, to reduce memory usage when it becomes too large, but keep in mind that the simplification process itself may cause a substantial spike in memory usage.

When running under SLiM, other tools for monitoring memory usage include the `slim` command-line options `-m[em]` and `-M[emhist]`, and the `outputUsage()` method of `SLiMSim`; see the SLiM manual for more information.

`(float)version([logical$ print = T])`

Get Eidos's version. There are two ways to use this function. If `print` is `T`, the default, then the version number is printed to the Eidos output stream in a formatted manner, like "`Eidos version 2.1`". If Eidos is attached to a Context that provides a version number, that is also printed, like "`SLiM version 3.1`". In this case, the Eidos version number, and the Context version number if available, are returned as an invisible `float` vector. This is most useful when using Eidos interactively. If `print`

is `F`, on the other hand, nothing is printed, but the returned `float` vector of version numbers is not invisible. This is useful for scripts that need to test the Eidos or Context version they are running against.

In both cases, in the `float` version numbers returned, a version like 2.4.2 would be returned as `2.42`; this would not scale well to subversions greater than nine, so that will be avoided in our versioning.

3.11 Built-in methods

These methods are built into Eidos, although Eidos has no object classes of its own. All objects defined by the Context will automatically inherit these methods.

`+ (integer$)length(void)`

Returns the size (e.g., length) of the receiving object. This is equivalent to the `length()` (or `size()`) function; in other words, for any `object x`, the return value of the function call `length(x)` equals the return value of the class method call `x.length()`. This method is provided solely for syntactic convenience. Note that `+length()` is a synonym for `+size()`.

`+ (void)methodSignature([Ns$ methodName = NULL])`

Prints the method signature for the method specified by `methodName`, or for all methods supported by the receiving object if `methodName` is `NULL` (the default).

`+ (void)propertySignature([Ns$ propertyName = NULL])`

Prints the property signature for the property specified by `propertyName`, or for all properties supported by the receiving object if `propertyName` is `NULL` (the default).

`+ (integer$)size(void)`

Returns the size of the receiving object. This is equivalent to the `size()` (or `length()`) function; in other words, for any `object x`, the return value of the function call `size(x)` equals the return value of the class method call `x.size()`. This method is provided solely for syntactic convenience. Note that `+length()` is a synonym for `+size()`.

`– (void)str(void)`

Prints the internal property structure of the receiving object; in particular, the element type of the object is printed, followed, on successive lines, by all of the properties supported by the object, their types, and a sample of their values.

4. User-defined functions

Beginning in Eidos 1.5 (SLiM 2.5), it is now possible to define your own functions in Eidos, rather than being limited to the built-in functions described in chapter 3. Doing so is quite simple; first you declare the new function, and then you supply a compound statement that defines the code of the new function. Unlike in languages such as C, where declaration and definition of functions are separate and are often done in different source files, in Eidos these two actions are always done in conjunction, as described in the following subsection.

4.1 Declaring and defining new functions

As a trivial example, suppose we wish to define a function that doubles whatever `float` value is passed to it. This is very easy to do:

```
function (float)double(float x)
{
    return 2 * x;
}
```

The `function` keyword initiates the declaration of a new function. It is followed by the full signature for the new function, as described in sections 2.7.4–2.7.7; here the signature declares that the function is named `double`, takes a parameter named `x` that is of type `float`, and returns type `float`. This signature is then followed by the definition of the new function, in the form of a compound statement; here, the `double()` function is defined as returning two times the value it was passed. Note that a `return` statement is used here to explicitly return a specified value from the function (see section 2.6.6); if no `return` statement is encountered, `void` will be implicitly returned (which will result in a type-check error unless the function was declared as potentially returning `void`).

Calling such functions works in exactly the same way as calling built-in functions:

```
> double(5.35)
10.7
```

Functions may be recursive; a simple `factorial()` function might be defined recursively as:

```
function (integer)factorial(integer x)
{
    if (x <= 1)
      return 1;
    else
      return x * factorial(x - 1);
}
```

This works well enough, as you can see:

```
> factorial(13)
6227020800
```

As with the built-in Eidos functions, user-defined functions may take multiple parameters, each of which may be allowed to be one of several different possible types. Parameters to user-defined functions may also be optional, with a default value if left unsupplied. Finally, functions are *scoped*; the code inside them executes in a private namespace in which only the parameters to the function are available, and variables defined inside a function will not persist beyond the end of the function's execution. These topics will be covered in more detail in the following sections.

4.2 Complex type specifications

The functions defined above used only simple built-in types; one took a `float` and returned a `float`, the other took an `integer` and returned an `integer`. Just like some of the built-in functions in Eidos, however, your user-defined functions may use a more complex signature that allows and disallows specific types for each parameter.

Eidos is a stickler for types, and does not automatically coerce variables to a different type to satisfy a function's signature, so calling the `double()` function we defined above with an `integer` argument is an error:

```
> double(7);
ERROR (EidosCallSignature::CheckArguments): argument 1 (x) cannot be type
integer for function double().
```

This may be remedied by passing `7.0` instead, or by explicitly coercing the argument to `float` with `asFloat()`; or, more usefully, the function may be fixed to work with `integer` arguments as well. To demonstrate this last possibility, we can redefine `double()` (the new definition will replace the old):

```
function (numeric)double(numeric x)
{
    return 2 * x;
}
```

Recall that `numeric` is an Eidos keyword that refers to either (both) `integer` and `float` type (see, e.g., section 2.7.6). Now, if called with an `integer` argument, `double()` returns an `integer`:

```
> double(7)
14
```

If called with a `float` argument, it will instead return a `float`; the function's code will be executed with whatever type of argument is passed to it, since Eidos is a dynamically-typed language.

To underline that last point about dynamic typing, let's expand our definition of `double()` again, this time to support `string` arguments as well:

```
function (ifs)double(ifs x)
{
    return x + x;
}
```

The `ifs` type specifier indicates that the type of the parameter and the return value may be `integer`, `float`, or `string` (see section 2.7.6); it could just as well have been specified as `sif` or `fsi`, since the order of the letters is unimportant. This definition leverages the fact that while the `*` operator does not work with strings, the `+` operator does. Our function behaves just as it did before for `integer` and `float` arguments, but now it works for `string` too:

```
> double("foo")
"foofoo"
```

So far, so good. Let's expand our function to work with `logical` as well:

```
function (lifs)double(lifs x)
{
    return x + x;
}
```

Now if we call it with a `logical` argument we get this:

```
> double(T)
ERROR (EidosInterpreter::Evaluate_Plus): the combination of operand types
logical and logical is not supported by the binary '+' operator.
```

Of course this makes sense; `T+T` has no meaning in Eidos and is not legal. But suppose we would like `double()` to do something useful with a `logical` argument, such as replicate the argument into a vector of twice the length. We can implement that by having `double()` check the type of its argument explicitly:

```
function (lifs)double(lifs x)
{
    if (isLogical(x))
       return rep(x, 2);
    else
       return x + x;
}
```

This now works with `logical`:

```
> double(T)
T T
```

It has replicated the `logical` vector supplied using the `rep()` function. Speaking of vectors, since we didn't specify that `double()`'s argument or return value is a singleton (which we would do using the `$` symbol), `double()` works with non-singleton vectors automatically:

```
> double(c(1.6, -9.4))
3.2 -18.8
> double(c(3, 7))
6 14
> double(c("foo", "bar"))
"foofoo" "barbar"
> double(c(T, F))
T F T F
```

Finally, note that this version of `double()` could be written more concisely using the ternary conditional operator (see section 2.3.5):

```
function (lifs)double(lifs x)
{
    return isLogical(x) ? rep(x, 2) else x + x;
}
```

We will use that syntax in the sections that follow, for brevity.

4.3 Type specifications that include objects

In the previous section we built a function that works with arguments of type `logical`, `integer`, `float`, and `string`. What about arguments of type `object`? Since the base Eidos language, with no Context, does not contain any `object` classes, in this section we will consider this question from the perspective of SLiM, the Context in which most users will be using Eidos. SLiM defines a class named `Subpopulation`, and instances of that class – individual subpopulations – are often named `p1`, `p2`, etc.

First of all, let's expand the `double()` function to take any `object` argument and replicate it:

```
function (lifso)double(lifso x)
{
    return (isLogical(x) | isObject(x)) ? rep(x, 2) else x + x;
}
```

In SLiM's console, we can use this to replicate a vector containing subpopulations, for example:

```
> double(c(p1, p2))
Subpopulation<p1> Subpopulation<p2> Subpopulation<p1> Subpopulation<p2>
```

So far, so good. But suppose we wanted it to do something smarter with a subpopulation object, such as return an `integer` that is double the number of individuals in the subpopulation supplied? Let's implement that, allowing the function to take only objects of type `Subpopulation`:

```
function (lifs)double(lifso<Subpopulation> x)
{
    if (isObject(x))
      return x.individualCount * 2;
    else
      return isLogical(x) ? rep(x, 2) else x + x;
}
```

The new `<Subpopulation>` syntax after the `o` in the parameter type specifier indicates that if an `object` is present, it is required to be of type `Subpopulation`. The type-checking machinery in Eidos will raise an error if an `object` parameter of a different class is passed in. Note that since the function returns an `integer` when a subpopulation is passed in, the return-type specifier still does not list `object` as a possibility. This new function behaves as desired; when passed subpopulations of size `500`, it provides double their size:

```
> double(c(p1, p2))
1000 1000
```

As a final exercise, let us suppose that we want this behavior when a `Subpopulation` argument is supplied, but we want the old behavior of vector replication when an `object` of any other class is supplied. This is also straightforward:

```
function (lifso)double(lifso x)
{
    if (elementType(x) == "Subpopulation")
      return x.individualCount * 2;
    else
      return (isLogical(x) | isObject(x)) ? rep(x, 2) else x + x;
}
```

This uses the `elementType()` function to diagnose the case in which a `Subpopulation` has been passed, and treats it specially; all other cases go through the old version of the code:

```
> double(p1)
1000
> double(sim)
SLiMSim SLiMSim
```

In practice, functions of this sort that do radically different things depending upon the type of their arguments are generally a bad idea. However, sometimes providing different behaviors for the same "message" sent to different classes can make sense, as in object-oriented programming.

4.4 Optional parameters and default values

Parameters for user-defined functions may be optional, just as was described for built-in functions in section 2.7.5. Optional parameters are declared using brackets, just as they are shown in function signatures. A default value must be supplied for all optional parameters; if a value is not given by the caller for an optional parameter when a function is called, the default value is automatically supplied for that parameter by Eidos. In this way, all parameters are guaranteed to have defined values by the time the code of the function begins to execute.

As an example, let's go back to an earlier, simpler version of `double()` and make the `x` argument optional, with a default of our favorite `string` value, `"foo"`:

```
function (ifs)double([ifs x = "foo"])
{
    return x + x;
}
```

The brackets indicate that `x` is optional, and `= "foo"` supplies the default value when `x` is omitted. Calling `double()` with no argument is now legal:

```
> double()
"foofoo"
```

Default values may be any numeric or string constant (`6`, `"foo"`, `1.2`), or any of the built-in Eidos constants (`T`, `F`, `E`, `PI`, `INF`, `NAN`, `NULL`). At present it is not legal for a default value to be an expression, or a constant that is not built into Eidos, unlike in some languages such as R. However, it is common practice for `NULL` to be used as a default value indicating that the parameter should be considered to be missing, unspecified, or having some other special value.

Once an optional argument has been specified, all subsequent arguments must also be optional, so that the way that unnamed arguments are assigned is simple and unambiguous when a function is called.

Note that although some built-in Eidos functions, such as `c()`, take a variable number of arguments as specified by an ellipsis (`...`) in their function prototype, this capability is not presently supported for user-defined functions. However, parameters may certainly be vectors containing multiple elements, as we have seen above.

4.5 Scope with user-defined functions

Eidos is largely a scopeless language; there is no concept of block scope or file scope, for example, as there is in lexically scoped languages such as C or C++, and there is no support for dynamic scoping either – no equivalent of environments in R, nor of R's `<<-` operator that sets values in the enclosing scope, for example. Variables that are defined generally go into the global scope, exist forever unless removed with `rm()`, and are visible everywhere thenceforth. There is one major exception to this, however, and it is user-defined functions.

In Eidos, user-defined functions are executed in their own private scope. That private scope contains the parameters passed to the function as arguments, and if the function defines any new variables as it executes they will also be contained by the function's private scope. This private scope ceases to exist at the end of each execution of the user-defined function. There is no way to access the global scope from within the private scope of a user-defined function. In this aspect, then, Eidos embodies pure "functional programming"; functions cannot depend upon any variable state other than the parameters they are given, and cannot have any external side effects apart from the return value they generate. (In other ways, however, Eidos does not follow the functional programming paradigm, especially in its object-oriented aspects; for example, method calls can

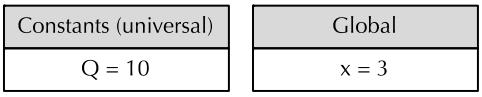
have side effects on objects. If an object is passed as a parameter to an Eidos user-defined function, the function can therefore then have external side effects after all, by modifying that object; and it can have external dependencies after all, on any other objects it can reach through the object it was given.)

The above statements are true insofar as variables are concerned. A particularly unusual feature of Eidos, however, is that defined constants lived in a separate scope that is always available. Constants defined outside of any function are visible inside user-defined functions, and if a user-defined function defines a new constant, that constant will live on even after the function completes. Defined constants are therefore an exception to the functional-programming paradigm stated above; for defined constants, Eidos is truly scopeless since all code shares the same universal constants scope.

To illustrate these scoping rules, consider this Eidos code:

```
function (void)foo(integer x)
{
    x = x + 1;
    d = 2 * x;
    defineConstant("P", d + 1);
}

function (integer)bar(integer y)
{
    return y * P * Q;
}

x = 3;
defineConstant("Q", x + 7);

// Point A

foo(x);
z = bar(5);

// Point B
```

The way in which this code behaves can be visualized pictorially. At the comment "Point A", the symbol tables defined in Eidos look like this:

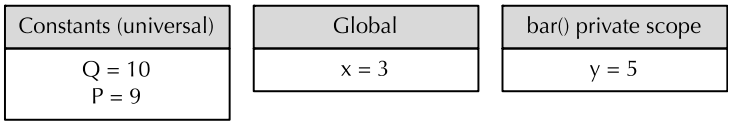


The universal constants table contains the constant `Q`, and the global scope contains the variable `x`.

When the user-defined function `foo()` is called, the symbol tables at the end of the execution of that function, just before it returns, then look like:

The function `foo()` has defined its own variable named `x` (a parameter to the function, in fact), within its private scope; this is unrelated to the `x` defined in the global scope. A variable `d` has also been defined in `foo()`'s private scope. The constant `P` has been added the universal constants table, and will persist beyond the end of `foo()`'s execution.

When `bar()` is called, the symbol tables at the end of the execution of that function, just before it returns, then look like:



And finally, at the comment "Point B" things look like this:

Note that after `foo()` finishes executing its private scope no longer even exists, and likewise for `bar()`. Note also that the symbol `x` inside `foo()` is a different symbol than `x` in the global scope, and that inside `foo()` there is no way to access the global `x` (although its value is passed to `foo()` as a parameter); once `foo()` increments its local `x` the divergence between its local `x` and the global `x` becomes more apparent, but it was the case from the very beginning of `foo()`'s execution. Recursive calls to a given function each receive their own private scope, and there is no way for one recursion of a given function to access variables in a different recursion of the function (except through passing values as arguments or receiving values as returns, as usual).

In case you are wondering, user-defined functions in Eidos are not themselves variables, and are unscoped – all defined functions are available everywhere, similarly to how Eidos treats constants. References to functions are resolved at the point of the call dispatch, so function `foo()`, in the example above, could call function `bar()` even though `bar()` is defined after `foo()`; there is no need for forward declaration of `bar()` as there is in languages like C and C++. If `bar()` has not been declared at the point in time when `foo()` is called and actually attempts to call `bar()`, an error will result; if `bar()` has been defined, on the other hand, then the implementation of `bar()` that is defined at that moment in time is called. It is not possible to redefine the built-in Eidos functions, but the definition of user-defined functions be changed, as we have seen above. (It would be wise to use this only to refine functions when working interactively, however, rather than designing spaghetti code that relies upon dynamically redefining functions during the course of execution.)

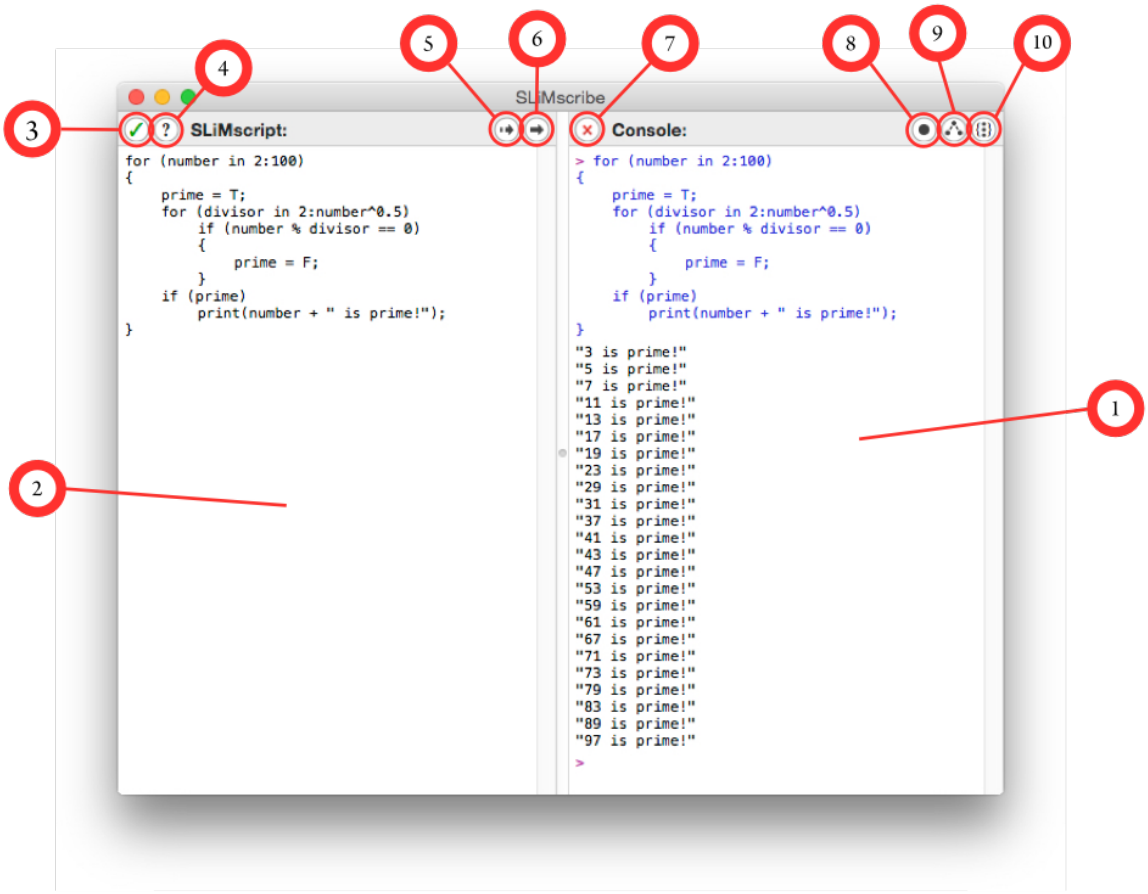
There is presently no way to tell, in executing code, whether a given function is built-in or user-defined, nor is there any way in code to obtain the source code for a user-defined function; as far as executing code is concerned, user-defined functions are identical to built-in functions.

5. EidosScribe

5.1 EidosScribe overview

EidosScribe is a Mac OS X application that provides an interactive scripting environment for Eidos. It's a very simple app, so this manual will provide only a very quick overview of it.

All the action in EidosScribe happens in one window, the scripting window:



The numbers in this screenshot mark controls and areas of the EidosScribe window:

1. the **Console** area,
2. the **Script** area,
3. the **Check Script** button,
4. the **Eidos Help** button,
5. the **Execute Selection** button,
6. the **Execute File** button,
7. the **Clear Console** button,
8. the **Show Tokens** button,
9. the **Show AST** button, and
10. the **Show Execution Trace** button.

5.2 Interactive scripting

The Console area (1) is the most important part of the EidosScribe window for interactive scripting. The > prompt shown by EidosScribe indicates that it is ready for new input. Input may be entered at the prompt; that single line may contain more than one statement, but it must be syntactically complete. In other words, unlike some other interpreter environments, a partial statement will not result in a continuation prompt requesting further lines of input; it will instead result in a parsing error. Multi-line input can be entered using option-Return (or option-Enter) to insert newlines; the input will not be processed until Return or Enter is pressed.

Given syntactically and semantically correct input, EidosScribe will tokenize, parse, and execute the input, and will show the output and results in the Console. Color-coding is used to differentiate among the different blocks of text in the Console; blue text is user input, whereas black text is output from Eidos. The Console area can be cleared by pressing the Clear Console button (7).

The Console remembers its history of previously executed commands. You may flip through that history using the up and down arrow keys. This is particularly useful when a like of input generated an error due to a typo; you can press up-arrow and edit the line to fix the problem.

Note that for interactive scripting, a semicolon is not required at the end of your input line to terminate your last statement. So `6+7` is acceptable input, and will be executed as `6+7;` after correction. This is intended to be a convenience feature for quick interactive sessions; remembering the semicolons can be frustrating, especially for those coming from languages such as R that do not require them.

5.3 File-based script execution

EidosScribe also allows for programming based upon a script file. The Script area (2) shows your script file. You may enter any text you wish in that area; it is of no significance until you tell EidosScribe to execute it in the Console.

To execute a block of your script, select it and press the Execute Selection button (5). If you wish to execute a single line of script, you can simply place the insertion point anywhere within that line and press Execute Selection, and the full line will be executed. You can also execute the full contents of your Script area by pressing the Execute File button (6).

You can check the syntax of your script at any time by pressing the Check Script button (3). This runs the tokenizer and parser on your script, so syntax errors will be found and flagged. It does not, however, execute your script, so semantic errors such as supplying the wrong number of parameters to a function or mixing incompatible types will not be found; those are runtime errors that are only found when your script is actually executed.

Help with Eidos can be brought up by pressing the Eidos Help button (4). In fact, that button may well bring up this very manual; but perhaps something briefer and more immediately helpful will appear, depending upon the version of EidosScribe you are using.

Unlike in interactive scripting (section 5.2), file-based script execution requires a semicolon at the end of every statement; EidosScribe will not correct your script prior to execution. Semicolons are, in fact, a required part of the grammar of Eidos, and the cheap hack that EidosScribe uses to add them in interactive mode would not work well as a general solution.

5.4 Code completion

It can be hard to remember the names of all of the properties and methods exported by Eidos objects, as well as the names of the functions and global variables. For this reason, EidosScribe provides a code completion mechanism to assist you in programming. The keyboard command to

initiate code completion is the Escape key (esc); Command-period (⌘.) may also work depending upon your key bindings.

The situations in which code completion will work are somewhat limited. If the selection is at a point in your script where a new statement is beginning (after a semicolon, `;`, or a right brace, `}`, for example), you will be offered a list of all of the global constants and variables, functions, and statement keywords available. If the selection is in an `object` key path such as `sim.chromosome.` you will be offered a list of the properties and methods supported for that key path. In both of these circumstances, a partial identifier just prior to the selection will be used to filter the choices available.

For example, if you begin a new statement and type `si` and then press esc, you will be offered completions beginning with "si": `sim` (the `SLiMSim` simulation global variable), if you are using Eidos in SLiM, and `sin()` and `size()`, the two functions whose names begin with "si". Similarly, if (in SLiMgui, in fact, not in EidosScribe) you type `sim.chromosome.r` and then press esc, you will be offered completions based upon the `Chromosome object` type that begin with "r": `recombinationEndPositions` and `recombinationRates`.

Eidos is generally aware of the types produced by functions, methods, and properties; it uses the same information that it uses to provide the output for the `functionSignature()` function (section 2.7.4) and the `methodSignature()` and `propertySignature()` methods (section 2.8.6). In some cases, that information is insufficient; the `sortBy()` function returns an object, for example, but it is not known what class that object will be until the code is actually executed. In such cases, code completion cannot provide suggestions. In most cases, however, it has the information it needs.

Code completion actually parses and analyzes your code whenever you invoke it. What this means is that it can anticipate the effects of code you have written before you actually execute that code. For example, if you go to EidosScribe's Script area and type `xyzzy = 17;` but do not execute that line, the symbol `xyzzy` is not defined in Eidos. Nevertheless, if you now go to the following line, type `xy` and press esc, you will see that code completion offers `xyzzy` as a candidate. Eidos expects that `xyzzy` will become defined when you eventually get around to executing the code that you are writing, and so it considers that variable to exist for purposes of code completion. In fact, if you are working in SLiMgui and write `sim.addSubpop("p6", 500);` and then go to the next line and press esc, `p6` will be offered as a candidate; Eidos knows that a side effect of the `addSubpop()` method is to define a new global variable for the new subpopulation.

5.5 Debugging controls

EidosScribe provides some facilities for examining the internals of Eidos. This is intended primarily for debugging, but might also be of interest to those who are curious about how the language is implemented.

5.5.1 Showing tokenization

The Show Tokens button (8), if pressed, causes the token stream obtained from the user's input to be displayed in the Console. Tokens are small chunks into which the input is broken prior to processing. In the English language, tokens would be words and punctuation marks; in Eidos, tokens are operators, numeric and string literals, function names, variable names, and such. Showing tokens allows you to verify that your input was broken into tokens correctly. For example:

```
> x = 3+7*3^2
@x = #3 + #7 * #3 ^ #2 ; EOF
```

You can see that x was tokenized as an identifier (marked with an @), the numbers were tokenized as numeric literals (and thus marked with #), the operators were given their own tokens, and the end of the input was marked with an EOF (End Of File) token.

5.5.2 Showing the abstract syntax tree (AST)

The Show AST button (9), if pressed, causes the AST (Abstract Syntax Tree) generated from the token stream to be shown. The AST is an intermediate representation of the user's input as a tree, where operators are nodes in the tree and their operands are their children. The nesting of the tree is shown with parentheses. So for the input above, we would have:

```
> x = 3+7*3^2
($>
  (=
    @x
    (+
      #3
      (*
        #7
        (^ #3 #2)
      )
    )
  )
)
```

The deepest node is the exponentiation, 3^2. The next node up multiplies the result of that exponentiation by 7, and so forth. At the top of the tree is the parent node for the whole input line, which is marked with the special designator $> representing the EidosScribe prompt. You can examine the AST to confirm that the correct grouping and precedence of operations is being expressed by the tree.

5.5.3 Showing the evaluation trace

The Show Execution Trace button, if pressed, causes EidosScribe to output a trace of Eidos's internal execution as it processes the AST, evaluates it, and generates a result. Evaluating the AST depends upon an algorithm that recursively "walks" the AST, visiting parents and then visiting each of their children in turn. As each child is evaluated, it generates a return value; those return values are collected, and once assembled together, allow the evaluation of the parent, which can then generate its own return value. The execution trace shows entry into, and exit out of, each node in the AST as it is walked, as well as each intermediate return value generated. Since this execution trace refers to functions and state that is internal to the implementation of Eidos, it may be of limited utility; at a minimum, however, it can be used to confirm that execution of the AST results in a call sequence that makes logical sense. For example, for the example above, the execution trace would be:

```
> x = 3+7*3^2
EvaluateInterpreterBlock() entered
  EvaluateNode() : token =
  Evaluate_Assign() entered
    Evaluate_LValueReference() : token @x
    EvaluateNode() : token +
    Evaluate_Plus() entered
      EvaluateNode() : token #3
      Evaluate_Number() entered
      Evaluate_Number() : return == 3
      EvaluateNode() : token *
      Evaluate_Mult() entered
        EvaluateNode() : token #7
        Evaluate_Number() entered
```

```
          Evaluate_Number() : return == 7
          EvaluateNode() : token ^
          Evaluate_Exp() entered
            EvaluateNode() : token #3
            Evaluate_Number() entered
            Evaluate_Number() : return == 3
            EvaluateNode() : token #2
            Evaluate_Number() entered
            Evaluate_Number() : return == 2
          Evaluate_Exp() : return == 9
        Evaluate_Mult() : return == 63
      Evaluate_Plus() : return == 66
    Evaluate_Assign() : return == NULL
  EvaluateInterpreterBlock() : return == NULL
```

At the deepest level, for example, you can see how the exponentiation, `3^2`, is evaluated by first evaluating each operand (getting `3` and `2` as results), and then performing an exponentiation of `3` and `2` to produce a result of `9`. That `9` is then passed up as an operand to its parent, `Evaluate_Mult`, which evaluates the multiplication of `7` and `3^2`. When the recursive tree walk completes, the topmost node returns `NULL`, because the assignment operator always generates `NULL` as its result (the assignment of a value to a variable is a side effect).

If all that made no sense, it doesn't matter. It is of interest mainly to those who are interested in debugging Eidos and EidosScribe – although an understanding of it might be useful in writing scripts, too.

6. Colors in Eidos

Eidos has no direct support for graphical user interfaces at present; it is a language that is used purely at the command line or as a scripting interface. It can nevertheless be useful for Eidos to provide some support for the control of a graphical user interface implemented by the Context. The first foray that we have taken in this direction is to allow Eidos script to control the colors of some of the graphical elements presented in SLiMgui (see the `color` and `colorSubstitution` properties on various SLiM classes). To facilitate this, some basic support has been added to Eidos for the specification of colors and the conversion of colors between the RGB and HSV color spaces.

Basically, Eidos follows the lead of the R language in this respect: colors are represented by `string` values, as described in detail below. Colors can also be described with `float` vectors containing either RGB (red/green/blue) or HSV (hue/saturation/value) values, but the intention is that the Context would not use such color descriptions directly; instead, the `float` vector description would be converted into a color string that the Context would use. The color string is therefore intended to be the "native" color format. The `rgb2color()` function is provided to convert an RGB `float` vector into a color string, while the `color2rgb()` function can perform the reverse conversion. The `hsv2rgb()` and `rgb2hsv()` functions provide the ability to convert between color spaces when using `float` vector color descriptions. (See section 3.8 for descriptions of these functions.)

Two ways to specify a color string are supported: named colors, and hexadecimal RGB values. To specify a named color, simply give the name of the color (in quotes, since it is a `string` literal value). For example, `"red"` is a standard shade of red, and `"coral"` is a particular shade of orange. In all, 657 named colors are supported, exactly matching the named colors supported by R. A quick internet search will supply several websites with lists of these named colors presented in various ways; for example, http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf is one useful list. It should be noted that R's color list has some flaws; for example, `"azure"` is not azure at all, but instead a shade just slightly off white, for no apparent reason. For compatibility, however, these flaws are mirrored precisely in the named colors available in Eidos.

To specify a color with a hexadecimal RGB string, supply a string of the form `"#RRGGBB"`, where `RR` is a two-digit hexadecimal value for red, and `GG` and `BB` are likewise values for green and blue. Hexadecimal is base 16; digits in hexadecimal are `0123456789ABCDEF`, where base 10 math of course uses only `0123456789`. A hexadecimal value of `00` represents zero, whereas `FF` represents `255` in base 10. The red, green, and blue values can therefore range from `0` to `255`. Black is represented by `"#000000"`, white by `"#FFFFFF"`. Pure red is `"#FF0000"`, pure green is `"#00FF00"`, and pure blue is `"#0000FF"`. An R color chart showing the hexadecimal values for all of the standard named colors is at http://research.stowers-institute.org/efg/R/Color/Chart/ColorChart.pdf; this can be helpful for devising new hexadecimal colors if you aren't used to this representation scheme. This reveals, for example, that `"coral"` is `"#FF7F50"`, a fact that could also be discovered with a call to `color2rgb("coral")`; a shade with a bit less green in it could then be easily derived by changing the `7F` portion to a smaller hexadecimal number.

Usually, the colors a model uses will be hard-coded into the model as `string` literals. It is certainly possible to create color strings at runtime, though. The simplest way is to use the `rgb2color()` function. It expects a `float` vector containing the red, green, and blue components as values in the interval `[0, 1]`; it rescales these values to `[0, 255]` and returns the corresponding hexadecimal string. For example, to get a shade of bright red with a little green mixed in (making it a little bit orange), do:

```
rgb2color(c(1.0, 0.3, 0.0));
```

The `format()` function of Eidos can also produce a two-digit hexadecimal string for a given `integer`; with three calls to `format()` to generate the red, green, and blue portions of the color string, and some use of the `+` operator to stick it all together, making a color string representing arbitrary RGB color components should be straightforward. For example:

```
r = 127;
g = 63;
b = 255;
"#" + format("%.2X",r) + format("%.2X",g) + format("%.2X",b);
```

As long as `r`, `g`, and `b` are `integer` values within the interval [0, 255], this code ought to produce a properly-formatted color string.

7. Eidos language reference sheet

This reference sheet may be downloaded as a separate PDF from http://messerlab.org/slim/.

Types (in promotion order):

- `NULL`: no explicit value
- `logical`: true/false values
- `integer`: whole numbers
- `float`: real numbers
- `string`: characters
- `object`: Context objects, such as SLiM objects

Constants:

- `E`: *e* (2.7182...) (float)
- `PI`: π (3.1415...) (float)
- `F`: false (logical)
- `T`: true (logical)
- `INF`: infinity (float)
- `NAN`: not a number (float)
- `NULL`: a `NULL`-type value

Operators (precedence order):

`[]`, `()`, `.`	subset, call, member
`+`, `−`, `!`	unary plus/minus, logical not
`^`	exponentiation
`:`	sequence construction
`*`, `/`, `%`	multiplication, division, modulo
`+`, `−`	addition and subtraction
`<`, `>`, `<=`, `>=`	less-than, greater-than, etc.
`==`, `!=`	equality and inequality
`&`	logical (Boolean) and
`|`	logical (Boolean) or
`?else`	ternary conditional
`=`	assignment

Empty statement: `;`
Compound statement: `{ ... }`
Single-line comment: `// ...`
Block comment: `/* ... */`

Special Statements:

`if (condition) statement [else statement]`	conditional statement with optional alternative statement
`while (condition) statement`	loop while `T`, with a condition test at the loop top
`do statement while (condition)`	loop while `T`, with a condition test at the loop bottom
`for (identifier in vector) statement`	iterate through the values in a vector, executing statement
`next`	skip the remainder of this iteration of the enclosing loop
`break`	break out of the enclosing loop entirely
`return [return-value]`	exit a script block, returning a value if one is given
`function (return)name(params) { ... }`	create a user-defined function (only at the top level)

Math:

`(numeric)abs(numeric x)`: absolute value of `x`
`(float)acos(numeric x)`: arc cosine of `x`
`(float)asin(numeric x)`: arc sine of `x`
`(float)atan(numeric x)`: arc tangent of `x`
`(float)atan2(numeric x, numeric y)`: arc tangent of `y/x`, inferring the correct quadrant
`(float)ceil(float x)`: ceiling (rounding toward +∞) of `x`
`(float)cos(numeric x)`: cosine of `x`
`(numeric)cumProduct(numeric x)`: cumulative product along `x`
`(numeric)cumSum(numeric x)`: cumulative summation along `x`
`(float)exp(numeric x)`: base-*e* exponential of `x`, e^x
`(float)floor(float x)`: floor (rounding toward −∞) of `x`
`(integer)integerDiv(integer x, integer y)`: integer division of `x` by `y`
`(integer)integerMod(integer x, integer y)`: integer modulo of `x` by `y` (the remainder after integer division)
`(logical)isFinite(float x)`: `T` or `F` for each element of `x`; "finite" means not `INF`, `-INF`, or `NAN`
`(logical)isInfinite(float x)`: `T` or `F` for each element of `x`; "infinite" means `INF` and `-INF` only
`(logical)isNAN(float x)`: `T` or `F` for each element of `x`; "infinite" means `NAN` only
`(float)log(numeric x)`: base-*e* logarithm of `x`
`(float)log10(numeric x)`: base-10 logarithm of `x`
`(float)log2(numeric x)`: base-2 logarithm of `x`
`(numeric$)product(numeric x)`: product of the elements of `x`, Πx
`(float)round(float x)`: round `x` to the nearest values; half-way cases round away from `0`
`(*)setDifference(* x, * y)`: set-theoretic difference, $x \setminus y$
`(*)setIntersection(* x, * y)`: set-theoretic intersection, $x \cap y$
`(*)setSymmetricDifference(* x, * y)`: set-theoretic symmetric difference $x \triangle y$
`(*)setUnion(* x, * y)`: set-theoretic union, $x \cup y$
`(float)sin(numeric x)`: sine of `x`
`(float)sqrt(numeric x)`: square root of `x`
`(numeric$)sum(lif x)`: summation of the elements of `x`, Σx
`(float$)sumExact(float x)`: exact summation of `x` without roundoff error, to the limit of floating-point precision
`(float)tan(numeric x)`: tangent of `x`
`(float)trunc(float x)`: truncation (rounding toward `0`) of `x`

Statistics:

`(float$)cor(numeric x, numeric y)`: sample Pearson's correlation coefficient between `x` and `y`
`(float$)cov(numeric x, numeric y)`: corrected sample covariance between `x` and `y`
`(+$)max(+ x, ...)`: largest value within `x` and the additional optional arguments
`(float$)mean(lif x)`: arithmetic mean of `x`
`(+$)min(+ x, ...)`: smallest value within `x` and the additional optional arguments
`(+)pmax(+ x, + y)`: parallel maximum of `x` and `y` (the element-wise maximum for each corresponding pair)
`(+)pmin(+ x, + y)`: parallel minimum of `x` and `y`(the element-wise maximum for each corresponding pair)
`(numeric)range(numeric x, ...)`: range (min/max) of `x` and the additional optional arguments
`(float$)sd(numeric x)`: corrected sample standard deviation of `x`
`(float$)ttest(float x, [Nf y = NULL], [Nf$ mu = NULL])`: run a one-sample or two-sample *t*-test
`(float$)var(numeric x)`: corrected sample variance of `x`

Vector construction:

`(*)c(...)`: concatenate the given vectors to make a single vector of uniform type
`(float)float(integer$ length)`: construct a `float` vector of `length`, initialized with `0.0`
`(integer)integer(integer$ length, [integer$ fill1 = 0], [integer$ fill2 = 1], [Ni fill2indices = NULL])`: construct an `integer` vector of `length`, initialized with the given fill values
`(logical)logical(integer$ length)`: construct a `logical` vector of `length`, initialized with `F`
`(object<undefined>)object(void)`: construct an empty `object` vector
`(*)rep(* x, integer$ count)`: repeat `x` a given number of times
`(*)repEach(* x, integer count)`: repeat each element of `x` a given number of times
`(*)sample(* x, integer$ size, [logical$ replace = F], [Nif weights = NULL])`: sample from `x`
`(numeric)seq(n$ from, n$ to, [Nif$ by = NULL], [Ni$ length = NULL])`: construct a sequence
`(integer)seqAlong(* x)`: construct a sequence along the indices of `x`
`(integer)seqLen(integer$ length)`: construct a sequence with `length` elements, counting upward from `0`
`(string)string(integer$ length)`: construct a `string` vector of `length`, initialized with `""`

Value inspection / manipulation:

`(logical$)all(logical x, ...)`: `T` if *all* values supplied are `T`, otherwise `F`
`(logical$)any(logical x, ...)`: `T` if *any* values supplied are `T`, otherwise `F`
`(void)cat(* x, [s$ sep = " "])`: concatenate output
`(void)catn([* x = ""], [s$ sep = " "])`: concatenate output with trailing newline
`(string)format(string$ format, numeric x)`: format the elements of `x` as strings
`(logical$)identical(* x, * y)`: `T` if `x` and `y` are identical in all respects, otherwise `F`
`(*)ifelse(logical test, * trueValues, * falseValues)`: vector conditional
`(integer$)length(* x)`: count elements in `x` (synonymous with `size()`)
`(integer)match(* x, * table)`: positions of matches for `x` within `table`
`(integer)nchar(string x)`: character counts for the `string` values in `x`
`(integer)order(+ x, [logical$ ascending = T])`: indexes of `x` that would produce sorted order
`(string$)paste(* x, [string$ sep = " "])`: paste together a string with separators
`(string$)paste0(* x)`: paste together a string with no separators
`(void)print(* x)`: print `x` to the output stream
`(*)rev(* x)`: reverse the order of the elements in `x`
`(integer$)size(* x)`: count elements in `x` (synonymous with `length()`)
`(+)sort(+ x, [logical$ ascending = T])`: sort non-`object` vector `x`
`(object)sortBy(object x, string$ property, [l$ ascending = T])`: sort `object` vector `x` by a property
`(void)str(* x)`: print the external structure of a value
`(string)strsplit(string$ x, [string$ sep = " "])`: split `string` `x` into substrings by separator `sep`
`(string)substr(string x, integer first, [Ni last = NULL])`: get substrings from `x`
`(*)unique(* x, [logical$ preserveOrder = T])`: unique values in `x` (`preserveOrder = F` is faster)
`(integer)which(logical x)`: indices in `x` which are `T`
`(integer$)whichMax(+ x)`: first index in `x` with the maximum value
`(integer$)whichMin(+ x)`: first index in `x` with the minimum value

Distribution drawing / density:

(float)dmvnorm(float x, numeric mu, numeric sigma): multivariate normal density function values
(float)dnorm(float x, [numeric mean = 0], [numeric sd = 1]): normal density function values
(float)pnorm(float q, [numeric mean = 0], [numeric sd = 1]): normal distribution CDF values
(float)rbeta(integer $n, numeric alpha, numeric beta): beta distribution draws
(integer)rbinom(integer $n, integer size, float prob): binomial distribution draws
(float)rcauchy(integer $n, [numeric location = 0], [numeric scale = 1]): Cauchy distribution draws
(integer)rdunif(integer $n, [integer min = 0], [integer max = 1]): discrete uniform distribution draws
(float)rexp(integer $n, [numeric mu = 1]): exponential distribution draws
(float)rgamma(integer $n, numeric mean, numeric shape): gamma distribution draws
(integer)rgeom(integer $n, float p): geometric distribution draws
(float)rlnorm(integer $n, [numeric meanlog = 0], [numeric sdlog = 1]): lognormal distribution draws
(float)rmvnorm(integer $n, numeric mu, numeric sigma): multivariate normal distribution draws
(float)rnorm(integer $n, [numeric mean = 0], [numeric sd = 1]): normal distribution draws
(integer)rpois(integer $n, numeric lambda): Poisson distribution draws
(float)runif(integer $n, [numeric min = 0], [numeric max = 1]): uniform distribution draws
(float)rweibull(integer $n, numeric lambda, numeric k): Weibull distribution draws

Type testing / coercion:

(float)asFloat(+ x): convert x to type float
(integer)asInteger(+ x): convert x to type integer
(logical)asLogical(+ x): convert x to type logical
(string)asString(+ x): convert x to type string
(string$)elementType(* x): element type of x; for object x, this is the class of the object-elements
(logical$)isFloat(* x): T if x is of type float, F otherwise
(logical$)isInteger(* x): T if x is of type integer, F otherwise
(logical$)isLogical(* x): T if x is of type logical, F otherwise
(logical$)isNULL(* x): T if x is of type NULL, F otherwise
(logical$)isObject(* x): T if x is of type object, F otherwise
(logical$)isString(* x): T if x is of type string, F otherwise
(string$)type(* x): type of vector x; this is NULL, logical, integer, float, string, or object

Matrix and array functions:

(*)apply(* x, integer margin, string$ lambdaSource): apply code across margins of matrix/array x
(*)array(* data, integer dim): create an array from data, with dimensionality dim
(*)cbind(...): combine vectors and/or matrices by column
(integer)dim(* x): dimensions of matrix or array x
(*)drop(* x): drop redundant dimensions from matrix or array x
(*)matrix(* data, [Ni$ nrow = NULL], [Ni$ ncol = NULL], [logical$ byrow = F]): create a matrix
(numeric)matrixMult(numeric x, numeric y): matrix multiplication of conformable matrices x and y
(integer$)ncol(* x): number of columns in matrix or array x
(integer$)nrow(* x): number of rows in matrix or array x
(*)rbind(...): combine vectors and/or matrices by row
(*)t(* x): transpose of x

Filesystem access:

(logical$)createDirectory(string$ path): create a new filesystem directory at path
(logical$)deleteFile(string$ filePath): delete file at filePath
(logical$)fileExists(string$ filePath): check for the existence of a file (or directory) at filePath
(string)filesAtPath(string$ path, [logical$ fullPaths = F]): get the names of the files in a directory
(string$)getwd(void): get the current filesystem working directory
(string)readFile(string$ filePath): read lines from the file at filePath as a string vector
(string$)setwd(string$ path): set the filesystem working directory
(logical$)writeFile(string$ filePath, string contents, [logical$ append = F]): write to a file
(numeric)writeTempFile(string$ prefix, string$ suffix, string contents): write to a temporary file

Color manipulation:

(string)cmColors(integer$ n): generate colors in a “cyan-magenta” color palette
(float)color2rgb(string color): convert color string(s) to RGB values
(string)heatColors(integer$ n): generate colors in a “heat map” color palette
(float)hsv2rgb(float hsv): convert HSV color(s) to RGB values
(string)rainbow(integer$ n, [float$ s = 1], [float$ v = 1], [float$ start = 0], [Nf$ end = NULL], [logical$ ccw = T]): generate colors in a “rainbow” color palette
(string)rgb2color(float rgb): convert RGB color(s) to color string(s)
(float)rgb2hsv(float rgb): convert RGB color(s) to HSV values
(string)terrainColors(integer$ n): generate colors in a “terrain” color palette

Miscellaneous:

(void)beep([Ns$ soundName = NULL]): play a sound or beep
(void)citation(void): print the reference citation for Eidos and the current Context
(float$)clock([string$ type = "cpu"]): get the current CPU usage clock, for timing of code blocks
(string$)date(void): get the current date as a formatted string
(void)defineConstant(string$ symbol, + value): define a new constant with a given value
(*)doCall(string$ functionName, ...): call the named function with the given arguments
(*)executeLambda(string$ lambdaSource, [ls$ timed = F]): execute a string as code
(logical)exists(string symbol): T for defined symbols, F otherwise
(void)functionSignature([Ns$ functionName = NULL]): print the call signature(s) for function(s)
(integer$)getSeed(void): get the last random number generator seed set
(void)license(void): print license information for Eidos and the current Context
(void)ls(void): list all variables currently defined
(void)rm([Ns variableNames = NULL], [logical$ removeConstants = F]): remove (undefine) variables
(*)sapply(* x, string$ lambdaSource, [string$ simplify = "vector"]): apply code across elements of x
(void)setSeed(integer$ seed): set the random number generator seed
(void)source(string$ filePath): execute a source file as code
(void)stop([Ns$ message = NULL]): stop execution and print the given error message
(logical$)suppressWarnings(logical$ suppress): suppress (or stop suppressing) warning messages
(string)system(string$ command, [string args = ""], [string input = ""], [logical$ stderr = F], [logical$ wait = T]): run a Un*x command with the given arguments and input
(string$)time(void): get the current time as a formatted string
(float$)usage([logical$ peak = F]): get the current or peak memory usage of the process
(float)version([logical$ print = T]): get the Eidos and Context version numbers

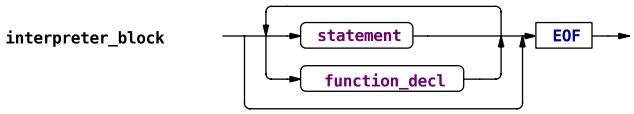
Eidos methods (defined for all classes):

+ (integer$)length(void): count elements in the target object vector (synonymous with size())
+ (void)methodSignature([Ns$ methodName]): print the signature for methodName, or for all methods
+ (void)propertySignature([Ns$ propertyName]): print the signature for propertyName, or for all properties
+ (integer$)size(void): count elements in the target object vector (synonymous with length())
– (void)str(void): print the internal structure (properties, types, values) for an object vector

8. Railroad diagrams

These "railroad diagrams" represent Eidos's grammar in a graphical form equivalent to its EBNF grammar. This grammar has a couple of conflicts. One is due to the `if–else` ambiguity as usual; this is resolved by associating the `else` clause with the closest preceding `if` statement to which it could apply, as in many other languages. Others are resolved by one-token lookahead. For example, if a `param_list` begins with `void` then the following token is checked to determine whether it is a closing parenthesis; if it is, the simple path is taken, otherwise the parser treats the `void` identifier as the beginning of a `param_spec` (and thus the beginning of a `type_spec`). Other cases of grammar conflict are similarly resolved.

8.1 Start rule

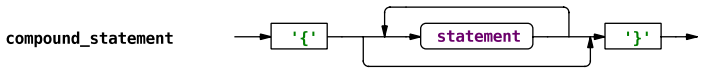


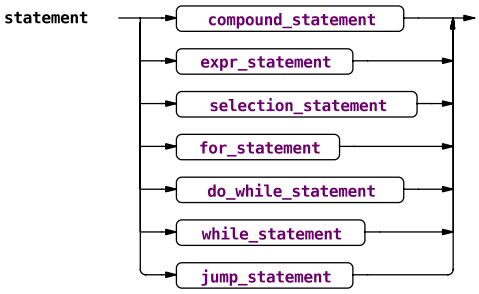
This is Eidos's start rule, used in the interactive interpreter; an interpreter block consists of a series of zero or more statements and function declarations. Statements are described in section 8.2; function declarations are put off until section 8.4.

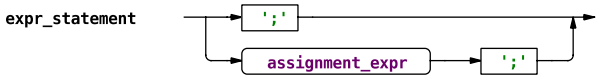
Note that a semicolon terminating the final statement may be optional in the interactive interpreter, which is not reflected in this grammar; it is a convenience added by the interactive interpreter.

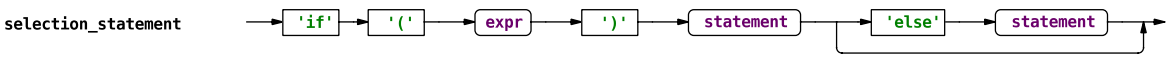
This is not the start rule used by SLiM; SLiM defines a modified Eidos grammar with a different start rule for use in parsing SLiM input files, as described in the SLiM manual.

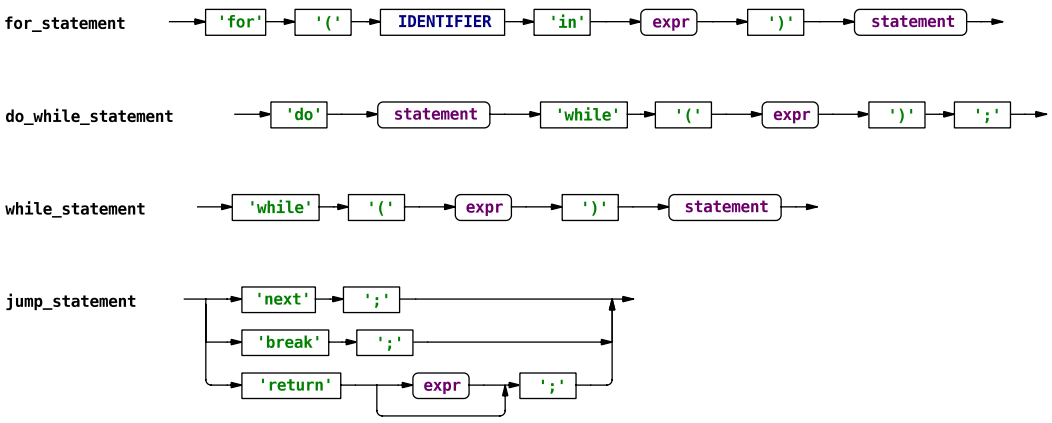
8.2 Statements







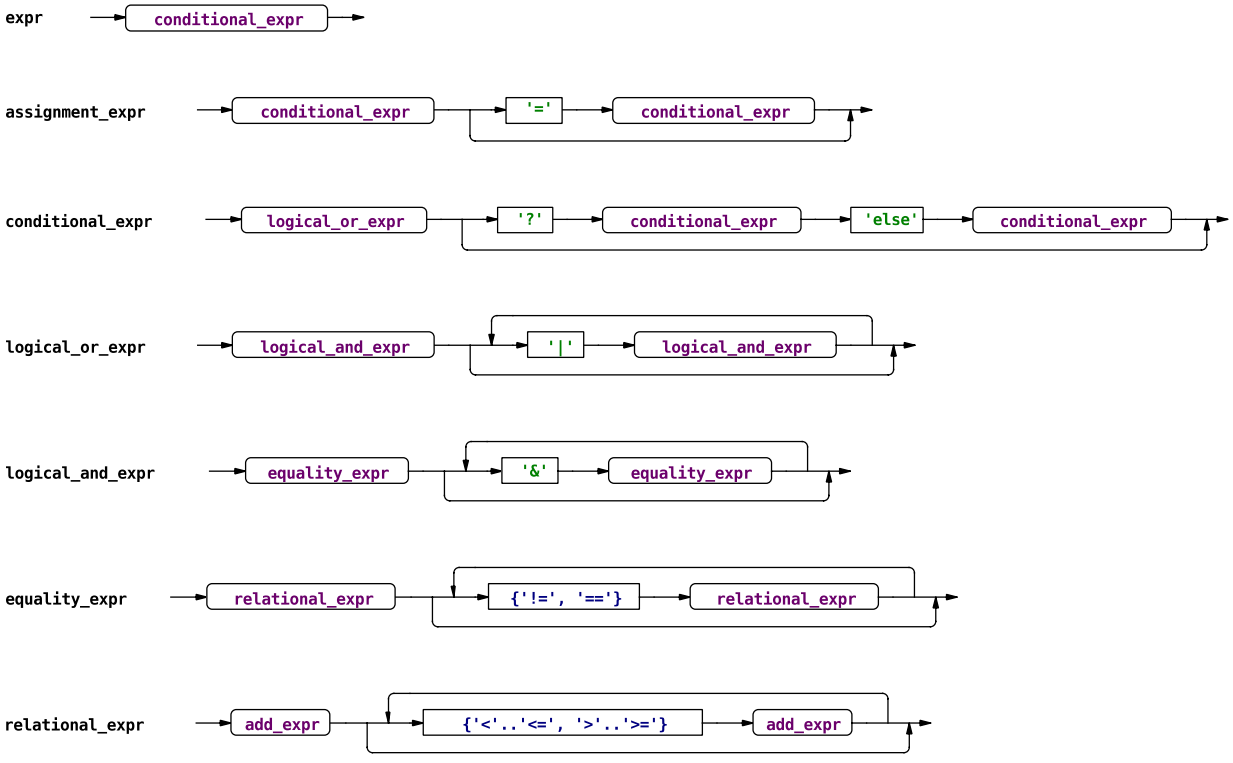


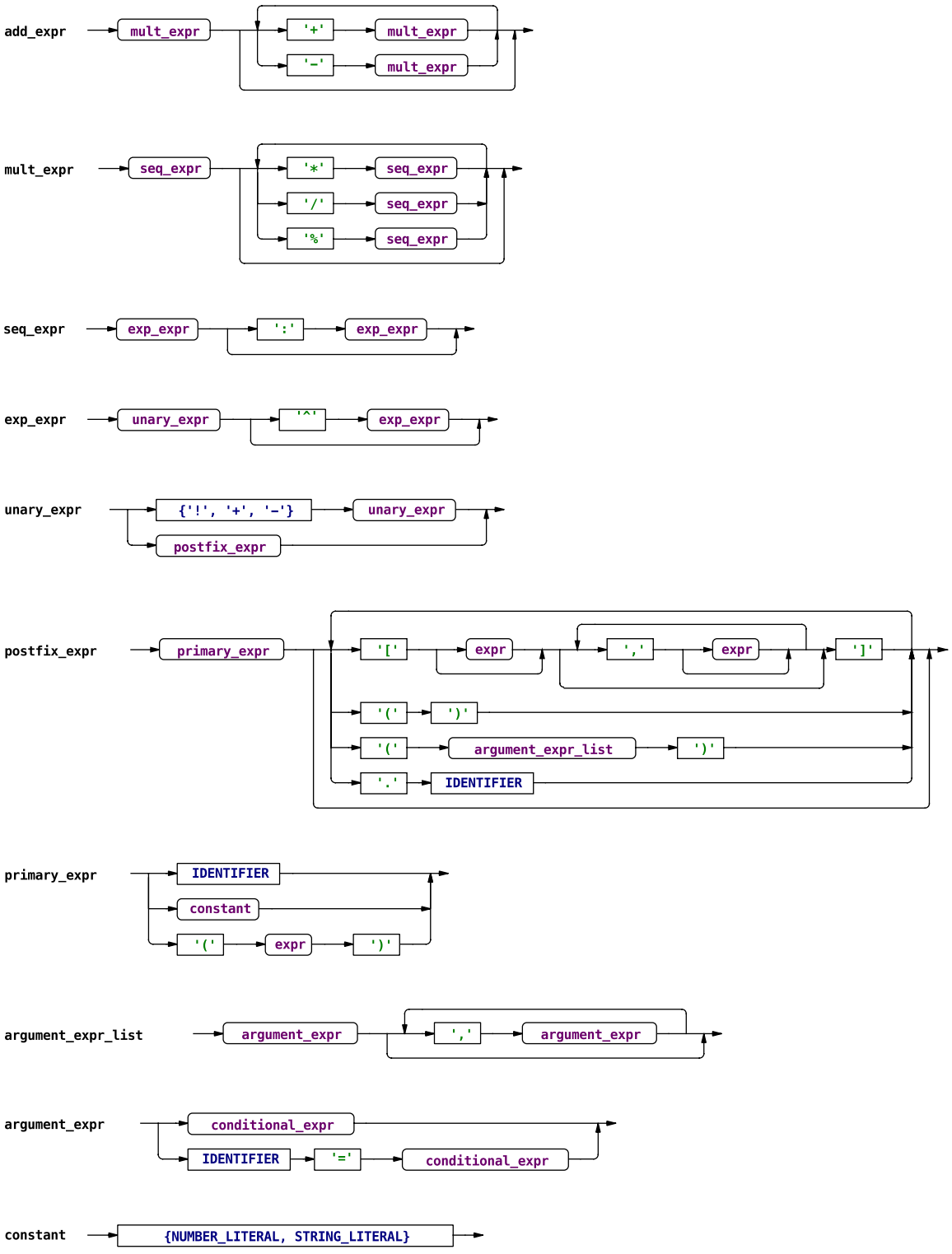


Compound statements are a series of zero or more statements enclosed by braces.

Statements are either compound statements, null statements (a lone semicolon), assignment-expressions (as defined in the next section), or language constructs beginning with one of the set of statement keywords defined by Eidos: `if`, `for`, `do`, `while`, `next`, `break`, or `return`. Note that function declarations are not a type of statement (see section 8.4); functions can only be declared at the top level in Eidos, not anywhere that a statement is allowed.

8.3 Expressions



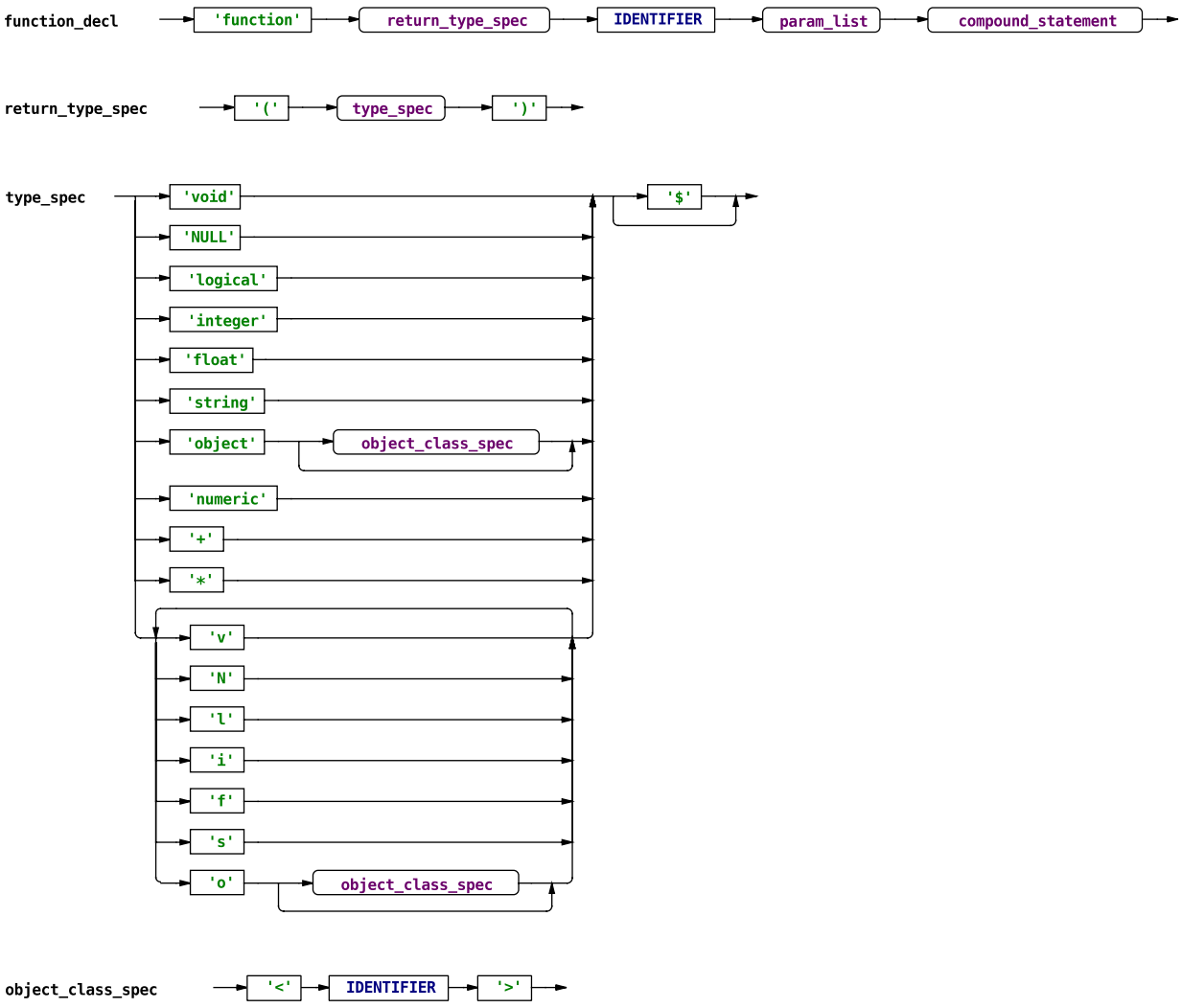


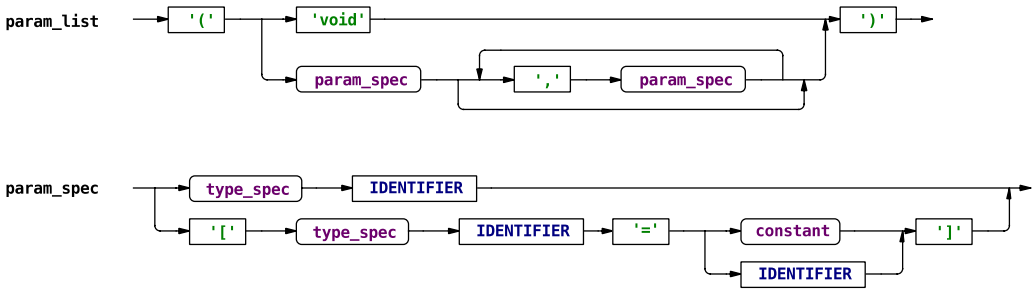
Expressions generally involve operators, and are defined in this grammar using a hierarchy of expression types that establish the precedence rules for those operators (see section 2.3.6), as in many languages. This hierarchy bottoms out at primary expressions, which are either an identifier (i.e., a variable name, in this case), a string or numeric constant, or a parenthesis-delimited subexpression.

Note that the left-hand side of an assignment statement is restricted at runtime to be an *lvalue expression*, an expression that identifies a specific set of defined values to be modified. For example, `x` is an lvalue expression since `x` is a specific identifier whose value can be (re)defined, but `x*y` is not. Since this restriction is checked at runtime, the grammar shown is correct, but it does not express this restriction.

Note also that assignment is not valid in Eidos expressions, in general; it is only allowed in the context of an `expr_statement`. Eidos differs from many other languages, such as C, in this respect; constructs like `if (x=y) ...` are not legal in Eidos, for safety and simplicity (see section 2.4.1). In Eidos 1.1 (SLiM 2.1) named function arguments were added, so `argument_expr_list` was modified, and `argument_expr` was added, to allow for this possibility; the use of the `=` symbol in this context is not assignment, however, and does not modify the values of any symbols (i.e., calling `abs(x=-10)` does not assign `-10` to `x`, it just designates `-10` as the value to be passed for the function argument named `x`).

8.4 Function declarations



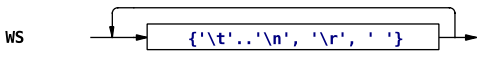


Beginning in Eidos version 1.5 (SLiM version 2.5), declaration of new user-defined functions is supported in Eidos (see chapter 4). The option of declaring a function has been added to the start rule (see section 8.1), and the rules above establish the form of such declarations. Note that, following the way that function prototypes are shown in Eidos, functions that return no value or that take no parameters are declared using the special type specifier `void`. Beginning in Eidos version 2.0 (SLiM version 3.0), `void` has been elevated to a formal type rather than simply a special language keyword, and so it is now integrated into `type_spec`; this change has few practical consequences, but does allow runtime type-checking to be more rigorous in some cases (see section 2.7.3).

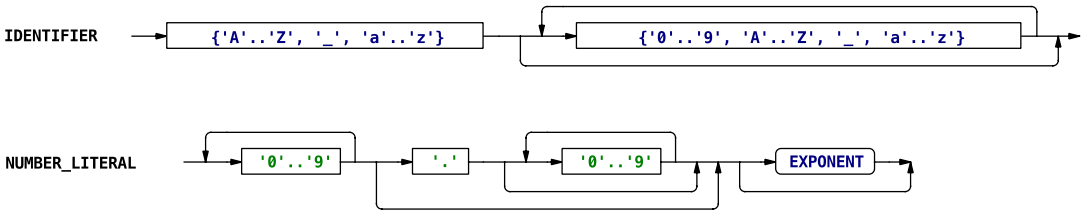
The only real complication here is the way types are specified in function declarations, for both the return type and the types of function parameters. These type specifiers may always be `*`, for maximum flexibility; this dispenses with almost all of the type-checking normally performed by Eidos. If a type is known more specifically, however, it is beneficial to provide that information in the function declaration; Eidos will then automatically type-check parameters and return values for you, and your code may then assume that all parameters and return values will conform to the types given in the function declaration. Note that a type specifier may be a simple base type (`logical`, `integer`, `float`, `string`, or `object`), one of the predefined compound types (`numeric` for either `integer` or `float`, `+` for any non-`void` base type except `object`, and `*` for any non-`void` base type including `object`), or a user-defined compound type given by a sequence of single letters specifying base types (`v`, `N`, `l`, `i`, `f`, `s`, or `o`), in any order. In the latter case, a given base type may not be specified more than once, and type-specifiers involving only `void` and `NULL` may not be declared singleton; these are semantic restrictions, not syntactic, and are thus not represented in this grammar. When an `object` type is allowed (except in the case of `*`), an optional class specifier may be supplied that gives a required class for the type when an `object` value is supplied; again, this allows Eidos to perform automatic type checking, so it should be specified when possible. Finally, if a value must always be a singleton (or `void` or `NULL`), that may be specified with a trailing `$`, which again allows Eidos to perform automated checks of parameter and return values – in this case, that those values are singletons when required to be.

Parameters in function declarations may be optional, specified by enclosing the parameter specification in brackets. In this case, a default value must be given; when the function is called, any unsupplied parameter will be defined as having its default value. Default values may be `string` or `numeric` constants, or may be an identifier designating a built-in Eidos constant (`T`, `F`, `NULL`, `E`, `PI`, `NAN`, or `INF`). At this time, user-defined constants, constant expressions, and expressions more generally are not allowed as default parameter values.

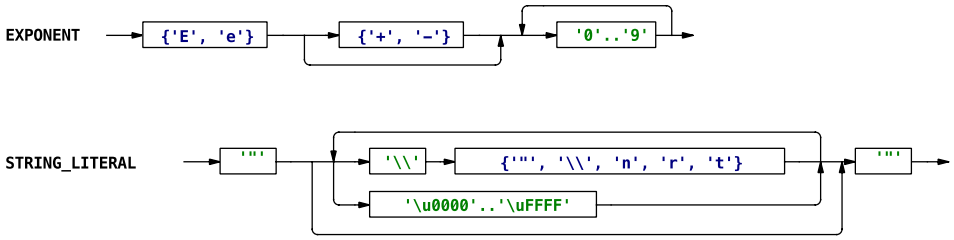
8.5 Tokens



Whitespace is not significant in Eidos, so these tokens are removed from the token stream. It is a bit odd that it is shown as `'\t'..'\n'`; in fact those characters are adjacent in ASCII.



Note that `integer` and `float` constants are not different token types; both tokenize and parse as numeric literals, and the distinction between them is made when the AST is interpreted. Upon interpretation, numbers with a decimal point or a negative exponent are taken to be `float`; all other numeric literals are taken to be `integer`. Note that this means that `integer` literals in Eidos may have a positive exponent.



This railroad diagram is not strictly correct. The lower path does not allow every possible Unicode character; backslash, quote, newline, and carriage return are excluded. Literal newlines and carriage returns are thus not legal within `string` literals; they must be escaped as `\n` and `\r`, respectively. Quotes and backslashes must also be escaped, but tabs may be included literally. Single-quoted `string` literals and multiline `string` literals are not shown in this railroad diagram; see section 2.1.4.

Comments are also tokens, technically, but they are removed from the token stream prior to parsing, and they are not readily depicted using railroad diagrams because of their unusual syntax (particularly nesting block comments). See section 2.10 for an overview of comments in Eidos.

9. The future of Eidos

It is not presently clear what the future holds for Eidos. The design of the language is pulled in two different directions at the moment. On the one hand, it would be wonderful to extend it in some obvious ways to make it more powerful and general. On the other hand, the radical simplicity of Eidos right now is both a design goal and a selling point; we want the language to be immediately approachable by people with no programming experience (biologists, in particular, in the case of SLiM). Every feature added to the language moves us away from that goal. Every little bit of new complexity has to be documented, and makes Eidos that much less approachable.

We have not yet decided on the precise balance we wish to strike; since the primary goal for Eidos is to provide scriptability to SLiM, a lot depends upon what we hear back from the users of SLiM. If you have feedback for us, either as a user of SLiM or as a user of Eidos in some other Context, we'd love to hear it.

10. Credits and licenses for incorporated software

Eidos incorporates code from various other software frameworks and packages. This section provides credit and license information for the software thus incorporated, as well as for Eidos itself.

10.1 Eidos

Eidos is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

Eidos is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with Eidos. If not, see http://www.gnu.org/licenses/.

10.2 GNU Scientific Library (GSL)

Eidos contains a modified distribution of the GNU Scientific Library (GSL), which is also licensed under the GNU General Public License under the same terms stated above. Thanks to the authors of the GSL for their very useful software, which can be found in full form at http://www.gnu.org/software/gsl/.

10.3 Boost

Eidos contains modified code from Boost (for a smart pointer implementation), which is licensed under the Boost Software License version 1.0 (http://www.boost.org/LICENSE_1_0.txt). The Boost Software License is compatible with the GPL, allowing its use here. Thanks to all of the authors of Boost for their very useful software, which can be downloaded in its full form from their website at http://www.boost.org/users/download/#live. Thanks especially to Peter Dimov, who appears to be the author of the class in question.

10.4 MT19937-64

Eidos contains a 64-bit Mersenne Twister implementation (a random number generator) by Takuji Nishimura and Makoto Matsumoto; thanks to them for this code, which was obtained from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/VERSIONS/C-LANG/mt19937-64.c. Following the terms of their license, the following text is provided from them:

```
Copyright (C) 2004, Makoto Matsumoto and Takuji Nishimura,

All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

1. Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.
```

```
3. The names of its contributors may not be used to endorse or promote
products derived from this software without specific prior written
permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

10.5 WiggleTools

Eidos contains code from WiggleTools to perform *t*-tests. WiggleTools is licensed under the Apache 2.0 license (http://www.apache.org/licenses/LICENSE-2.0), which is compatible with the GPL, allowing its use here. Thanks to EMBL-European Bioinformatics Institute for making this code available.

10.6 ObjectPool

Eidos contains a modified version of a C++ object pool class published by Paulo Zemek at http://www.codeproject.com/Articles/746630/O-Object-Pool-in-Cplusplus. His code is under the Code Project Open License (CPOL), available at http://www.codeproject.com/info/cpol10.aspx. The CPOL is not compatible with the GPL. Paulo Zemek has explicitly granted permission for his code to be used in Eidos and SLiM, and thus placed under the GPL as an alternative license. An email granting this permission has been archived and can be provided upon request. This code is therefore now under the same GPL license as the rest of Eidos, as stated above.

10.7 Exact summation

Eidos contains modified (translated to C from Python) code to compute the exact summation (within available precision limits) of a vector of floating-point numbers. This code is adapted from Python's `fsum()` function, as implemented in the file `mathmodule.c` in the `math_fsum()` C function, from Python version 3.6.2, downloaded from https://www.python.org/getit/source/. The authors of that code appear to be Raymond Hettinger and Mark Dickinson; thanks to them. The PSF open-source license for Python 3.6.2, which the PSF states is GSL-compatible, may be found on their website at https://docs.python.org/3.6/license.html.

10.8 Other contributions

Smaller snippets of code have been gleaned from the web, particularly from stackoverflow. Credit is given in the Eidos source code where this has occurred, and in all cases the code is licensed under terms compatible with the GSL (as far as we, who are not lawyers, can tell). Thanks to everyone whose code has found its way into Eidos.

11. References

Cox, B. J., & Novobilski, A. J. (1991). *Object-oriented programming: An evolutionary approach* [Second edition]. Reading, MA: Addison-Wesley.

Kernighan, B. W., & Ritchie, D. M. (1988). *The C programming language* [Second edition]. Englewood Cliffs, NJ: Prentice-Hall.

R Core Team (2014). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

Bovet, J., & Parr, T. (2008). ANTLRWorks: An ANTLR grammar development environment. *Software: Practice and Experience*, *38*(12), 1305-1332.

Parr, T. (2009). *Language Implementation Patterns: Create your own domain-specific and general programming languages*. Frisco, TX: Pragmatic Bookshelf.