Sxript Programming Language
MANUSCRIPT

William F. Barnes
[1]

February 15, 2025

[1]Copyright © 2014-2025 by William F. Barnes. All rights reserved. Unauthorized retention, duplication, distribution, or modification is not permitted.

Contents

1 Sxript Programming Language **3**
- 1 Hello World . . . 3
- 2 Numerical Operations . . . 4
  - 2.1 Arithmetic . . . 4
  - 2.2 Numeric Primitives . . . 5
- 3 Logical Operators . . . 6
- 4 Quote Operations . . . 7
  - 4.1 Special Characters . . . 8
  - 4.2 Textual Primitives . . . 9
- 5 Vectors . . . 9
  - 5.1 Vector Operations . . . 10
  - 5.2 Vector Primitives . . . 10
- 6 Mixed-Type Operations . . . 11
  - 6.1 Clerical Primitives . . . 12
- 7 Apply, Map, Reduce . . . 13
- 8 Variables . . . 14
- 9 Code Blocks . . . 15
  - 9.1 Subprograms . . . 16
- 10 Functions . . . 17
  - 10.1 Func Primitive . . . 17
  - 10.2 Multi-Line Functions . . . 18
  - 10.3 Anonymous Functions . . . 19
  - 10.4 Report Primitive . . . 20
  - 10.5 Special Operators . . . 20
- 11 Flow and Procedure . . . 20
  - 11.1 Iff Primitive . . . 20
  - 11.2 For Primitive . . . 21
  - 11.3 Do... Loop . . . 22
  - 11.4 Anchor... Goto . . . 24
  - 11.5 If... Goto . . . 24
- 12 Large Numbers . . . 25
- 13 ASCII Plotting . . . 29
  - 13.1 Graphing y=f(x) . . . 29
  - 13.2 Scatter Plot . . . 30
- 14 Functional Programming . . . 32
  - 14.1 Purity . . . 32
  - 14.2 High-Order Functions . . . 32
  - 14.3 Composition . . . 34
  - 14.4 Recursion . . . 35
  - 14.5 Impure Functions . . . 39
- 15 Structured Programs . . . 41
- 16 Applications . . . 43
  - 16.1 Calculus 101 . . . 43
  - 16.2 Complex Numbers . . . 44
  - 16.3 Linear Algebra . . . 46
- 17 Problems and Solutions . . . 48
  - 17.1 100 Doors . . . 48
  - 17.2 Babbage Problem . . . 49
  - 17.3 Coconut Pyramids . . . 50
  - 17.4 Golden Ratio . . . 51
  - 17.5 Haskell Tutorial . . . 52
  - 17.6 Koch Fractal . . . 53
  - 17.7 Nth . . . 54
  - 17.8 Pascal Triangle . . . 55
  - 17.9 Points on a Circle . . . 57
  - 17.10 Square Root Table . . . 58

Chapter 1

Sxript Programming Language

```
  .
  +@@@@,        @@
 @@@@@@@@'      @@@:
@@@@@@@@@      +@@@@@:
#    @@@@#     @@@@@@
     ,@@@@    .++@@@+
      @@@@#   @    @
      ;@@@@  ,@
       @@@@# @
       #@@@@'@
        @@@@@,
    '###@@@@@#######
    @@@@@@@@@@@@@@@.
   .@@@@@@@@@@@@@@@
    ``....@@@@#
         ,@@@@@
         @`@@@@#
        `@ @@@@@
        @;  @@@@'
        @   @@@@@
  @;   #+   `@@@@'
 ,@@@' @     @@@@@
 @@@@@@@     .@@@@; +
 @@@@@@       @@@@@@+
  +@@@@       .@@@@#
    @@.        #@@@
     `          `+
```

Sxript is an interpreted functional/procedural programming language designed for numerical evaluation and problem solving with high-order functions, variable storage, scoped subprograms, and arbitrary-precision numbers, along with file I/O, graphical output, and more. Sxript may be used as a stand-alone tool on most devices, or included as a component of any `BASIC`, `JavaScript`, or `C++` application. All source code is written in a strict subset of the `QB45` language, and is auto-translated by the `BAS-TO-X` utility into target languages.

1 Hello World

Successfully booting a stand-alone instance of `sxript`(.exe) in the console should produce something like the following:

```
*** Welcome to Sxript ***
      (QB64 Build)

[1]:
```

REPL Prompt

The core of any stand-alone instance is a typical *REPL* prompt, which means 'read-eval-print-loop'. In the same way you type commands into a terminal and press Enter to continue, the above prompt awaits a string you prepare, which is sent through the `SxriptEval` function when you press `Enter`. God willing, the output emerges on the next line.

Jokes aside, this means `Sxript` is an *expression-oriented* language, which means every piece of code return a value. There are no statements for toggling modes, settings, or otherwise 'doing something' in the computer. This is a text-in, text-out language.

First Session

Typing a few test statements into the prompt, a typical first session could be appear as in the code box that follows. Right away there are a few things that stand out.

```
[1]: hello to the world
[1]: hellototheworld

[2]: `hello to the world'
[2]: hello to the world

[3]: 1 + 1
[3]: +2.0

[4]: 3
[4]: 3
```

Whitespace

`Sxript` condenses and therefore ignores all unclaimed spaces, tabs, and line breaks. For instance, the input `hello to the world` is condensed down to a single word `hellototheworld` with all spaces removed. In isolation, a piece of ambiguous text such as `helloworld` or `sqrt` is formally called a *word*.

The same whitespace condensation, although hidden, turns `1 + 1` (five characters) into `1+1` (three characters) before attempting the addition.

Quotes

One way to preserve the integrity of text is to enclose the string in bracketing symbols as indicated on input `[2]` above. The left bracketing symbol is the back-tick key (to the left of 1, under `Esc`). The right bracketing character is the single quote (apostrophe).

As a proper quote, the input `‘hello to the world’` results in `hello to the world`, which is at the level of human readability. The bracketing symbols are stripped as the output reaches the console.

Double Quotes

The double quote symbol is not used or reserved in `Sxript` whatsoever. The symbol (`"`) can be used freely.

Numbers

For the third input we have supplied the arithmetic statement `1 + 1`, resulting in the third output `+2.0`. The result of a numerical operation is always formatted as a *formatted* number, which means to always include a sign prefix, either plus or minus, along with a decimal component, even for whole numbers.

Scientific notation is used for numbers large or small enough. Under the hood, number are represented by a floating-point variable type that behaves similarly among `JavaScript` and `C`-family languages.

Plain numbers like `3` are called *unformatted*, which work as regular numbers in an arithmetic sense.

Len() Primitive

The `len(x)` function, also known as the `len(x)` *primitive*, is a built-in function that takes an argument `x` and returns the length in characters of `x`.

When used on a quote, the length excludes enclosing quote symbols. When used a on a number, the length includes all sign and decimal information.

```
[4]: len(`hello world')
[4]: 11

[5]: len(+5.0)
[5]: 4

[6]: len(a b c d e)
[6]: 5
```

2 Numerical Operations

2.1 Arithmetic

The *arithmetic* operators are one-character symbols for exponents (`^`), multiplication (`*`), division (`/`), modulus (`%`), addition (`+`), subtraction (`-`) as follows:

```
^ * / % + -
```

Evaluation

A numerical evaluation is triggered when an operator is situated between two numbers, either formatted or non-formatted.

The result of a numerical evaluation is always a formatted number, i.e. includes a sign and a decimal. Expressions containing multiple operators are evaluated internally without showing each step. For instance:

```
[1]: 4 * 2 + 3
[1]: +11.0
```

Precedence

The order in which the operators are listed above also denotes their precedence of evaluation, reading left-to-right. No two operators have the same precedence.

When multiple instances of a given operator occur in an expression, evaluation proceeds in *right-to-left* order.

Since multiplication takes precedence over addition, the previous result can be recovered via:

```
[2]: 3 + 4 * 2
[2]: +11.0
```

Embedding

A way to subvert operator precedence is to enclose part(s) of an expression with parentheses. Priority is

passed to the most-embedded, most-left set of parentheses. The 'evaluation edge' passes recursively upward through layers of parentheses until finished.

```
[4]: 2*(3+6)/3
[4]: +6.0

[5]: 24/(3*(1+1))
[5]: +4.0
```

Precision and Magnitude

Depending on the host language's implementation of floating-point numerals, there are usually about fifteen digits of precision per number. Exact fractions like 1/3 are approximated via:

```
[6]: 1/3
[6]: +0.3333333333333333
```

For numbers requiring scientific notation, the suffix `D` (`QB64`) or `e` (`JavaScript`) occurs after the decimal portion:

```
[7]: 256^16            / QB64
[7]: +3.402823669209385D+38
```

```
[7]: 256^16            / JavaScript
[7]: +3.402823669209385e+38
```

The largest number allowed in a double-precision variable is around 10^{308}.

Negative Sign Duality

One unfortunate accident of history is the dual-use of the negative sign. On one hand, we've already reserved the dash symbol (`-`) for the subtraction operator. On the other hand, the same dash symbol is always used for negative numbers such as `-3.5` that do not invoke subtraction at all.

With this dilemma, `Sxript` adopts a rule where the negative sign prefers to bind to numbers (like an electron) to make them negative, *before* going between numbers to act as a subtraction operator.

Binding of the negative sign is demonstrated in the box below. By operator precedence, the exponent operator is treated first, which triggers a scan for candidate numbers surrounding the operator. On the right, we have `2`, whereas on the left, the scan captures `-3`. Since $(-3)^2$ is solved by `+9.0`, the expression simplifies to `0+9.0` or `+9.0`.

```
[8]: 0-3^2
[8]: +9.0
```

This example begs the question of how the result `-9.0` is attained. As a note of caution, one way that *won't* work is demonstrated box below. The result is `+9.0` again due to the parentheses taking precedence over any operators. Internally, the quantity `(3)` is evaluated to `3`, and then we're back on the path to getting `0+9.0`.

```
[9]: 0-(3)^2
[9]: +9.0
```

The result `-9.0` is attained by enclosing the term 3^2 in parentheses before subtracting from zero:

```
[10]: 0-(3^2)
[10]: -9.0
```

Operator Reduction

All input passes through an operator reduction filter prior to evaluation, which reduces things like `+++` to a single `+` and so on. Occurrences of `-+` are interpreted as `-`, making inputs such as `3-+3` evaluate to `+0.0`. Occurrences of a 'double negative' `--` are converted to a single positive `+`. For a mental exercise, check that the following expressions are consistent with your expectations:

```
[11]: 0---(-3^2)
[11]: -9.0

[12]: 0----(-3^2)
[12]: +9.0
```

2.2 Numeric Primitives

Recall that a primitive function is one that is hard-wired into the `Sxript` lexicon. Letting `x` represent an arbitrary number, we have the following primitives, in alphabetical order:

`abs(x)`
Removes the `+/-` sign from `x`.

`atan(x)`
Trigonometric arctangent of `x`.

`atan2(x,y)`
Cartesian arctangent of the polar angle formed between the positive x-axis and the point `x`, `y`.

`cos(x)`
Trigonometric cosine of `x`.

`eul(x)`
Euler exponential e^x.

`greater(x,y)`
Returns `+1.0` if `x` is greater than `y`. Returns `+0.0` otherwise.

`int(x)`
Convert `x` to integer.

`ln(x)`
Natural logarithm of `x > 0`. Returns an error otherwise.

`rand(x)`
Random number between zero and `x`.

`sgn(x)`
Returns `+1.0` if `x` is greater than zero. Returns `+0.0` otherwise.

`cos(x)`
Trigonometric sine of `x`.

`sqrt(x)`
Square root of `x > 0`. Returns an error otherwise.

`cos(x)`
Trigonometric tangent of `x`.

`unf(x)`
Removes number formatting from `x`.

3 Logical Operators

Following the numerical operators in precedence are the *logical* operators for `inversion` (`!`), `equality` (`=`), the logical `and` (`&`), and the logical `or` (`|`):

```
! = & |
```

Like the arithmetic operators, the logical operators situate between any two numbers *or other types*. However, the logical operators only emit binary output, which is to say `0` or `1`.

Inversion

The `inversion` operation takes a single argument to the left of the (`!`) symbol. If the argument is not zero, then `0` is returned. Otherwise, `1` is returned as shown:

```
[1]: -3!
[1]: 0

[2]: 0!
[2]: 1

[3]: (2+3)!
[3]: 0
```

Equality

As the name suggests, the logical `equality` operator returns `1` when situated between two equivalent numbers. Otherwise, the result is `0`. Use of the equality operator is straightforward, however things get interesting when all available floating-point digits are used as hinted below:

```
[4]: 1/3 = +0.333
[4]: 0

[5]: 1/3 = +0.3333333333333333
[5]: 1
```

The fraction $1/3$ is equivalent to $0.33\overline{3}$ where the string of 3's runs infinitely. The best a floating-point number can do is store about fifteen of the trailing digits, and then no real distinction exists between the fraction and its decimal equivalent.

And, Or

The logical `and` returns `1` if both inputs are greater than zero. The logical `or` returns `1` when at least one of the inputs is greater than zero. Otherwise, each output is `0`. Some trivial use cases are illustrated below:

```
[6]: 1 & (3-1)
[6]: 1

[7]: (4-2*2) | 0
[7]: 0

[8]: 4 + 3|0
[8]: 1
```

Numeric Comparison

There is no operator notation for numeric comparison with the angle brackets `<`, `>`, as these symbols are reserved for enclosing vectors. Instead the `greater` primitive (function gets) the job done by taking the two numbers as function parameters. Then, the `greater` primitive only returns `+1.0` if the first argument is greater than the second.

```
[9]: greater(5,6)
[9]: 0

[10]: greater(5,0)
[10]: 1

[11]: greater(5,5)
[11]: 0
```

Of course, we have an effective `lesser` by tacking on the inversion operator at the end, reversing each output:

```
[12]: (greater(5,6))!
[12]: 1

[13]: (greater(5,0))!
[13]: 0

[14]: (greater(5,5))!
[14]: 1
```

4 Quote Operations

In `Sxript`, the 'string' data type is called a *quote*. A quote is a sequence of characters enclosed on the left by the back-tick symbol (`), and enclosed on the right by an apostrophe (´).

Formatted Output

Before reaching the terminal or output file, the enclosiung characters (`´) are erased from any quotes. For example, the qwuote `'Hello'` is seven characters in the string space of the host language, however we want the length to resolve to five characters, which is indeed the case when using the `len()` primitive:

```
[1]: `Hello'
[1]: Hello

[2]: len(`Hello')
[2]: 5
```

Note that when two quotes are juxstaposed, as in the example `'Hello''World'`, the internal behavior is *undefined*. Such an input formats to `HelloWorld` as expected, with all enclosing chracters hidden, however the length of the resulting quote is not ten, but instead twelve. The back-to-back enclosing characters are still counted. There is no reason to do this.

```
[3]: len(`Hello'`World')
[3]: 12
```

Operator Overloading

Quotes can be manipulated using the same operators `*`, `/`, `+`, `-` that occur in numerical evaluation, a typical move called *operator overloading*.

The way an operator is used should be evident from context, but this isn't a guarantee. For instance, we know that the expression `5+6` is undoubtedly mathematical, whereas `'hello ' + 'world'` is the concatenation of quotes.

Concatenation

Proper quote concatenation is done with the addition operator placed between quotes, which evaluates to a single quote.

```
[4]: `Hello ' + `World'
[4]: Hello World

[5]: len(`Hello ' + `World')
[5]: 11

[6]: `do ' + `re ' + `mi'
[6]: do re mi
```

Subtraction

The tail end of a quote can be subtracted off using the negation operator. If the second argument matches the right-most characters of the first argument, then the first quote minus truncated before the second quote is returned.

```
[7]: `file.txt' - `.txt'
[7]: file

[8]: len(`' - `xyz')
[8]: 0
```

Multiplication

When the multiplication operator occurs between quotes, the result has a copy of the second argument inserted after each character of the first argument.

```
[9]: `abc' * `xy'
[9]: axybxycxy

[10]: `abcde...' * ` '
[10]: a b c d e . . .
```

Division

When the division operator occurs between quotes, the result has the first argument with each instance of the second argument removed.

```
[11]: `axybxycxy' / `xy'
[11]: abc

[12]: `abc' / `xy' * `xy'
[12]: abc
```

Binary Logic

Binary logical operator overloading extends to quotes. Any non-empty quote (‘’) is analogous to a zero-input, and any non-empty quote is considered non-zero.

```
[13]: `race' + `car' = `racecar'
[13]: 1

[14]: `race' & `car'
[14]: 1

[15]: `' | `car'
[15]: 1

[16]: `' & `'
[16]: 0
```

4.1 Special Characters

Several special ASCII characters, namely the `tab` (9), `line feed` (10), and `carriage return` (13) are given special treatmeant because `Sxript` normally ignores or eats such whitespace entities. It's also necessary to have a way to include the backtick (‘’) and apostrophe (’) characters in the body of a formatted quote.

Tab

To have a `tab` inside a quote, use the backslash-t combination as shown below. Note this also works outside of the quote environment as well.

```
[1]: `Hello\tWorld'
[1]: Hello      World

[2]: Hello\tWorld
[2]: Hello      World
```

Backtick and Apostrophe

In order to have a backtick or apostrophe appear within the body of a quote, precede these characters with a backslash.

```
[3]: `Here\'s a \`quote\''
[3]: Here's a `quote'
```

Line Feed, Carriage Return

The ‘line ending characters’ LF and CR, respectively, are produced with backslash-n and backslash-r. The behavior of these can vary based on implementation. Using a `QB64` implementation, the line feed and carriage return characters operate as shown:

```
[1]: `Hello\nWorld'
[1]: Hello
World

[2]: `Hello\rWorld'
WorldHello
```

Observe how the carriage return in line `[2]` moves the cursor to the beginning of the quote and bumps all characters to the right to accomodate `World` on the left.

Note also that the matching `[2]` notation on the output line has vanished. This is a case where `Sxript` has an effect on its hosting environment, and certain

information has been invisible 'scrolled upward' before hitting the terminal.

Different behaviors emerge from `JavaScript` and `C++` implementations of `Sxript` under the same test.

Escape Character

Formally, the backslash is called the *escape character*. When encountered, `Sxript` treats the symbol immediately following the escape character as a special case.

In order to have a backslash show up on purpose, use two of them, unless at the end of a quote. For this, a special construct is used, as the enclosing apostrophe needs to be shielded.

```
[1]: `Careful with \\ slashes'
[1]: Careful with \ slashes

[2]: `Careful with \\ slashes\\{\}'
[2]: Careful with \ slashes\
```

Null

For some situations, there is reason to return nothing - which really means nothing - not an empty quote, not an empty set of parentheses. For this we have that when the internal result of an expression is {`null`}, such is erased.

```
[3]: {null}
[3]:
```

4.2 Textual Primitives

Let `c` represent any single-character quote, and let `x` represent a quote of arbitrary length. Also let any instance of `n` denote an integer. The primitives for dealing with quotes are as follows:

`asc(c)`
Integer ASCII code of `c`.

`chr(n)`
Character matching the ASCII code of `n`.

`instr(x,y)`
Return `n` if quote `x` contains quote `y` at starting at position `n`. Returns `+0.0` otherwise.

`lcase(x)`
Replace all uppercase characters in `x` with lowercase.

`left(x,n)`
Return the first `n` characters in quote `x`.

`len(x)`
Return the number of characters in quote `x`.

`mid(x,n1,n2)`
Return a subquote of `x` with length `n2` starting at position `n1` .

`quote(x)`
Enclose `x` in bracketing symbols, i.e. '`x`'.

`right(x,n)`
Return the last `n` characters in quote `x`.

`lcase(x)`
Replace all lowercase characters in `x` with uppercase.

`unquote(x)`
Remove bracketing symbols from `x`.

5 Vectors

A *vector* is a data structure for keeping numbers, quotes, or other structures in a linear list, somewhat like an array.

Vector Anatomy

A vector is enclosed by left- (`<`) and right-angle brackets (`>`). Each member in the vector or list is called an *element*, and each element is separated by a comma: (`,`)

```
[1]: <1,2,1+2,`Computer'>
[1]: <1,2,+3.0,Computer>

[2]: <a, b, c, d, e>
[2]: <a,b,c,d,e>

[3]: <`hello',`world'>
[3]: <hello,world>
```

Vectors can contain data of mixed types, including other vectors.

```
[4]: <1,2,3,<a,b,c,d,e,f>,<4,5,6>>
```

Evaluation

By default, the elements in a vector are evaluated in left-to-right order. However, the 'deepest and left-most sub-expression' rule always applies, meaning the most-embedded content in the whole vector is always simplified first.

Note that outside the console environment where pressing `Enter` doesn't trigger an evaluation, as in a text file, the above vector can be written with any configuration of whitespace for added clarity. For instance:

```
<1,2,3,
  <a,b,c,d,e,f>,
<4,5,6>>
```

Vector Length

The `len()` primitive operating on a vector returns the number of elements in the vector. The 'empty' vector `<>` contains one element, and it is undefined for a vector to have zero elements.

```
[5]: len(<a, b, c, d, e>)
[5]: 5

[6]: len(<1,2,3,<a,b,c>,<4,5,6>>)
[6]: 5

[7]: len(<>)
[7]: 1
```

Matrix

If each element of a vector is itself a vector with a fixed number of elements, such a structure is a *matrix*. A matrix is a rectangular list of lists.

```
<
  <a,b,c>,
  <d,e,f>,
  <g,h,i>
>
```

5.1 Vector Operations

The overloading of operators `*`, `/`, `+`, `-` extends to vectors of the same length. Internally, the operator is situated between corresponding vector components and the result is a vector of the same length as the inputs.

For instance, the expression `<1,2,3>+<4,5,6>` is first transformed to `<1+4,2+5,3+6>`, which becomes `<+5.0,+7.0,+9.0>`. Had the operator been multiplication, division, etc., we would get:

```
[1]: <1,2,3> * <4,5,6>
[1]: <+4.0,+10.0,+18.0>

[2]: <1,2,3> / <4,5,6>
[2]: <+0.25,+0.4,+0.5>

[3]: <1,2,3> ^ <4,5,6>
[3]: <+1.0,+32.0,+729.0>

[4]: <`hello ', `world'> * <`!',`#'>
[4]: <h!e!l!l!o! !,w#o#r#l#d#>
```

Vector operations apply recursively into embedded vectors. For a less trivial example, one could have:

```
[5]:
<<1,2>,
  <3,<9,0>,`Toyota Camry'>>
-
<<6,7>,
  <8,<9,10>,`Camry'>>

[5]:
<<-5.0,-5.0>,
  <-5.0,<+0.0,-10.0>,Toyota >>
```

5.2 Vector Primitives

The primitives for dealing with vectors are as follows:

`column(x,n)`
Return the `n`th column (as a vector) of matrix `x`.

`elem(x,n)`
Return the `n`th element of vector `x`.

`left(x,n)`
Return a subvector of the first `n` elements of vector `x`.

`len(x)`
Return the number of elements in vector `x`.

`mid(x,n1,n2)`
Return the subvector with `n2` elements starting al element number `n1` in vector `x`.

`replace(x,n,y)`
Replace the `n`th element in vector `x` with `y`.

`right(x,n)`
Return a subvector of the last `n` elements of vector `x`.

`smooth(x)`
Remove all empty elements from vector `x`.

`stack(x,y)`
Append vector `y` to the right of vector `x`.

`unvector(x)`
Remove the bracketing symbols (`<>`) from vector `x`.

`vector(x)`
Enclose `x` in bracketing symbols (`<>`) to return a vector.

6 Mixed-Type Operations

As we've seen with quote operations and vector operations, the arithmetic and comparison syntax

```
^ * / % + - ! = & |
```

also extends to *mixed-type* operations. This is to say that numbers, quotes, and vectors can be musually added, multiplied, etc. to yield new structues.

Quote-Number Addition

The simplest mixed-type operation is number-quote addition (`+`), which works much like concatenation. The result of such an operation always yields a proper quote, never a number. The same rules hold for the commuted scenario of quote-number addition.

```
[1]: (1 + 2) + ` Three'
[1]: +3.0 Three

[2]: 1 + (2 + ` Three')
[2]: 12 Three

[3]: 1 + 2 + ` Three'
[3]: 1+2 Three

[4]: `Four ' + 2 + 2
[4]: Four 4.0
```

To make sense of the above, recall that evaluation goes right to left when identical operators occur in the samne expression.

Quote-Number Multiplication

An exotic number-quote interaction, or to be more precise, whole number-quote interaction, is multiplication (`*`). Multiplying an integer `n` into a quote `c` results in a quote that is `n` copies of `c` juxtaposed.

```
[5]: 3 * `Hello '
[5]: Hello Hello Hello

[6]: `World ' * 2
[6]: World World
```

Quote-number multiplication is commutative, which means swapping the arguments leaves the result unchanged. Note that multiplying any quote by zero results in an empty quote.

```
[7]: 0 * `Hello'
[7]:

[8]: type(0 * `Hello')
[8]: quote

[9]: len(0 * `Hello')
[9]: 0
```

Distribution into Vector

An operator situated between a vector and a number or quote results in a modified vector of the same size where the outer content is distributed with the operator into each component. Distribution is non-commuting, which is to say the outer content will occur left- or right-distributed based on whether the operater occurs to the left or to the right of the vector.

```
[7]: <`file',`notes',`book'> + `.txt'
[7]: <file.txt,notes.txt,book.txt>

[8]: 2 * <1, `Two ', <2, `One '>>
[8]: <+2.0,Two Two ,<+4.0,One One >>
```

Undefined Operations

Certain combinations of data types and operators yield undefined results. For instance, if we try the equality (=) operator between a number and a quote, no operation takes place and we instead get a formatted version of the input.

```
[9]: `Apple'=7
[9]: Apple=7

[10]: len(7=`Apple')
[10]: 9
```

To reinforce that something like `7=‘Apple’` should not occur, observe that the `len` primitive around this structure includes the operator and the quote bracketing symbols.

Summary

A summary of allowed operations is listed in the Table below. The diagonal entries correspond to same-type operations with numbers, quotes, and vectors. Off-diagonal entries are mixed-type, and indicate to which operations take place as described above.

Note that the `word` type doesn't interact with the others or with itself, with the exception of the equality (=) operation.

		right			
		number	quote	vector	word
left	number	Numeric Operations	* +	right dist.	
	quote	* +	Quote Operations	right dist.	
	vector	left dist.	left dist.	Vector Operations	
	word				=

Table 1.1: Allowed type operations.

6.1 Clerical Primitives

Seeral language primitives that serve clerical purposes are reviewed here, namely `type`, `join`, and `identity`, from most- to least-complicated.

Type Primitive

The `type` primitive is a single-argument function that takes any data structure as an argument. The return from `type` is a quote containing one of: `number`, `quote`, `vector`, `word`, `occult`, `symbol`, `paren`.

Note that there is no type for ‘operator’, which is to say a type check on the plus sign (+) is different from a check on the asterisk (*).

The following expressions return `number`:

```
[1]: type(2*3)
[2]: type(+-)
[3]: type(7Apple)
```

The following expressions return `quote`:

```
[4]: type(`Hello' + ` World')
[5]: type(`')
[6]: type(type(1))
```

The following expressions return `vector`:

```
[7]: type(<a,b,c,1,2,3>)
[8]: type(<<1,2>,<3,4>>)
[9]: type(<>)
```

The following expressions return `word`:

```
[10]: type(Hello)
[11]: type(len)
[12]: type(*)
```

For completeness, the three thus-far unmentioned outputs from `type` are extracted as follows:

```
[13]: type({})
[13]: occult

[14]: type([])
[14]: symbol

[15]: type()
[15]: paren
```

Join Primitive

The `join` primitive is a two-argument function whose purpose is to juxtapose the two arguments as if the programmer typed them without using `join`.

```
[1]: join(abc,123)
[1]: abc123

[2]: join(5,5)
[2]: 55

[3]: join(1+1, 2+2)
[3]: +6.0
```

Note that the juxtaposition of two formatted numbers induces addition or subtraction between the two.

Identity Primitive

The most benign primitive is the `identity`, which takes a single argument and returns the argument unchanged. Of course, the output of this function is still caught in the evaluation loop and is rendered down to a stable result as if `identity` was never there.

```
[1]: identity(`hello world')
[1]: hello world

[2]: identity(1+1)
[2]: +2.0
```

Date and Time

`Sxript` embeds two special primitives for reporting the system date and the system time. Not surprisingly, these are called via `date()` and `time()`, respectively:

```
[1]: date()
[1]: 1-8-2025

[2]: time()
[2]: 22:4:15
```

Note the return from each is formally a quote, however the contents is a mixture of numbers and symbols.

7 Apply, Map, Reduce

Here we introduce the `apply`, `map`, `reduce` primitives, each being a collection-controlled loop maneuver for using functions with vectors.

Apply Primitive

Consider a vector `v` with components `v1`, `v2`, ..., `vN`, where `N` is the length of the vector. The `apply` primitive takes a function name `f`, along with the vector `v` and returns a new vector `<f(v1), f(v2), ..., f(vN)>`.

```
[1]: apply(sqrt,<9,16,25,121>)
[1]: <3,4,5,11>

[2]: apply(len,<`dog',`cat',`mouse'>)
[2]: <3,3,5>

[3]: apply(type,<1+1,`Hello',type>)
[3]: <number,quote,word>
```

Map Primitive

Consider two vectors `v` and `w` of equal length `N`. For a given two-argument function `f`, the `map` primitive produces a new vector `<f(v1,w1), f(v2,w2), ..., f(vN,wN)>`.

```
[1]: map(<10,5,3>,greater,<6,5,4>)
[1]: <1,0,0>

[2]: map(<`dog',`cat',`mouse'>,
        instr,<`o',`z',`u'>)
[2]: <2,0,3>

[3]: map(<1,2,3>,join,<4,5,6>)
[3]: <14,25,36>

[4]: map(<1,2,3>,join,<4,5,6>+0)
[4]: <+5.0,+7.0,+9.0>
```

Reduce Primitive

The `reduce` primitive uses a two-argument function `f` to collapse a vector `v` of length `N` down to a single value via `f(...f(f(f(v1,v2),v3),v4),...,vN)`.

```
[1]: reduce(join,<a,b,c,d,e>)
[1]: abcde

[2]: reduce(join,<1,2,3+0,4,5>)
[2]: +15.045

[3]: reduce(join,<5,*,4,*,3,*,2,*,1>)
[3]: +120.0

[4]: reduce(greater,<4,3,0,-2>)
[4]: 1
```

Application: Vector Dot Product

The vector 'dot product' i.e. scalar product, is the projection of one vector `v` onto another vector `w` of the same length `N`. In mathematical notation, one would write

$$\vec{v} \cdot \vec{w} = v_1 w_1 + v_2 w_2 + \cdots + v_N w_N \ .$$

Using `sxript` apparatus, it turns out that the products $v_1 w_1$, $v_2 w_2$, etc. are naturally handled by vector operations. Taking a three-dimensional example, we already have:

```
[1]: <1,2,3> * <4,5,6>
[1]: <+4.0,+10.0,+18.0>
```

Explicitly, the dot prouduct between the vectors `<1,2,3>` and `<4,5,6>` is the sum of the elements in `<+4.0,+10.0,+18.0>`. Given that each of these is a formatted number, it follows that all we effectively need to do is remove the commas and bracketing symbols to give `+4.0 +10.0 +18.0`, which is valid input for numerical evaluation. This is done using the `reduce` primitive with `join` as the working function.

```
[2]: reduce(join,<1,2,3> * <4,5,6>)
[2]: +32.0
```

8 Variables

Like most programming languages, a chunk of data can be stored in a *variable*. After being stored, the content of a variable is referenced through the corresponding variable name.

Let Primitive

The syntax for variable storage is `let(name,x)`, where `name` is a word representing the variable name and x stands for any valid data structure. The return from the `let` primitive is `x` fully simplified.

For instance, the expression `let(a,1+2)` is first reduced to `let(a,+3.0)`, thus the variable `a` holds `+3.0`. Note that variable storage follows *eager evaluation*, which is to say `1+2` is not stored, rather `+3.0` is computed and stored

A variable name can be any word, with a short list of exceptions. To prevent interference with user-defined functions, the letters `x`, `y`, `z`, `t`, `u`, `v` are reserved.

Restoration

A stored variable is recalled by enclosing its name in square brackets (`[]`). When embedded in an expression, variable restoration is given priority over function evaluation and mathematical evaluation. By the time the information in a variable 'hits' a function or operator, the data is already in literal form, as if never stored in a variable.

Mutability

A variable cannot be 'edited' in the sense that there are no specialized operators `++`, `+=`, etc., as found in other languages. What is allowed however is to use `let` to overwrite new content into a previously-used variable name.

Symbol Type

Formally, any structure enclosed in square brackets (`[]`) is a *symbol*. There is a `symbol` primitive designed for this in case manually enclosing a word in square brackets is disadvantageous.

```
[1]: let(a,1+2)
[1]: +3.0

[2]: [a] + cos([a])
[2]: +2.010007503399555

[3]: let(a,[a]-1)
[3]: +2.0

[4]: symbol(a)
[4]: +2.0
```

Corollaries

Sine the output of the `let` primitive is its stored content, it folows that variable storage can be used in-line with compatible operations. For instance,

`4+let(a,2)` outputs `+6.0` after storing `3` in the name `a`.

As an ordinary primitive, `let` can be embeded within other primitives, including another `let`. For instance, `let(b,4+let(a,3))` will first store `3` in `a`, simplifying to `let(b,+7.0)`.

```
[1]: 4+let(a,2)
[1]: +6.0

[2]: let(b,4+let(a,3))
[2]: +7.0

[3]: <[a],[b]>
[3]: <3,+7.0>
```

Toward Functional Programming

It should be emphasized that variable storage works not only on the standard data structures, but also *words*, indluding the names of functions.

```
[1]: let(f,cos)
[1]: cos

[2]: let(g,sin)
[2]: sin

[3]: [f]([g](3))
[3]: .9900590857598653

[4]: cos(sin(3))
[4]: .9900590857598653
```

9 Code Blocks

A *code block* provides a way to contain several lines of `Sxript` code in a single entity.

A code block begins with the word `block`, followed by a set of curled braces enclosed in parentheses `({})`. Inside this structure, we write a sequence of epressions with each separated by the colon symbol `(:)`. At least one of the expressions must be prefixed by `print_`, which is considered the output of the code block.

Code blocks do not have private scope, which is to say that any variables defined inside the block are accessible after the block is used. Similarly, any variables defined outside the block can be used within.

```
[1]: block({let(a,5):print_[a]+1})
[1]: +6.0

[2]: [a]
[2]: 5

[3]: block({let(b,3):print_[a]+[b]})
[3]: +8.0
```

Occult Type

Any structure enclosed in curled braces ({}) is called `occult`, which is yet another creature in the zoo of data types.

Occulted code is 'protected' from evaluation until triggered by `block` or similar primitive. This means occulted code can be stored in a variable and recalled for later use.

```
[4]: let(b,{let(a,5):print_[a]+1})
[4]: {let(a,5):print_[a]+1}

[5]: [b]
[5]: {let(a,5):print_[a]+1}

[6]: block([b])
[6]: +6.0
```

The language primitives `elem`, `left`, `len`, `mid`, `replace`, `right`, `smooth`, and `stack` defined for vectors also work on occulted structures. There are two new primitives, namely `occult` and `unoccult`, whose purpose is self-explanatory.

Print

Observing that `block` it itself a language primitive, anything following `print_` is considered part of the output. There can be more than one `print_` statement per code block, but there should never be zero. The simplest return from a code block is a set of empty parentheses:

```
[7]: block({print_()})
[7]: ()
```

Some caution must be used when using more than one `print_` statement, as each output is joined with the next without disgression. You are not, for instance, automatically handed a vector.

To illustrate, puzzle out why the following code block returns `+8.0`:

```
block({
  print_1+1:
  2+2:
  print_3+3
})
```

Evaluation

Occulted code is evaluated in strictly linear order, which is to say an expression is completely processed before evaluation moves beyond the colon (:) sepatator.

Some subtleties still lurk, however. Consider a code block that references a variable a before it is stored:

```
block({
  let(b,[a]+1):print_[b],:
  let(a,5)    :print_[b]
})
```

In the above, note the liberal use of whitespace (the evaluator doesn't care) with the particular use of punctuation. This block is designed to emit two numbers separated by a comma, which is by itself bad form. It would more appropriate to enclose the whole structure in angle brackets to make a vector:

```
<block({
  let(b,[a]+1):print_[b],:
  let(a,5)    :print_[b]
})>
```

Since the symbol [a] is referenced before it is defined via let, it can't possibly happen that b stores the number +6.0, rather we expect [a]+1 to occur at least once. However, on evaluation, the above gives:

```
<+6.0,+6.0>
```

We could have predicted the pair of +6.0's by realizing that the variable a is defined outside the scope of the code block, i.e. a *global* variable. The output of the code block is truly [a]+1,+6.0, however such a structure is still caught in the evaluation cycle, and 5 is filled in for [a] before the final answer appears.

One can check this by printing occulted code instead of the 'raw' symbol for b:

```
<block({
  let(b,[a]+1):print_occult([b]),:
  let(a,5)    :print_occult([b])
})>
```

Wrapping each instance of [b] accordingly we see a modified output there the first reference to [a] is protected as occulted code:

```
<{[a]+1},{+6.0}>
```

The unoccult primitive can be used to dig out the vector contents and get the pair of +6.0's again:

```
apply(unoccult,<{[a]+1},{+6.0}>)
```

Embdedding

The block primitive can be used to return occulted code. For a minimal example, consider the following structure:

```
block({
  print_{print_`Hello World'}
})
```

Such a code block returns the occulted structure {print_Hello World}. Note that the output is formatted, which means there are no enclosing symbols around the embedded quote. Let this be a caution that copying code directly from the terminal, as it often does, comes with hazard.

In order to access the embedded occulted code, surround the entire code block with another block primitive as shown:

```
block(
  block({
    print_{print_`Hello World'}
  })
)
```

Note that the outer block does not enclose its content directly in curled braces ({}). The formatted result is Hello World, and running the entire structure above through the type primitive returns quote as expected.

9.1 Subprograms

There is a variation on the code block called the *subprogram* that makes use of privately-scoped variables.

A subprogram has access to all variables and data in its environment, but can only send information back to the environment via the print_ statement. All else is forgotten when the subprogram ends.

A subprogram is initiated with the sub primitive. Right away, we can test the tricky case reviewed previously with sub replacing block. Fully simplified, the following structure reduces to <[a]+1,+6.0>.

```
<sub({
  let(b,[a]+1):print_[b],:
  let(a,5)    :print_[b]
})>
```

For a less trivial example, consider the following code block with three variables a, b, c.

```
block({
  let(a,1):
  let(c,
    sub({
      let(b,2):
      let(a,[a]+[b]):
      print_[a]
    })
  ):
  print_<[a],[b],[c]>
})
```

Working from the inside out, we see a subprogram that emits whatever value [a] represents, and this is stored in the variable c via the enclosing let. Within the subprogram the valriable b is defined, but notice there is no definition for b outside the subprogram. When the final print_ is encountered, we have variables a and c properly filled, but b remains empty. The final output of the code block is <1,[b],+3.0>.

10 Functions

Subroutines

Combining the ability to store variables via let with the notion of storing occulted code that is activated using block or sub, one can imagine building custom subroutines or functions.

For a trivial example, consider the 'function' that adds one to the variable a and returns the result:

```
let(plusone,{print_[a]+1})
```

Executing this mentally, we have a variable plusone storing one line of occulted code. This means any instance of [plusone] is replaced accordingly, which needs either block or sub to finish evluating. For this it's convenient to define another variable b to save some typing, where any instance of [b] is replaced by the contents of a plus one as defined:

```
let(b,sub([plusone]))
```

Of course, this is a clunky way to program, particularly because the variable a needs to be hard-coded into both the environment and the occulted line(s). What we've got so far is a subroutine at best.

10.1 Func Primitive

A user-defined function, or UDF, is constructed with the func primitive. Analogous to variables, any function consists of a name associated with stored data. The function body must occur in the form of occulted code:

```
func(newUDF,{...})
```

As a language primitive, the return from func is the function name itself, which occurs as a word. Like variables, any UDF defined in a subprogram is forgotten when the subprogram closes.

Once defined, a UDF is referenced like any other function or language primitive, which is to say we write the name of the function immediately followed by any required arguments enclosed in parentheses.

Arguments sent to a UDF are referenced within the body as parameters using the symbols [x], [y], [z], [t], [u], [v], and these are the only symbols allowed. If there is need to send more than six arguments, one must pack some of them into a vector or comparable data structure, all of which fits into a single symbol such as [x]. (One could get by using *only* [x].)

Single-Line Functions

The simplest UDF has a single line as its function body. For instance, one could write and test a set of functions for converting temperature units between Fahrenheit and Celsius:

```
[1]: func(degC,{(5/9)*([x]-32)})
[1]: degC

[2]: func(degF,{(9/5)*[x]+32})
[2]: degF

[3]: degC(degF(0))
[3]: +0.0

[4]: degC(-40) = degF(-40)
[4]: 1
```

Immediate Use

Any UDF can be used immediately after being defined, even on the same line. Generically, for a two-argument function, this looks like:

```
func(newUDF,{..[x]..[y]..})(x,y)
```

For a trivial example, take a function that we use immediately for multiplying two numbers:

```
[1]: func(mult,{[x]*[y]})(4,5)
[1]: +20.0
```

Recursion

User-defined functions that reference themselves are allowed. While there are numerous ways to implement recursion, we can make clever use of mixed-type operations to have a UDF call itself if a certain condition is met. In the example that follows, the `fac` function calculates the factorial of an integer greater than one.

```
func(fac,{
  [x]*(unquote(
    greater([x],2)*`fac'
  ))([x]-1)
})
```

A quick test of the above might look like:

```
[1]: fac(6)
[1]: +720.0

[2]: fac(fac(3))
[2]: +720.0
```

10.2 Multi-Line Functions

Technically, any given UDF stores just one line of code. However, we can pack more than one line into a `block`, or even better, for privately-scoped multi-line functions, a `sub`.

Privacy and Scope

Because `sub` carries occulted code, it follows that the parameters `[x]`, `[y]`, etc. are buried in at least two layers of curled braces ({}) within the body of a function.

There is an eventual limit to how 'deep' the paremter references work, thus it may be prudent to store `[x]`, `[y]`, etc. as local variables in the `sub`. To reinforce this, consider the following session in which a variable `a` is defined and then referenced in a variety of ways:

```
[1]: let(a,1)
[1]: 1

[2]: {a}
[2]: 1

[3]: {{a}}
[3]: {1}

[4]: {{{a}}}
[4]: {{1}}
```

```
[5]: {{{{a}}}}
[5]: {{{1}}}

[6]: {{{{{a}}}}}
[6]: {{{{{a}}}}}

[7]: {[a]}
[7]: {[a]}
```

We see structures such as {`a`} being replaced as if equivalent to `[a]`, and this pattern penetrates through three additional layers of curled braces ({}). Any more than four total layers and the variable isn't seen by the evaluator.

The rule is more strict for function parameters `[x]`, `[y]`, etc. In the following example, we witnes the technique of storing the incoming parameter `[x]` in a new variable `a`. Next we open an embedded `sub` which makes reference to `a` and `x`.

```
func(test,{
  sub({
    let(a,[x]):
    print_sub({
      print_<[a],[x]>
    })
  })
})(1)
```

Running the above example produces a vector `<1,[x]>`. That is, we see `[a]` properly restored, whereas the inner `sub` doesn't know what to do with `[x]`.

Application: Polish Notation Calculator

Jumping into an almost-practical example of a multi-line function, you may know about *Polish* notation, also known as *prefix* notation, which is a way of writing mathematical expressions by placing the operator before any numbers. For instance, the infix expression $3+4$ is written $+\,3\;4$ in Polish notation. The advantage to prefix notation is the list can grow on the right. If we write $\times\,3\;4\;9$, the infix translation is $3\times4\times9$.

The plan is to create a function that can take a Polish arithmetic expression and return the simplified result, which requires converting to infix notation familiar to `Sxript`. The most suitable format for Polish notation is a vector, with the operator as the first element and each number separated by a comma, for instance `<*,3,4,9>`. This is the kind of structure we expect to send to the function.

Converting to infix notation is done by plucking off the operator from the front of the vector, and then inserting the operator between each number. As a final step we use `reduce` and `join` to distill a vector to a number. The function `Polish` does exactly this as shown:

```
func(Polish,{
  sub({
    let(op,unvector(left([x],1))):
    let(dat,right([x],len([x])-2)):
    let(nfx,
      stack(mid([x],2,1),
        apply([op],[dat]))
    ):
    print_reduce(join,[nfx])
  })
})
```

The use case would be, for instance:

```
[2]: Polish(<*,3,4,9>)
[2]: +108.0
```

To help trace what happens to the input, we can sketch (in fake notation) what happens internally:

```
op  =  *
dat =  <4,9>
nfx =  <3,*4,*9>
print_ 3*4*9
```

10.3 Anonymous Functions

There is another species of function, called an *anonymous* function, also known as a *lambda*, which has instructions but no name. To motivate this, recall the example used to highlight the immediate use of a UDF:

```
[1]: func(mult,{[x]*[y]})(4,5)
[1]: +20.0
```

What we have is the name, specification, and use of a function all in one line. Supposing `mult` is used nowhere else in the program, one coud argue that the name `mult` didn't play a material role in multiplying two numbers. That is, only the body of the function and the arguments sent to it are significant to the program and the programmer.

Lambda Primitive

Anonymous functions are initiated with the `lambda` primitive, which takes one argument as occulted code. This is very much like `func` without the leading name argument.

Recasting the previous example as an anonymous function, one would write:

```
[2]: lambda({[x]*[y]})(4,5)
[2]: +20.0
```

Note that any `lambda` must be accompanied on the right by the list of arguments enlosed in parentheses. If we exclude this, the return from `lambda` is the *internal* name of the anonymous function. If you see this, you're probably making a stylistic error.

```
[3]: lambda({[x]*[y]})
[3]: lambda4115
```

The $ Shortcut

The word `lambda` can be replaced by a dollar sign (`$`). This is merely for saving keystrokes and reducing clutter.

```
[4]: $({[x]*[y]})(4,5)
[4]: +20.0
```

10.4 Report Primitive

A peculiar primitive called `report` is designed to take the name of a UDF as input and return the body of the function as occulted code.

For a trivial example of this, we could set up the followin session:

```
[1]: func(mult,{[x]*[y]})
[1]: mult

[2]: report(mult)
[2]: {[x]*[y]}
```

Since the output of `report` is occulted code, we should be able to wrap the whole expression with `lambda` to use the UDF in a roundabout way.

```
[3]: $(report(mult))(3,4)
[3]: +12.0
```

Note, however, that the *visible* return from the `report` primitive is formatted for the terminal. If the function contains special characters, particularly pertaining to quotes, these are 'formatted away' before reaching the screen. Internally, of course, the `report` primitive does not go through the formatting step.

10.5 Special Operators

There are two 'operators', namely the question mark (`?`) and the tilde (`~`), that may ease the burden of using many parentheses when calling functions.

Question Operator

Whenever the pattern `x?y` is encountered, where `x` and `y` stand for anything, the interpretation `x(y)` is returned. A typical use case could go as follows:

```
[5]: cos ? 3
[5]: -.9899924966004454

[6]: cos ? cos ? 3
[6]: .5486961336030971

[7]: cos(cos(3))
[7]: .5486961336030971
```

Tilde Operator

Consider any two-argument function `f(x,y)`. When the structure `a ~ b` is encountered, where `a` and `b` stand for anything, the interpretation `f(a,b)` is returned. A typical use case could go as follows:

```
[1]: func(tmult,{cos([x])*sin([y])})
[1]: tmult

[2]: 3 ~tmult~ 4
[2]: +0.7492287917633427

[3]: 3 ~tmult~ 4 ~tmult~ 5
[3]: -0.5806818746830253

[4]: cos(3)*sin(cos(4)*sin(5))
[4]: -0.5806818746830253
```

11 Flow and Procedure

Procedural control pertains to the flow of a program taking one channel or another based on a condition. `Sxript` offers a standard implementation of *procedural* statements, namely the 'if' structure present in most languages, along with 'for' loops and 'do' loops, and even the notorious 'goto'. Several of these are quite unlike the standard language primitives.

11.1 Iff Primitive

The 'if' primitive is written `iff`, which is a special function that takes three arguments. The first argument is always a binary condition, which must numerically reduce to either one or zero. If the condition resolves to one, control is passed to the second argument sent to `iff`. If the condition resolves to zero, control is passed to the third argument sent to `iff`. The second and third arguments must occur as a single line of occulted code.

```
[1]: iff(1=1,{`True'},{`False'})
[1]: True

[2]: iff(0=1,{`True'},{`False'})
[2]: False
```

For a less trivial example, a typical use case of `iff` would involve multiple lines per eventuality, achieved with code blocks or subprograms. Needles to mention, the `iff` primitive can be embedded within an enclosing code block or subprogram.

```
iff([condition],{
  block({
    a:b:print_c
  })
},{
  block({
    d:e:print_f
  })
})
```

The `iff` primitive makes quick work of a recursive factorial function:

```
func(fac,{
  iff([x]=1,{+1.0},{[x]*fac([x]-1)})
})
```

11.2 For Primitive

There is a primitive designated to the classic 'for' loop, denoted `for`. The `for` primitive takes two bulky arguments: (i) a vector specifying the looping variable, the lower bound, the upper bound, and increment, and (ii) occulted code to be looped over in which the looping variable is referenced. For a simple example, a for loop could look like:

```
[1]: for(<k,-3,5,2>,{[k],})
[1]: -3.0,-1.0,+1.0,+3.0,+5.0,
```

Had it not been for the comma after `[k]`, the above would evaluate to `+5.0`. Also, it's prudent to make sure a proper data structure emerges from the calculation, so we can enclose the whole thing in angle brackets (`<>`), and then use `smooth` to get rid of the extra comma.

```
[2]: for(<k,-3,5,2>,{[k]})
[2]: +5.0

[3]: smooth(<for(<k,-3,5,2>,{[k],})>)
[3]: <-3.0,-1.0,+1.0,+3.0,+5.0>
```

Fill UDF

The above is useful enough to adapt into a UDF for creating vectors with evenly-spaced numeric elements:

```
func(fill,{
  smooth(<for(
    <k,[x],[y],[z]>,{[k],}
  )>)
})
```

```
[5]: fill(-1,2,.5)
[5]: <-1.0,-0.5,+0.0,
      +0.5,+1.0,+1.5,+2.0>
```

Yet another factorial function can be made using `fill`. This one is non-recursive:

```
func(fac,{sub({print_
  reduce($({[x]*[y]}),fill(1,[x],1))
})})
```

Note that the `sub` apparatus would not be needed had it not been for the anonymous function containing the symbol `[x]`. Instead we could have first defined a UDF for multiplication so `fact` becomes simpler:

```
func(mult,{[x]*[y]}):

func(fact,{
  reduce(mult,fill(1,[x],1))
}):
```

For a variation on the filled vector, we can also create an unfilled vector using a one-argument UDF:

```
func(emptyvec,{
  <for(<i,1,[x]-1,1>,{,})>
})
```

Application: FizzBuzz

A classic problem in computer science and job interviews is the *FizzBuzz* problem, which goes like:

Write a program that scans the integers from 1 to 100. Print Fizz for multiples of three, and print Buzz for multiples of five. For multiples of both, print FizzBuzz.

As a `Sxrit` program, we can solve the FizzBuzz problem using a `for` loop:

```
print_
for(<i,1,100,1>,{sub({
  let(a,`'):
  iff([i]%3=0,{let(a,[a]+`Fizz')}):
  iff([i]%5=0,{let(a,[a]+`Buzz')}):
  print_unf([i]) ` : ' iff(([a]=`')!,{
    [a]`\n'},{[i]`\n'}):
})})
```

For a sanity check, the output `FizzBuzz` emerges for the numbers 15, 30, 45, 60, 75, and 90.

Application: ASCII Table

Using a `for` loop and the `iff` primitive together, we can make a miniature ASCII table provided the special characters are either ignored or otherwise stepped around.

```
for(<i,32,126,1>,{block({let(out,`'):
  iff([i]=39 ,{let(out,  `\''   )}):
  iff([i]=40 ,{let(out,  `('    )}):
  iff([i]=41 ,{let(out,  `)'    )}):
  iff([i]=44 ,{let(out,  `,'    )}):
  iff([i]=60 ,{let(out,  `<'    )}):
  iff([i]=62 ,{let(out,  `>'    )}):
  iff([i]=91 ,{let(out,  `['    )}):
  iff([i]=92 ,{let(out, `\\{\}' )}):
  iff([i]=93 ,{let(out,  `]'    )}):
  iff([i]=96 ,{let(out,  `\`'   )}):
  iff([i]=123,{let(out,  `{'    )}):
  iff([i]=125,{let(out,  `}'    )}):
  print_quote(unf([i])) + ` ':
  print_iff([out]=`',{
    quote(chr([i]))},{[out]})\t:
  print_iff(([i]+1-32)%5=0,
          {\n},{`'})})})
```

Columnar output is made possible by the `tab` character (9).

```
32      33 !    34 "    35 #    36 $
37 %    38 &    39 '    40 (    41 )
42 *    43 +    44 ,    45 -    46 .
47 /    48 0    49 1    50 2    51 3
52 4    53 5    54 6    55 7    56 8
57 9    58 :    59 ;    60 <    61 =
62 >    63 ?    64 @    65 A    66 B
67 C    68 D    69 E    70 F    71 G
72 H    73 I    74 J    75 K    76 L
77 M    78 N    79 O    80 P    81 Q
82 R    83 S    84 T    85 U    86 V
87 W    88 X    89 Y    90 Z    91 [
92 \    93 ]    94 ^    95 _    96 `
97 a    98 b    99 c    100 d   101 e
102 f   103 g   104 h   105 i   106 j
107 k   108 l   109 m   110 n   111 o
112 p   113 q   114 r   115 s   116 t
117 u   118 v   119 w   120 x   121 y
122 z   123 {   124 |   125 }   126 ~
```

11.3 Do... Loop

The `do...loop` is a *count-controlled* loop, which means the contens of the loop is executed a fixed number of times. To set the number, simply let a number hang before the leading `do` as shown:

```
block({
  print_`Hello w' + :
  5:
  do:
    print_`o' + :
  loop:
  print_`rld!'
})
```

The loop executes exactly five times, thus there are five `o`'s repeated in the output:

```
Hello wooooorld!
```

The `do...loop` structure can be self-nested, and the total number of iterations need not be a hard-coded number. Puzzle out why the following code block returns `+24.0`:

```
block({
  3:do:
    1+1:do:
      2+2:do:
        print_1+0:
loop:loop:loop})
```

Application: Quote Permutation

The `do...loop` structure also plays well with recursive functions. In the following we create a UDF called `perm`, which is designed to generate all permutations of an input quote. Initially, the argument `[x]` is empty, and the argument `[y]` takes the input. As evaluation goes on, the input is chopped up and sent back to `perm` in various chunks until all permutations are generated.

```
func(perm,{sub({
  let(a,[x]):let(b,[y]):
  print_iff(len([b])=0,{[a],},{
    sub({let(i,1):
    len([b]):
    do:let(newpre,
        [a] + mid([b],[i],1)):
      let(newsuf,
        left([b],[i]-1)
        + right([b],len([b])-[i])):
      print_perm([newpre],[newsuf]):
      let(i,[i]+1):
    loop })})})}):
```

A support function `permquote` is defined to serve as a front-end to `perm`.

```
func(permquote,{
  smooth(<perm(`',[x])>)
}):
```

```
[3]: permquote(`abcd')
[3]: <abcd,abdc,acbd,acdb,adbc,adcb,
      bacd,badc,bcad,bcda,bdac,bdca,
      cabd,cadb,cbad,cbda,cdab,cdba,
      dabc,dacb,dbac,dbca,dcab,dcba>
```

Application: Riemann Sums

For approximating the area under a curve, we employ the notion of Riemann sums, particularly the trapezoid rule, the midpoint rule, and Simpson's rule. Any one of these suffices for an approximation, however it's instructive to show several running simultaneously. Moreover, one can show that the result for Simpson's rule is a weighted average of the others mentioned, helping to verify overall accuracy against a direct implementation.

To demonstrate, consider the function $f(x) = 4x - x^2$ in the interval $0 \leq x \leq 4$. Let us slice the interval into $n = 15$ discrete bins. In `Sxript` notation, one writes:

```
func(f,{(
  4*[x] - [x]*[x]
)}):
let(x0,0):let(xn,4):let(n,15):
```

It is necessary to specify the spatial interval $\Delta x = (x_n - x_0)/n$, and also initialize the accumulator variables for left sum, right sum, midpoint sum, trapezoid sum, and the first of two Simpson's rule sums.

```
let(dx,([xn]-[x0])/[n]):
let(ls,0):let(rs,0):let(ms,0):
let(ts,0):let(si1,0):
```

Next we set up a loop that makes `n` iterations to approximate the left sum, right sum, and midpoint sum. The trapezoid sum and the first Simpson's sum are calculated after the loop.

```
let(j,0):
[n]:
do:
  let(xj,[x0] + [j]*[dx]):
  let(x1,[xj] + [dx]):
  let(xm,([xj] + [x1])/2):
  let(ls,[ls] + [dx]*(f([xj]))):
  let(rs,[rs] + [dx]*(f([x1]))):
  let(ms,[ms] + [dx]*(f([xm]))):
  let(j,[j]+1):
loop:
let(ts,(1/2)*([ls]+[rs])):
let(si1,(1/3)*([ts]+2*[ms])):
```

In order to verify the contents of `[si1]`, we construct a second loop that recalculates the Simpson's rule result.

```
let(h,([xn]-[x0])/(2*[n])):
let(si2,0):
let(j,0):
[n]:
do:
  let(xj, [x0] +
    [h] + ([j]/[n])*([xn]-[x0])):
  let(f1,f([xj]-[h])):
  let(f2,f([xj])):
  let(f3,f([xj]+[h])):
  let(si2, [si2] +
  [h]*(1/3)*([f1] + 4*[f2] + [f3])):
  let(j,[j]+1):
loop:
```

Check all outputs simultaneously using:

```
print_<[ls]\n,[rs]\n,[ms]\n,
        [ts]\n,[si1]\n,[si2]>
```

```
<+10.619259259259259
,+10.619259259259259
,+10.690370370370369
,+10.619259259259259
,+10.666666666666664
,+10.666666666666666>
```

We see various approximations either under- or overshooting the average result, with the last two being spot on. Ignoring the last floating-point digit in each result, the two approximations via Simpson's rule are in agreement. For completeness, the exact solution to the area under the curve in this example is:

$$\int_0^4 \left(4x - x^2\right) dx = \frac{32}{3} = 10.666\overline{6}$$

11.4 Anchor... Goto

Not needing line numbers, `Sxript` instead has 'anchors', which are a way to bookmark a location in code to which control may be passed using `goto_`.

An anchor is initiated by the `anchor_` keyword. The characters following the underscore constitute the name of the anchor, and these must be hard-coded. There cannot be a variable, function, or other structure used to 'calculate' an anchor name.

Control is passed to an anchor using the keyword `goto_`. Note that `anchor_` and `goto_` only apply locally to the code block or subprogram in which they're defined. Jumping to other code blocks isn't possible.

The information following `goto_` must match an anchor name, but does not have to be hard-coded. The target of `goto_` may be calculated from a variable or otherwise. In the following example we store in a variable `a` some word representing a randomly-selected anchor elsewhere in the code.

```
let(a,unquote(
  `cat'+unf(int(rand(3))))):

print_`A cat says ': goto_[a]:
anchor_cat0:print_`meow.':goto_cat9:
anchor_cat1:print_`purr.':goto_cat9:
anchor_cat2:print_`hiss.':goto_cat9:
anchor_cat9
```

Across multiple trials, the output is any one of three possibilities:

```
[1]: A cat says purr.
[2]: A cat says meow.
[3]: A cat says meow.
[4]: A cat says purr.
[5]: A cat says hiss.
```

11.5 If... Goto

The `if_` keyword initiates a conditional jump. Following the underscore is a condition that evaluates to one or zero. If the condition is zero, control is passed to the next line as if nothing happened. Otherwise, after the condition is the 'at' character (`@`), followed by an anchor name to which program flow is passed. Like `goto_`, the anchor reference need not be hardcoded.

Application: Fractal Fern

Going for a slightly nontrivial example, we demonstrate the whole procedural subsytem to make the classic 'fractal fern'. To initialize the program, we first need a helper function `inrange`, along with two variables `xx`, `yy` set to zero:

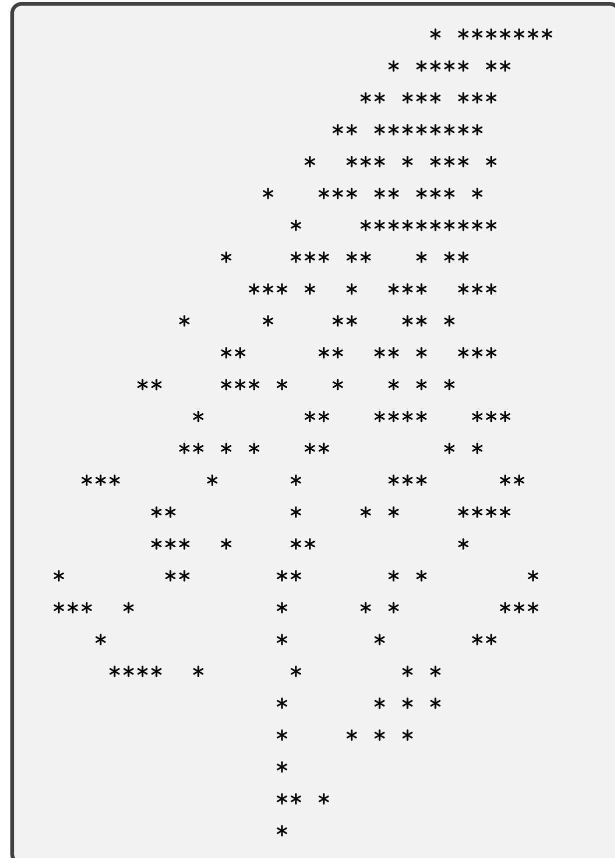
```
func(inrange,{
  iff(greater([y],[x]),{
    iff(greater([z],[y]),{1},{0})
  },{0})}):

let(xx,0):
let(yy,0):
```

The main loop makes 350 total iterations to play the so-called 'chaos game'. With each iteration, the vector `<[xx],[yy]>` is evaluated and sent to output.

```
print_scatter(smooth(<sub({
350:do:
  let(ii,1+int(rand(99))) ,:
  if_greater(1,[ii])    @ a:
  if_inrange(1,[ii],8)  @ b:
  if_inrange(8,[ii],15) @ c:
  if_greater([ii],15)   @ d:
  anchor_a:
  let(xx,  0                       ):
  let(yy,  .16*[yy]                ):
  goto_doneloop:
  anchor_b:
  let(xx,  .20*[xx]-.26*[yy]       ):
  let(yy,  .23*[xx]+.22*[yy]+1.6   ):
  goto_doneloop:
  anchor_c:
  let(xx, -.15*[xx]+.28*[yy]       ):
  let(yy,  .26*[xx]+.24*[yy]+.44   ):
  goto_doneloop:
  anchor_d:
  let(xx,  .85*[xx]+.04*[yy]       ):
  let(yy, -.04*[xx]+.85*[yy]+1.6   ):
  goto_doneloop:
  anchor_doneloop:
print_<[xx],[yy]>,:loop})>),36,27,0)
```

Positions in the Cartesian plane are marked with an asterisk (`*`) in accordance with the `scatter` primitive (not yet formally introduced). Regardless, you should be able to make out an attempted fern.



12 Large Numbers

By default, all numerical operations in `Sxript` are passed to the floating-type system of the host language, i.e. `QB64`, `JavaScript`, `C++`. This usually means the programmer is allowed roughly sixteen digits of precision, which is more than enough for most programing tasks.

Certain situations, however, call for more than sixteen digits of numerical precision, sometimes *many* more. For this, `Sxript` embeds a large floating-point number subsystem with its own language primitives, namely `largeadd`, `largesub`, `largemul`, `largediv`, `largeabs`, and `largesci`.

Large Number Format

For our purposes, the 'precision' or 'largeness' of a number refers to the amount of information required to represent the number. When large numbers are combined via numerical operations, the precision of result is determined by the precision of the inputs in accordance with ordinary arithmetic.

Large numbers are adorned with all of the same attributes as ordinary numbers in terms of signs, decimals, the way leading zeros are handled, etc. How-

ever, the ordinary language primitives such as `cos`, `greater`, `ln`, etc., are not compatible with large number formats, perhaps for obvious reasons.

As scientific notation varies per host language, an in-house solution for scientific notation of large numbers is used. The `largesci` primitive takes a single large number as an input, and returns a vector with the decimal information in the first element and the power of ten in the second element.

```
[1]: largesci(123456789)
[1]: <+1.234567890,+8.0>
```

Note that `123456789` isn't particularly large, however the mechanism works on numbers of any size.

```
[2]: largesci(-0.0000000012345)
[2]: <-1.2345,-9.0>
```

Numerical Operations

The `largeadd`, `largesub`, `largemul`, `largediv` primitives constitute the basis of large-number numeric operations. Things like exponents, roots, trigonometry functions, etc., have no definition in terms of primitives. These must be purpose-built from the basis primitives.

Operations like addition, subtraction, and multiplication churn out straightforwardly. The more precise the input, the more precise the output:

```
[1]: largeadd(-7,15.123456789)
[1]: +8.123456789

[2]: largesub(7,15.123456789)
[2]: -8.123456789

[3]: largemul(2.000001, 3.000001)
[3]: +6.000005000001
```

Large-number division behaves more subtly than the other three operations. For instance, if asked to compute the fraction 1/3, a standard floating-point implementation would churn out roughly sixteen digits. This is also the case when feeding the same problem to `largediv`.

```
[4]: 1/3
[4]: +0.3333333333333333

[5]: largediv(1,3)
[5]: +0.3333333333333333
```

In order to get more `3`'s in the output, one of the inputs needs to carry more precision. By the same token, hard-coding the precision into the input is the only clue `largediv` knows to stop:

```
[6]: largediv(1.000000000000000000,3)
[6]: +0.333333333333333333333

[7]: largediv(1,3.000000000000000000)
[7]: +0.333333333333333333333
```

Application: Fibonacci by Division

An interesting application of large-number division involves calculating (note the `8` buried in the center of the string of `9`'s):

```
1 / 999999999999999999999998999999999999999999999999
```

We need to evaluate the above to a few thousand digits, which can be done by replacing the numerator with `1.000...` with many trailing zeros. Putting the zeros into the program is best done by `for` loop:

```
1.(for(<i,1,2495,1>,{0}))
```

With this, the problem can be properly fed to `largediv`:

```
largediv(
  1.(for(<i,1,2495,1>,{0})),
  999999999999999999999998999999999999999999999999
)
```

Playing with the output, one notices that only the information after the decimal is relevant, thus we replace it with a newline character:

```
replace(
  quote(
    largediv(
      1.(for(<i,1,2495,1>,{0})),
      999999999999999999999998999999999999999999999999
    )
  ),`.',`\n'
)
```

Adjusting the width of the output window, one notices that the first few dozen Fibonacci numbers are spelled out in the result. (Formatted to multiple boxes and truncated for brevity.)

```
000000000000000000000000000000000000000000000001000000000000000000000001
000000000000000000000002000000000000000000000003000000000000000000000005
000000000000000000000008000000000000000000000013000000000000000000000021
000000000000000000000034000000000000000000000055000000000000000000000089
000000000000000000000144000000000000000000000233000000000000000000000377
000000000000000000000610000000000000000000000987000000000000000000001597
```

```
000000000000000000002584000000000000000000004181000000000000000000006765
000000000000000000010946000000000000000000017711000000000000000000028657
000000000000000000046368000000000000000000075025000000000000000000121393
000000000000000000196418000000000000000000317811000000000000000000514229
000000000000000000832040000000000000000001346269000000000000000002178309
000000000000000003524578000000000000000005702887000000000000000009227465
```

```
000000000000000014930352000000000000000024157817000000000000000039088169
000000000000000063245986000000000000000102334155000000000000000165580141
000000000000000267914296000000000000000433494437000000000000000701408733
000000000000001134903170000000000000001836311903000000000000002971215073
000000000000004807526976000000000000007778742049000000000000012586269025
000000000000020365011074000000000000032951280099000000000000053316291173
```

```
000000000000086267571272000000000000139583862445000000000000225851433717
000000000000365435296162000000000000591286729879000000000000956722026041
000000000001548008755920000000000002504730781961000000000004052739537881
000000000006557470319842000000000010610209857723000000000017167680177565
000000000027777890035288000000000044945570212853000000000072723460248141
000000000117669030460994000000000190392490709135000000000308061521170129
```

Appliction: Fibonacci Sequence

For a more direct attack on the problem of generating large Fibonacci sequence tables, one could set up a `for` loop to get the job done:

```
print_ 1`: ' (largesci ? let(a,0)) \n:
print_ 2`: ' (largesci ? let(b,1)) \n:
let(c,largeadd([a],[b])):
print_for(<i,3,10,1>,{block({
  let(a,[b]):
  let(b,[c]):
  print_unf([i]) `: '
    largesci(
      let(c,largeadd([a],[b])))\n:
    })})
```

```
1: <+0.0,+0.0>
2: <+1.0,+0.0>
3: <+2.0,+0.0>
4: <+3.0,+0.0>
5: <+5.0,+0.0>
6: <+8.0,+0.0>
7: <+1.3,+1.0>
8: <+2.1,+1.0>
9: <+3.4,+1.0>
10: <+5.5,+1.0>
```

Application: Large Factorials

No large-number system demonstration is complete without several takes on the factorial, thus we oblige the expectation here. In the folowing we establish a `for` loop that reports the first fifteen integer factorials. Results are reported in scientific notation.

```
sub({let(n,1):
  print_for(<k,1,15,1>,{
    (quote(unf([k])) + `! = ')
    (largesci(let(n,
     largemul([n],[k]))))\n})})
```

```
1! = <+1.0,+0.0>
2! = <+2.0,+0.0>
3! = <+6.0,+0.0>
4! = <+2.4,+1.0>
5! = <+1.2,+2.0>
6! = <+7.2,+2.0>
7! = <+5.04,+3.0>
8! = <+4.032,+4.0>
9! = <+3.6288,+5.0>
10! = <+3.6288,+6.0>
11! = <+3.99168,+7.0>
12! = <+4.790016,+8.0>
13! = <+6.2270208,+9.0>
14! = <+8.71782912,+10.0>
15! = <+1.307674368,+12.0>
```

For something more 'portable' it's handy to have a factorial function contained in a UDF. To thie end, we construct a recursive function `largefac` to get the job done.

```
func(largefac,{
  iff(([x]=0)|([x]=1),{1},{
    largemul([x],
      largefac(largesub([x],1)))})})
```

For a quick show of force, the above is readily applied to integers in the hundreds or thousands. For instance, one thousand factorial, i.e.

$$1000! = 1000 \cdot 999 \cdot 998 \cdots 2 \cdot 1$$

is given by:

```
print_largesci(largefac(1000))
```

The result is has more than 2500 decimals of precision:

```
<+4.0238726007709377354370243392300398571937486421071463254379991042993851239862
902059204420848696940480047998861019719605863166687299480855890132382966994459099
974245040870737599188236277271887325197795059509952761208749754624970436014182780
946464962910563938874378864873371191810458257836478499770124766328898359557354325
131853239584630755574091142624174743493475534286465766116677973966688202912073791
4385371958824980812686783837455973174613608537953452422158659320192809087829
```

```
3084313928444032812315586110369768013573042161687476096758713483120254785893207б
```

Application: Binomial Coefficients

- Toggle axis numerals, `1` or `-1`.

For a demonstration, let us plot `cos` in a window appropriae to show its character:

```
plot(cos,-5, 5, 0.01, 40, 20, 1)
```

```
{cos, -5, 5, 0.01, 40, 20}
________________*^^^**__________________1
_______________**____**_________________0.8947369088564453
_______________*______*_________________0.7894738177128906
______________**______**________________0.6842107265693359
______________*________*________________0.5789476354257812
_____________**________**_______________0.4736845442822264
<____________*__________*_____________**0.3684214531386718
<____________*__________*_____________*_0.2631583619951171
*___________**__________**___________**_0.1578952708515623
*___________*____________*___________*__0.0526321797080076
*__________**____________**__________*__-0.05263091143554721
**_________*______________*_________**__-0.15789400257910202
_*_________*______________*_________*___-0.2631570937226564
_**_______**______________**_______**___-0.3684201848662112
__*_______*________________*_______*____-0.4736832760097658
__**_____**________________**_____**____-0.5789463671533206
___*_____*__________________*_____*_____-0.6842094582968754
___**___*___________________**___**_____-0.78947254944043
____*****____________________*****______-0.8947356405839848
______*_________________________________-0.9999987317275394

--------------------++++++++++++++++++++
5444333322221111000000001111222233334445
........................................
0742974296419641863113681469146924792470
48371604937261594822849516273940617384
37047158259360371488417306395285174073
51739517384062840622604826048371593715
87654321098765432100123456789012345678
99988887777666655555555666677778888999
74297429641964186311368146914692479247
```

The plot is headed by the function name, along with relevant information sent to the `plot` primitive. Points in the plot are represented by an asterisk. Points slightly outside the boundary of the window are indicated with an arrow-like character.

Note that the function being plotted need not be a language primitive. Any UDF can go as the first argument in `plot`, including an anonymous function.

13.2 Scatter Plot

A second two-dimensional graph, namely the scatter plot, is invoked by the `scatter` primitive, which takes strictly four arguments:

- List of ordered pairs `<x,y>` contained in a `vector`.
- Horizontal window size as integer.
- Vertical window size as integer.
- Toggle axis numerals, `1` or `-1`.

For a trivial example, suppose we want to plot the four points `<1,4>`, `<2,7>`, `<5,3>`, `<4,5>` in a window fifteen units across and then units tall. In a single line, we submit:

```
scatter(
  <<1,4>,<2,7>,<5,3>,<4,5>>,
15,10,1)
```

Somewhat resembling a standard function plot, the scatter plot lives in the Cartesian plane in an automatically-fitted window:

```
  *            7
               6.555555555555555
               6.111111111111111
               5.666666666666667
          *    5.222222222222222
               4.777777777777778
*              4.333333333333334
               3.888888888888889
               3.444444444444444
              *3

+++++++++++++++
111122233334445
...............
025814702581470
087542108754210
051728405172840
074185207418520
012457801245780
048271504827150
025814702581470
```

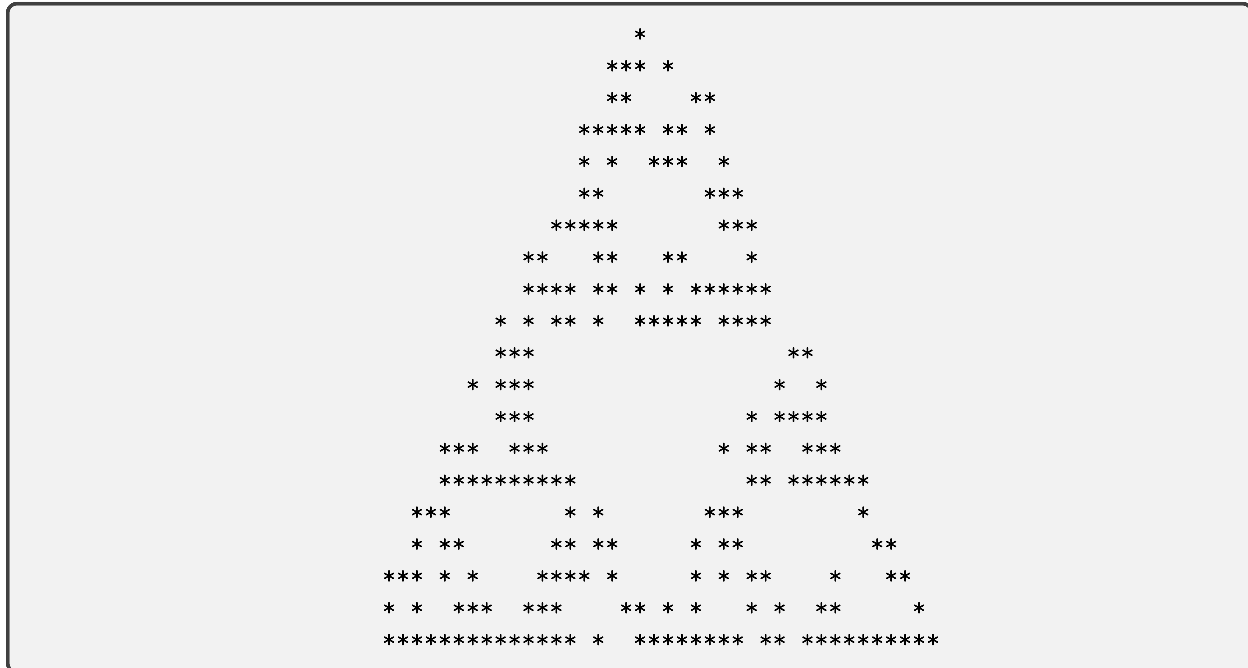
Application: Sierpinski Triangle

A scatter plot can be used for crudely visualizing a Sierpinski triangle, which is an infinite nesting of equilateral triangles. Setting up a well-known algorithm for plotting points within the Sierpinski triangle, we produce the following program:

```
let(xx1,0):let(yy1,0):
let(xx2,2.4/(3^.5)):let(yy2,0):
let(xx3,1.2/(3^.5)):let(yy3,1.2):
let(xx,0):let(yy,0):

print_scatter(smooth(<sub({
350:do:
  let(ii,int(rand(3))+1):
  iff([ii]=1,{block({
    let(xx,.5*([xx]+[xx1])):
    let(yy,.5*([yy]+[yy1])):
    print_()})},{}):
  iff([ii]=2,{block({
    let(xx,.5*([xx]+[xx2])):
    let(yy,.5*([yy]+[yy2])):
    print_()})},{}):
  iff([ii]=3,{block({
    let(xx,.5*([xx]+[xx3])):
    let(yy,.5*([yy]+[yy3])):
    print_()})},{}):
  print_<[xx],[yy]>,:loop})>)
  ,40,20,-1)
```

Note that all code is sent to the `scatter` primitive. Despite receiving decimal-value inputs, each point is placed in an appropriate bin determined by the size of the window. The axis numerals are disabled via the `-1` sent as the last argument.



---

14 Functional Programming

Despite being limited to the string space of the given host language, `Sxript` encourages a coding discipline classfied as *functional programming*. Right off the bat, functional programming generally means two things: (i) programming in a 'functional' language such as `Lisp` or `Haskell`, and (ii) programming in an ordinay language, such as `C++` or `Python` while under a self-imposed discipline that functional languages would enforce.

Functional programming looks and smells differently than the *imperative* paradigm, which is the usual step-by-step approach to coding. Rather than setting up the state of the computer to get ready for a calculation, as one might have in a `for` loop, one instead performs iterations directly using functions. One reason for this, apart from having to define extra variables, is to not trick the computer into a state that interferes unexpectedly with the program or its neighbors.

Some of the key tenets of functional programming are reviewed in no particular order:

14.1 Purity

A function is 'pure' if the function always returns the same output for the same input. In addition, a pure function causes no impact on any part of the program outside the function, which is to say a pure function has no *side effects*.

Nearly all of the `Sxript` primitives, and thereby most possible user-defined functions, are compatible with the rule of purity. We say 'nearly' because things like `date` and `time` are not pure by definition.

Sloppy use of the `block` primitive leads to side effects if variables and functions in the parent scope are overwritten. To erase this concern, the `sub`/`return_` combination is the safest way to perform calculations that can't escape their container.

Immutability

Following the rule of purity is the rule of immutability, which refers to the stability of data. Once created, immutable data is never deleted or edited. This includes variables and functions.

Of course, it's difficult to fathom programming without being able to assign a new value to an existing variable, at least certainly for complex tasks. Nonetheless, immutability is something to shoot for when writing clean code.

14.2 High-Order Functions

A functional programming language allows functions to be passed around as if they're variables or data, and functions are thus known as *first class citizens*. This is to say functions can be stored, sent to functions, returned from other functions, etc.

A function *high order* if it takes a function as input and/or returns a function as output. Sometimes these are called 'generators' or 'factories', depending on the resource.

It was mentioned previously that `Sxript` primitives and user-defined functions are straightforwardly stored as variables, for instance:

```
[1]: let(f,cos)
[1]: cos

[2]: let(g,sin)
[2]: sin

[3]: [f]([g](3))
[3]: .9900590857598653
```

One may also note that the primitives `apply`, `map`, `reduce` are each high-order functions by virtue of taking a function as an argument.

Callbacks

When a function is passed to another function as an argument, the function being passed is called a `callback`. The so-called callback function can be used within the function to which it is sent.

For an example, let us define two arithmetic user-defined functions `mult`, `add`. The `mult` UDF multiplies its arguments `x`, `y` and returns the product. For simplicity assume `x`, `y` are positive integers. Similar applies to `add`.

For a 'workhorse', define a UDF called `calculate`, taking three arguments, namely two numbers, and a function name, i.e. a callback.

```
func(mult,{[x]*[y]}):
func(add,{[x]+[y]}):

func(calculate,{[z]([x],[y])}):
```

Demonstrating on a trivial example, one applies the above via:

```
[4]: calculate(10,7,mult)
[4]: +70.0

[5]: calculate(10,7,add)
[5]: +17.0
```

Currying

As a way to manage complexity in a program, *currying* refers to condensing a multi-argument function to one with fewer arguments. Along the way we demonstrate the use of high-order functions.

To proceed, create a function to wrap around `mult` called `currymult`, whose job is to take a single integer argument `a` and then lay out a new function that multiplies its single argument by `a`. The return from `currymult` is a chunk of occulted code.

To help with calls to `currymult`, another UDF called `newmult` is defined to 'unlock' the new function from the occult brackets. Any call to `newmult` creates a new curried function named after the hard-coded number send to `newmult`. When executed, the following returns `+18.0, +30.0`.

```
func(mult,{[x]*[y]}):

func(currymult,{sub({
  let(a,unf([x])):
  print_{func(mult{a},{mult({a},[x])})}
})}):

func(newmult,{
  unoccult(currymult([x]))}):

newmult(3):
newmult(5):

print_mult3(6) ,:
print_mult5(6)  :
```

For an example that is more useful in the field, we can transform a natural logaritm function `ln()` of base e into a more general function `logn()` of base n.

```
func(logn,{ln([x])/ln([y])}):

func(currylog,{sub({
  let(n,unf([x])):
  print_{func(log{n},{logn([x],{n})})}
})}):

func(newlog,{
  unoccult(currylog([x]))}):

newlog(10):
newlog(5):

print_log10(1000) ,:
print_log5(1000)   :
```

14.3 Composition

Another way to build and manage complexity in a program is using function *composition*. This could mean something as simple as rolling two functions into one using an imperative practice, however the use of high-order functions makes composition a more powerful asset.

To illustrate, consider the `+1` operation that simply adds one to any number, and let's always take `0` as the starting point. Naturally, one could express the integer `4` using $0+1+1+1+1$, which requires the programmer to invoke the `+1` operation four times. Similar would apply for any positive integer.

To make something better, start with a function `f` that applies the `+1` operator to the argument sent to `f`. Also, define a two-argument high-order function `p` such that

$$p(x, y) = x(x(y)) \ ,$$

where x, y could stand for anything so far. As user-defined functions, these are:

```
func(f,{[x]+1}):

func(p,{
  [x]([x]([y]))
}):
```

For a sanity check, one could test `f(0)` to get `+1.0`, and `p(f,0)` to get `+2.0`. Note that the `p`-function takes the function `f` as its first argument.

The construction $p(f, x)$ at $x = 0$ attains the integer `+2.0`, thus it's worth abstracting this into a function `g` as shown. While we're at it, exploit the same pattern to define a new function `h` that takes `g` as the first argument, and so on for a few layers:

```
func(g,{p(f,[x])}):
func(h,{p(g,[x])}):
func(i,{p(h,[x])}):
func(j,{p(i,[x])}):
```

Choosing any for illustration, we have that the function `h` is defined such that

$$h(x) = p(g, x) \ ,$$

which simplifyies via:

$$\begin{aligned} h(x) &= g(g(x)) \\ h(x) &= g(p(f, x)) \\ h(x) &= g(f(f(x))) \\ h(x) &= p(f, f(f(x))) \\ h(x) &= f(f(f(f(x)))) \end{aligned}$$

Clearly then, we see $h(0)$ is equivalent to four applications of the `f`-function, therefore

$$h(0) = 4 \, .$$

As a matter of notational convenience, a shorthand way to write $h(0)$ takes advantage of the question operator (`?`). The following are equivalent representations of the integer four:

```
f(f(f(f(0)))) = f? f? f? f? 0 = h?0
```

In terms of the functions `f`, `g`, `h`, `i`, and remembering that `Sxript` ignores whitespace, we prepare a vector expressing the first sixteen positive integers:

```
print_apply(unf,<
                f? 0  ,
             g?    0  ,
             g? f? 0  ,
          h?       0  ,
          h?    f? 0  ,
          h? g?    0  ,
          h? g? f? 0  ,
       i?          0  ,
       i?       f? 0  ,
       i?    g?    0  ,
       i?    g? f? 0  ,
       i? h?       0  ,
       i? h?    f? 0  ,
       i? h? g?    0  ,
       i? h? g? f? 0  ,
    j?             0  >)
```

If you haven't spotted it yet, the 'digits' `f`, `g`, `h`, `i` are like binary switches counting upward in the binary number system.

```
<1,2,3,4,5,6,7,8,
  9,10,11,12,13,14,15,16>
```

Note that the above can be used for a few more purposes. For instance, we can replace the function `p` with a version that more-deeply embeds the `y`-argument:

$$p(x, y) = x(x(x(y)))$$

With this, a call to `j(0)` returns the number `+81.0`. The reason why is left as an exercise.

Or, leaving $p(x, y)$ in its original form, one may replace `f` with $f(x) = \cos(x)$. Then, the expression `j ?  0` is equivalent to sixteen layers of cosine applied to zero:

```
[8]: j ? 0
[8]: 0.7383692041223232

[9]: cos(cos(cos(cos(cos(cos(cos(
     cos(cos(cos(cos(cos(cos(cos(
     cos(cos(0))))))))))))))))
[9]: 0.7383692041223232
```

Application: Digits of Pi

If you can't remember the digits of π, start with $x = 3.14$ and calculate $x - \tan(x)$ to find

$$3.14 - \tan(3.14) = 3.14159265\ldots .$$

Interestingly, the result gives eight correct digits after the decimal point with floating-point junk afterward. Feed this result back into the recursive formula $x = \tan(x)$ to get more correct digits.

To set up this calculation quickly, redefine the function `f` via:

```
func(f,{[x] - tan([x])}):
```

Then, using the composite functions defined above, it only requires `g(3.14)` to get a result that exceeds the floating-point precision of the host language.

```
[1]: g(3.14)
[1]: +3.141592653589793
```

14.4 Recursion

In functional programming, performing repetitive actions using recursion is preferred (or mandadted) over other loops that introduce extra state to the program. We've seen this several times while defining various versions of the factorial UDF, with a tight functional-friendly expression being:

```
func(fac,{
  iff([x]=1,{+1.0},{[x]*fac([x]-1)})
})
```

Of course, sometimes it's a better to use a traditional flow control, even a `do...loop`, as long as other safe practices are also in play.

Application: Ackermann Function

The *Ackermann* function is an early example of a recursive function that is easy to write but takes a very nontrivial path to arrive at a final answer. For non-negative integers, the Ackermann funtion is specified by:

$$A(x,y) = \begin{cases} x = 0 : & y + 1 \\ x > 0, y = 0 : & A(x-1,1) \\ x > 0, y > 0 : & A(x-1, A(x, y-1)) \end{cases}$$

```
func(Ack,{sub({
  let(a,[x]):let(b,[y]):
  print_iff([a]=0, {
    [b]+1 },{
    iff(greater([a],0) & ([b]=0),
      {Ack([a]-1,1)},
      {Ack([a]-1,Ack([a],[b]-1))
    })})})})
```

The Ackermann function has an enormous convergence time but except when the most meager of arguments are sent.

```
[2]: Ack(0,0)
[2]: +1.0

[3]: Ack(3,1)
[3]: +13.0

[4]: Ack(3,2)
[4]: +29.0

[5]: Ack(3,3)
[5]: +61.0
```

Fixed-Point Combinator

Now we get into a somewhat high-minded idea. Consider first the standard factorial UDF, but switch the variable from `x` to `y`:

```
func(fac,{
  iff([y]=1,{+1.0},
    {[y]*fac([y]-1)})})
```

The next move is to turn the recursive call to `fac` into a variable, particularly `x`. This makes the factorial UDF into a two-argument function:

```
func(fac,{
  iff([y]=1,{+1.0},
    {([y])*[x]([x],[y]-1)})})
```

Note that [x] occurs twice in the function becasuse the embedded function call must also have two arguments. As a sanity check, one may use such a function via:

```
[2]: fac(fac,6)
[2]: +720.0
```

To proceed, it would be nice to abstract away the multiple calls to fac, which means replacing the arguments fac and 6 with variables. For this we define a fucntion for *self-application* as follows:

```
func(self,{
  [x]([x],[y])
})
```

```
[4]: self(fact,6)
[4]: +720.0
```

To make the above even more abstract, one notices that we didn't really need the self UDF in a language that has anonymous functions. This means we can write:

```
[5]: $({[x]([x],[y])})(fact,6)
[5]: +720.0
```

Similarly, we can replace the two-argument fact UDF with an anonymous function to write the final answer.

```
$({[x]([x],[y])})
  ($({
    iff([y]=1,{+1.0},
      {([y])*[x]([x],[y]-1)})
  }),6)
```

The above is called a *fixed-point combinator*, a class of computational objects that come in many flavors and have many purposes. In our case, we have used self-application to achieve recursion using the construction $x(x, y)$.

If the embedded argument 6 seems irritatingly imprisoned, we can cheat by setting the argument equal to a temporary constant, i.e., a variable that doesn't change in accordance with the rule of immutability. By doing so, one can fully separate the function body from its input.

```
$({sub({let(a,[x]):
  print_
  $({[x]([x],[y])})
  ($({
    iff([y]=1,{+1.0},
      {([y])*[x]([x],[y]-1)})
  }),[a])
})})(6)
```

Application: Newton's Method

Students of calculus learn of Newton's method for iteratively solving one-dimensional equations. Fundamental to this is the formula for the derivative of a continuous function, which as Sxript code is given by:

```
func(deriv,{
  (([x]([y]+[z]))
  -([x]([y]-[z]))) / (2*[z])
}):
```

In the above, x is the equation to be solved, y is the test point, and z is a small deviation from the test point. In terms of the same parameters, Newton's method applies a recursive formula captured by:

```
func(linapx,{
  [y] - ([x]([y]))/deriv([x],[y],[z])
}):
```

In a traditional scheme, one might implement Newton's method by defining a function to capture the problem, and then write a for loop to churn a fixed number of iterations. For instance, to solve the problem $x = \cos(x)$, one could write:

```
func(p,{[x] - cos([x])}):
```

Using an inital guess y=1 and going for ten iterations, one gets a satisfactory result using:

```
func(newtonloop,{
  sub({
  let(fxn,[x]):
  let(ans,[y]):
  for(<k,1,[z],1>, {
    let(ans,linapx([fxn],[ans],0.001))
  }):
  print_[ans]
  })}):

print_newtonloop(p,1,10)
```

Of course, we'd rather do this with a pure approach that avoids the `for` loop. Using a recursive function, the same result is achieved with:

```
func(newton,{
  iff(
  (greater(0.000001,abs(p([x]))))
  |(greater(abs(p([x])),1000000)),
  {[x]},
  {newton(linapx(p,[x],0.001))})
}):

print_newton(1)
```

Next, turn the recursive call to `newton` into a variable, which will take slot `x`, bumping the numeric argument to `y`. This transforms the function's appearance and behavior via:

```
func(newton,{
  iff(
  (greater(0.000001,abs(p([y]))))
  |(greater(abs(p([y])),1000000)),
  {[y]},
  {[x]([x],linapx(p,[y],0.001))})
}):

print_newton(newton,1)
```

Furthermore, let us recast the hard-coded calls to function `p` into a variable sent to the function, taking slot `y`. This bumps the numeric arument to `z`:

```
func(newton,{
  iff(
  (greater(0.000001,abs([y]([z]))))
  |(greater(abs([y]([z])),1000000)),
  {[z]},
  {[x]([x],[y],linapx([y],[z],0.001))})
}):

print_newton(newton,p,1)
```

This example is looking much like the factorial example above. At this stage, we need a self-application function of the form:

```
func(self,{
  [x]([x],[y],[z])
})
```

Implementing this as an anonymous function, the calculation now looks like:

```
print_
$({[x]([x],[y],[z])})
  ($({
  iff(
  (greater(0.000001,abs([y]([z]))))
  |(greater(abs([y]([z])),1000000)),
  {[z]},
  {[x]([x],[y],linapx
    ([y],[z],0.001))})}),
p,1)
```

Finally, we can abstract `linapx`, `deriv`, and `p` into anonymous functions to write the whole enchilada as a single line:

```
$({[x]([x],[y],[z])})
  ($({
    iff(
    (greater(0.000001,abs([y]([z]))))
    |(greater(abs([y]([z])),1000000)),
    {[z]},
    {[x]([x],[y],
      $({[y] - ([x]([y]))/
        $({(([x]([y]+[z]))
           -([x]([y]-[z])))/(2*[z])
        })([x],[y],[z])
      })([y],[z],0.0001))})}),
  $({[x] - cos([x])}),
  1)
```

Omega Combinator

Prevoiously, we encountered the two-argumemt self-application function

$$T = x(x, y)$$

in the form of:

```
func(self,{[x]([x],[y])})
```

In the mathematical notation, we choose the letter T after Alan Turing, where $x(x, y)$ is known as the *Turing* combinator, also known as the *Mockingbird* combinator.

Now comes a simpler creature called the *Omega* combinator, defined as

$$\Omega = x(x) \ ,$$

which is a one-argument self-application function. Denoting the omega combinator with the symbol `m`, the equivalent as a `Sxript` function reads:

```
func(m,{[x]([x])}):
```

In the lambda calculus notation, $x(x)$ is equivalent to xx, which is also true for this discussion. The body of function `m` could be changed to `[x][x]` (no parentheses) without consequence.

The Omega combinator is as simple as it looks. For instance `m(5)` returns `55`. Of course, `m(+5)` returns `+10.0`. Interestingly though, `m(5+)` returns `5+5+`. This essentially tricks the language into lazy-evaluating the incomplete math expression, which is to say, skips the evaluating.

Putting Ω to greater use, also define two functions r, s such that

$$r = \Omega(\Omega(x))$$
$$s = r(r(x)) \ ,$$

or equivalently:

```
func(r,{m(m([x]))}):
func(s,{r(r([x]))}):
```

For lack of proper names, let's call `r` the *second-order* self-applicator, and correspondingly `s` the *fourth-order* self-applicator. While this toolset may seem meager at face value, maybe a term like 'fourth-order' may foreshadow what's to come. For a quick test of the above, one finds:

```
[4]: r(0)
[4]: 0000

[5]: s(0)
[5]: 0000000000000000
```

That is, `r(0)` returns `0000`, a total of four digits. However, `s(0)` returns sixteen digits. Naturally, the functions `m`, `r`, `s` can be compounded by standard means, i.e. `m(r(0))` returns `00000000`, whereas `m(s(0))` returns 32 digits.

Next, recall an obscure shortcut of syntax reserved by the question mark operator (`?`). The question operator allows a single-argument function to be called as if in-line with arithmetic. For instance, `cos ? 0` is equivalent to `cos(0)`, or, `r?0` resolves to `0000`.

Trying the combinations `m(r?)(0)`, `r(m?)(0)`, we receive sixteen zeros in each case.

```
[6]: m(r?)(0)
[6]: 0000000000000000

[7]: r(m?)(0)
[7]: 0000000000000000
```

To understand these, it helps to take away the argument on the right to attain the lazy evaluation of the left.

```
[8]: m(r?)
[8]: r?r?

[9]: r(m?)
[9]: m?m?m?m?
```

We can insert any of `m`, `r`, `s` in place of the function calls above. For instance, each of `r(r?)(0)`, `m(s?)(0)`, return $4^4 = 256$ digits. Things get bigger from here, as `s(m?)(0)`, `r(s?)(0)` each return $4^8 = 65536$ digits.

```
[10]: len(r(r?)(0))
[10]: 256

[11]: len(m(s?)(0))
[11]: 256

[12]: len(s(m?)(0))
[12]: 65536

[13]: len(r(s?)(0))
[13]: 65536
```

Although no `Sxript` implementation has finished the next calculation to date, the expression `s(s?)(0)` is equivalent to 4^{16} zeros, where

$$4^{16} = 4294967296 \, .$$

At one byte per character, this is 4 GB of data!

More fun can be had with the Omega combinator. For this, let us define a few functions that are less abstract than the previous three:

```
func(e,{[x]+1}):
func(f,{[x]*2}):
```

To see the program and the output without having to execute anything, the following expressions have the anticipated result subtracted such that the return is all zeros.

```
(m(m(e?))(0)) - (2*2)
(m(r(e?))(0)) - (2*4)
(r(m(e?))(0)) - (2*4)
(r(r(e?))(0)) - (2*8)
(m(s(e?))(0)) - (2*16)
(s(m(e?))(0)) - (2*16)
(r(s(e?))(0)) - (2*32)
(s(s(e?))(0)) - (2*128)
```

```
(m(m(f?))(1)) - (4^2)
(m(r(f?))(1)) - (4^4)
(r(m(f?))(1)) - (4^4)
(r(r(f?))(1)) - (4^8)
(m(s(f?))(1)) - (4^16)
(s(m(f?))(1)) - (4^16)
(r(s(f?))(1)) - (4^32)
(s(s(f?))(1)) - (4^128)
```

We may also consider the rather ugly function `manycos`, which takes an initial number `x` along with an iteration number `y` and calculates

$$\text{manycos}\,(x) = \cos\left(\cos\left(\cos\left(\ldots x \ldots\right)\right)\right) \, ,$$

with `y` total cosines.

```
func(manycos,{sub({
  let(a,cos([x])):
  [y]-1:do:
    let(a,cos([a])):
  loop:print_[a]
})})
```

Interestingly, even for relatively small `y`, the return from `manycos` tends to converge to a single answer, regardless of the initial value `x`. The infinite-`y` return from `manycos` happens to be the solution to the transcendental equation $x = \cos(x)$, solved by

$$x = 0.7390851332151607\ldots \, .$$

To evaluate the near-equivalent of `manycos` using recursion, we recycle the `m`, `r`, `s` functions defined above.

```
(m(m(cos?))(0)) - manycos(0,4)
(m(r(cos?))(0)) - manycos(0,8)
(r(m(cos?))(0)) - manycos(0,8)
(r(r(cos?))(0)) - manycos(0,16)
(m(s(cos?))(0)) - manycos(0,32)
(s(m(cos?))(0)) - manycos(0,32)
(r(s(cos?))(0)) - manycos(0,64)
(s(s(cos?))(0)) - manycos(0,256)
```

We can get more abstract be defining a function

$$w = x\left(x\left(y\right)\right) \, ,$$

which generalize the functions `r` and `s` above. Then, using the tilde (`~`) operator, we build expressions such as `r ~ w ~ 0`, which is equivalent to `r(r(0))`, returning sixteen zeros.

14.5 Impure Functions

Closure

In functional programming, a *closure* is a function that captures its surrounding environment, allowing it to access and, in some cases, modify variables and functions from its defining scope, even when called in a different context. For an example, consider the following 'counter' program implemented in `JavaScript`:

```
function makecounter() {
  let num = 0;
  return function() {
    num++;
    console.log(num);
  };
}

let counterA = makecounter();
counterA();  // returns 1
counterA();  // returns 2

let counterB = makecounter();
counterB();  // returns 1
counterB();  // returns 2
counterB();  // returns 3
```

The high-order function `makecounter` initializes a variable `num` to one and then returns a function whose job is to increment `num` by one.

Using `makecounter` to create a new function, namely `counterA`, it turns out that a copy of `num`, initialized to one, is 'saved' invisibly and is associated with `counterA` in memory. Then, whenever `counterA` is invoked, the variable `num` is treated as mutable.

Using `makecounter` again to create another function `counterB`, this function gets its own fresh copy of the `num` variable, regardless of how `counterA` has treated its own copy.

Seemingly, closure threatens to breed confusion about hidden mutable state, breaking of scope, or other subtle dependencies. At face value, the behavior of the above counter functions appear to behave *impurely*, namely because the output always changes while the call to the function does not.

Mutable Functions

`Sxript` does not (need to) exhibit closure in the pure sense. To achieve a closure-like effect though, we should be able to define a function that embeds the data intended to be enclosed, whichs means building a function that is intentionally impure.

For an example comparable to the `JavaScript` case above, we write the following `Sxript` code:

```
func(makecounter,{sub({
  let(name,[x]):
  print_{
    func({name},{
    0:
    func({name},
      replace(report({name}),1,
      unoccult(elem(report({name}),1))
      + 1 ))})}})}):

unoccult(makecounter(counterA)):
unoccult(makecounter(counterB)):

func(tick,{block({
  unoccult(elem(report([x]),2)):
  print_unf(unoccult(
    elem(report([x]),1)))})}):

print_tick(counterA),:
print_tick(counterA),:
print_tick(counterB),:
print_tick(counterB),:
print_tick(counterB)
```

The `makecounter` UDF is a high-order function that takes the name of a new counter as the argument.

For a return, `makecounter` emits a piece of occulted code containing two items. First is the 'enclosed' data referenced by the function, a hard-coded zero in this case. Second is a `let` primitive containing an instruction to increment the enclosed data and overwrite the definition of the counter. That is, we have a high-order function giving rise to another high-order function that rewrites itself.

Initializing a new counter requires removing the oocculting structure returned by `makecounter`. In order to use a counter, we define another function `tick` that executes the instruction in a given counter and then reports the respective enclosed data. The output of the above example is the same as the `JavaScript` case:

```
1,2,1,2,3
```

Note that the same program can be written, behaving identically, if we replace all calls to `func` with `let`, and all calls to `report` with `symbol`.

```
func(makecounter,{sub({
  let(name,[x]):
  print_{
    let({name},{
    0:
    let({name},
      replace(symbol({name}),1,
      unoccult(elem(symbol({name}),1))
      + 1
    ))
})}})}):

func(tick,{block({
  unoccult(elem(symbol([x]),2)):
  print_unf(unoccult(
    elem(symbol([x]),1)))})}):
```

15 Structured Programs

Strictly speaking, there is no such thing as a multi-line `Sxript` program. As we know, multi-line programs are allowed within the `block` primitive (public scope) or `sub` primitive (private scope).

As it turns out, multi-line programs are also allowed to be stored via text file and read during execution. The contents of the file does not need to be wrapped in a `block`-like primitive. For example, consider the file `vector.txt`, which contains:

```
func(last,{elem([x],len([x]))}):

func(fill,{
  smooth(<for(
  <k,[x],[y],[z]>,{[k],})>)}):

func(dotprod,{reduce(join,[x]*[y])}):

func(crossprod,{
  sub({
    let(r1,[x]):
    let(r2,[y]):
    print_<
    (elem([r1],2)*elem([r2],3)-
     elem([r1],3)*elem([r2],2)),
 -1*(elem([r1],1)*elem([r2],3)-
     elem([r1],3)*elem([r2],1)),
    (elem([r1],1)*elem([r2],2)-
     elem([r1],2)*elem([r2],1))
    >})}):

print_`vector.txt':
```

The file `vector.txt` is stored in the `prg` directory (short for 'programs'), which is neighbor to `bin`, in which the `Sxript` executable is launched. (These details will vary per implementation but the in-program use will be the same throughout.)

Include Primitive

Clearly, the intent of `vector.txt` is to impart a handful of user-defined functions in a subsequent program, thus we want a `block`-like primitive having public scope to take on the file contents. For this, we use `include`, which takes the file location and name as a quote.

```
[1]: include(`../prg/vector.txt')
[1]: vector.txt
```

With the above, we can test the new functions a number of ways, for instance:

```
[2]: report(last)
[2]: {elem([x],len([x]))}

[3]: dotprod(<1,2,3>,<4,5,6>)
[3]: +32.0

[4]: crossprod(<1,2,3>,<4,5,6>)
[4]: <-3.0,+6.0,-3.0>
```

Run Primitive

A second primitive designated to running external programs, called `run` does everything `include` does with the exception of having private scope. That is, variables and functions within `run()` do not touch the environment unless carried away by `print_`.

For example, the examples used to discuss `apply`, `map`, `reduce` are contained in the file `applymapreduce.txt`.

```
print_apply(sqrt,<9,16,25,121>),:
print_apply(len,
       <`dog',`cat',`mouse'>),:
print_apply(type,<1+1,`Hello',type>),:
print_map(<10,5,3>,greater,<6,5,4>),:
print_map(<`dog',`cat',`mouse'>,
          instr,<`o',`z',`u'>),:
print_map(<1,2,3>,join,<4,5,6>),:
print_map(<1,2,3>,join,<4,5,6>+0),:
print_reduce(join,<a,b,c,d,e>),:
print_reduce(join,<1,2,3+0,4,5>),:
print_reduce(
       join,<5,*,4,*,3,*,2,*,1>),:
print_reduce(greater,<4,3,0,-2>)
```

Running this program gives all familiar outputs. For a sanity check, we also verifiy that the present scope is empty.

```
[1]: run(`../prg/applymapreduce.txt')
[1]: <3,4,5,11>,<3,3,5>,
     <number,quote,word>,
     <1,0,0>,<2,0,3>,<14,25,36>,
     <+5.0,+7.0,+9.0>,abcde,
     +15.045,+120.0,1

[2]: report()
[2]: {print_()}
```

Dependency

Using the `include` primitive, it's straightforward for external programs to depend on external programs, allowing the programmer to use the sum of all involved variables and functions.

For example, present in the `prg` directory is the file `cpmplexnum.txt`, which contains all of the user-defined functions for complex arithmetic. In addition, there exists a file `complexfunc.txt`, which contains:

```
include(`../prg/complexnum.txt'):

func(cderiv,{
  (([x]([y][c+][z]))[c-]
   ([x]([y][c-][z])))[c/]
     (<2,0>[c*][z])}):

func(clinapx,{
  [y] [c-] (([x]([y])) [c/]
    cderiv([x],[y],[z]))}):

func(cnewton,{
  iff(
  (greater(0.000001,
    abs(cmag([x]([y])))))
  |(greater(abs(cmag([x]([y]))),
    1000000)),
  {[y]},
  {cnewton([x],
    clinapx([x],[y],<0.001,0>))})}):

print_`complexfunc.txt':
```

Executing the above, we load the whole complex number toolkit.

```
[1]: include(`../prg/complexfunc.txt')
[1]: complexfunc.txt
```

For a test, consider the order-two equation

$$z + z^2 = 7 + 12i\,,$$

which has two solutions

$$z \approx \begin{cases} 2.7611 + 1.8398i \\ -3.7611 - 1.8398i \end{cases} .$$

The next job is to turn this into a `Sxript` function.

```
[2]: func(p,{
       ([x][c+]([x][c^]2)[c-]<7,12>)
     })
[2]: p
```

Supplying the initial guesses $(2, 2)$, $(-2, -2)$, repsectively, we use the loaded `cnewton` UDF to find two solutions.

```
[3]: cnewton(p,<2,2>,5)
[3]: <+2.7611405131341713,
      +1.8398471258245837>

[4]: cnewton(p,<-2,-2>,5)
[4]: <-3.7611405259465363,
      -1.839847138247968>
```

16 Applications

16.1 Calculus 101

Differentiation

Some of our calculus toolkit was developed either before or during the study on high-order functions, thus some streamlining is due for the sake of programmer-friendliness. To begin we restate the functions for the derivative and for the linear approximation at the heart of Newton's method in one dimension:

```
func(deriv,{
  (([x]([y]+[z]))
  -([x]([y]-[z]))) / (2*[z])
}):

func(linapx,{
  [y] - ([x]([y]))/deriv([x],[y],[z])
}):
```

For completeness, the 'baseline' UDF for Newton's method is stated as follows - note the hardcoded function `p`, which represents the problem to be solved.

```
func(newton,{
  iff(
  (greater(0.000001,abs(p([x]))))
  |(greater(abs(p([x])),1000000)),
  {[x]},
  {newton(linapx(p,[x],0.001))})
}):
```

Of course, it would be cleaner to abstract the call to `p` into a variable, thus bumping the numerical argument `x` over to slot `y`.

```
func(newton,{
  iff(
    (greater(0.000001,abs([x]([y]))))
    |(greater(abs([x]([y])),1000000)),
    {[y]},
    {newton([x],
      linapx([x],[y],0.001))})}):
```

Then, we can send the problem and the inital guess directly to the `newton` UDF to receive an answer. For instance, we can set $p(x) = x - \cos(x)$ and supply an initial guess of $x_0 = 1$ to find:

```
[4]: newton($({[x]-cos([x])}),1)
[4]: +0.7390851333834115
```

Integration

Recall that one-dimensional integrals are numerically approximated using Riemann sums. The program is required to specify (i) the function to be integrated, (ii) the lower- and (iii) upper-bounds on the dependent variable, always `x`, and (iv) the number of integration bins.

Building a four-argument function to evaluate the Riemann sum, one can write:

```
func(rsum,{sub({
  let(dx,([z]-[y])/[t]):
  let(ls,0):let(rs,0):let(ms,0):
  let(ts,0):let(si,0):let(j,0):
  [t]:do:
    let(xj,[y] + [j]*[dx]):
    let(x1,[xj] + [dx]):
    let(xm,([xj] + [x1])/2):
    let(ls,[ls] + [dx]*([x]([xj]))):
    let(rs,[rs] + [dx]*([x]([x1]))):
    let(ms,[ms] + [dx]*([x]([xm]))):
    let(j,[j]+1):loop:
  let(ts,(1/2)*([ls]+[rs])):
  let(si,(1/3)*([ts]+2*[ms])):
  print_[si]})}):
```

The output of the function is contained in `[ls]` for the left-hand sum, `[rs]` for the right-hand sum, `[ms]` for the midpoint sum, `[ts]` for the trapezoid sum, and `[si]` for Simpson's rule. By default, the function returns `si`. For a test, let us calculate the area under the curve $y = 4x - x^2$ in the interval $0 \leq x \leq 4$ using $t = 15$ bins to reproduce the example used previously:

```
[7]: rsum($({4*[x]-[x]*[x]}),0,4,15)
[7]: +10.666666666666664
```

16.2 Complex Numbers

Any complex number z contains a real part a and an imaginary part b and is written

$$z = a + ib\,,$$

where i is the imaginary unit $\sqrt{-1}$.

For representing such a complex number in `Sxript`, we use a vector `<a,b>`. Due to this, functions for addition and subtraction of complex numbers are straightforwardly written:

```
func(cadd,{[x]+[y]}):

func(csub,{[x]-[y]}):
```

For multiplication of two complex numbers z_1, z_2, the product is written

$$z_1 \cdot z_2 = (a_1 a_2 - b_1 b_2, a_1 b_2 + a_2 b_1)\ .$$

Churning this into a UDF, one writes:

```
func(cmul,{<
  elem([x],1)*elem([y],1)-
  elem([x],2)*elem([y],2),
  elem([x],1)*elem([y],2)+
  elem([x],2)*elem([y],1)>}):
```

For division of two complex numbers z_1, z_2, the quotient is written

$$\frac{z_1}{z_2} = \frac{z_1 \cdot \overline{z}_2}{|z_2|^2}\ .$$

For this we need functions for the complex conjugate and the magnitude of a complex number.

```
func(conj,{
  <elem([x],1),-1*elem([x],2)>}):

func(cmag2,{
  elem([x]*[x],1)+
  elem([x]*[x],2)}):

func(cmag,{sqrt(cmag2([x]))}):

func(cdiv,{
  (cmul([x],conj([y])))/
  (cmag2([y]))}):
```

For navigation in the complex plane, we must easily convert complex numbers from the Cartesian form $z = (a, b)$ to the polar form

$$z = (r\cos(\phi)\,, r\sin(\phi))\ ,$$

where r is the magnitude and ϕ is the phase. For this we define a function `cpol` that takes a complex number z and returns (r, ϕ). The `pcart` function implements the z-equation as written above.

```
func(cpol,{
  <cmag([x]),
  atan2(elem([x],2),elem([x],1))>}):

func(pcart,{
  elem([x],1)*
  <cos(elem([x],2)),sin(elem([x],2))>}):
```

Raising a complex number to a real-number (whole or decimal) power is achieved with the `cpow` UDF.

```
func(cpow,{
  sub({
    let(a,cpol([x])):
    let(rad,elem([a],1)):
    let(ang,elem([a],2)):
    let(pwr,[y]):
    let(rad,[rad]^[pwr]):
    let(ang,[ang]*[pwr]):
    print_pcart(<[rad],[ang]>)})}):
```

For complex exponents, we need something more intricate:

```
func(cpow2,{sub({
  let(ex,elem([x],1)):
  let(ey,elem([x],2)):
  let(ea,elem([y],1)):
  let(eb,elem([y],2)):
  let(lnz,[ex]*[ex] + [ey]*[ey]):
  print_iff([lnz]=0,{<0,0>},{sub({
    let(lnz,0.5*ln([lnz])):
    let(argz,atan2([ey],[ex])):
    let(mag,
      eul([ea]*[lnz]-[eb]*[argz])):
    let(ang,[ea]*[argz]+[eb]*[lnz]):
    print_pcart(<[mag],[ang]>)})})})}):
```

All of the work for a complex logarithm has already been done, thus we assemble the `clog` UDF with the following:

```
func(clog,{sub({
  let(ex,elem([x],1)):
  let(ey,elem([x],2)):
  let(lnz,[ex]*[ex] + [ey]*[ey]):
  print_iff([lnz]=0,{<0,0>},{sub({
    let(lnz,0.5*ln([lnz])):
    let(argz,atan2([ey],[ex])):
    print_<[lnz],[argz]>})})})}):
```

Finally, it's convenient to recast the functions for complex arithmetic to inline operators via the tilde (~) operator.

```
let(c+,~cadd~):
let(c-,~csub~):
let(c*,~cmul~):
let(c/,~cdiv~):
let(c^,~cpow~):
let(c2^,~cpow2~):
let(I,<0,1>):
```

For a quick test, suppose we put all of the functions for complex arithmetic into a single `block`. Then, a typical session in the terminal could look like:

```
[2]: <3,0> [c-] [I]
[2]: <+3.0,-1.0>

[3]: <1,2> [c+] <3,4>
[3]: <+4.0,+6.0>

[4]: <1,2> [c*] <3,4>
[4]: <-5.0,+10.0>

[5]: <3,0>[c+]<4,0>[c*][I]
[5]: <+3.0,+4.0>

[6]: conj(<4,-3>)
[6]: <4,+3.0>
```

```
[7]: cmag(<4,-3>)
[7]: 5

[8]: cdiv(<4,-3>,<1,2>)
[8]: <-0.4,-2.2>

[9]: <1,2> [c/] <4,-3>
[9]: <-0.08,+0.44>

[10]: cpow2(<3,4>,<1,2>)
[10]: <-0.4198131755619574,
       -0.6604516942073322>

[11]: clog(<3,4>)
[11]: <+1.6094379124341003,
        0.9272952180016122>
```

Complex Newton's Method

Here we combine the tools of calculus and complex numbers. Exhibit one is the derivative of a complex function, which takes the original `deriv` UDF and replaces any real numbers with complex numbers, along with any standard arithmetic operators with complex operators. Do the same with the `linapx` UDF.

```
func(cderiv,{
  (([x]([y][c+][z]))[c-]
   ([x]([y][c-][z])))[c/]
     (<2,0>[c*][z])}):

func(clinapx,{
  [y] [c-] (([x]([y])) [c/]
    cderiv([x],[y],[z]))}):
```

Next we retool the function for Newton's method to handle complex numbers.

```
func(cnewton,{
  iff(
  (greater(0.000001,
     abs(cmag([x]([y])))))
  |(greater(abs(cmag([x]([y]))),
    1000000)),
  {[y]},
  {cnewton([x],
    clinapx([x],[y],<0.001,0>))})}):
```

For a test, consider the order-three equation

$$z^3 + 2z^2 + 3z + 4 = 0\,,$$

which has one real solution and two complex solutions:

$$z \approx \begin{cases} -1.6506 \\ -0.1747 + 1.5468i \\ -0.1747 - 1.5469i \end{cases}$$

Encoding this as a `Sxript` function and testing, we write:

```
func(p,{(
  ([x][c^]3) [c+]
  (<2,0>[c*]([x][c^]2)) [c+]
  (<3,0>[c*][x]) [c+] <4,0>
)}):

print_cnewton(p,<-1,0>,5)  ,\n:
print_cnewton(p,<0,1>,5)   ,\n:
print_cnewton(p,<0,-1>,5)    :
```

The output of the above, as expected, consists of the three approximate solutions to the order-three equation.

```
<-1.6506291914510418,
-5.137166417968668e-17>,
<-0.17468540469990948,
 +1.5468688882006263>,
<-0.17468540469990948,
 -1.5468688882006263>
```

16.3 Linear Algebra

As briefly mentioned, a matrix is represented as a vector of vectors. The only primitive specialized for matrices is `column(x,n)`.

To develop a toolkit for matrices, it helps to have user-defined functions for manipulating individual elements, rows, and columns, along with things like transpose, etc. The contents of `/prg/matrix.txt` does just that:

```
include(`../prg/vector.txt'):

`Return element [y],[z]
 from matrix [x]':
func(melem,{
  elem(elem([x],[y]),[z])}):

`Transpose a matrix':
func(mtrans,{
  smooth(<sub({let(a,[x]):print_
    for(<i,1,len(elem([a],1)),1>,{
      column([a],[i]),})})>)}):

`Augment matrix [y]
 to the right of [x]':
func(maug,{smooth(<sub({
  let(k,1):
  len([x]):do:
    print_stack(elem([x],[k]),
                elem([y],[k])),:
    let(k,[k]+1):loop})>)}):
```

```
`Remove row [y] and column [z]
 from matrix [x]':
func(submatrix,{smooth(
  mtrans(smooth(
  replace(mtrans(smooth(
  replace([x],[y],))),[z],))))}):

`Replace element [y],[z]
 in matrix [x] with [t]':
func(mreplace,{
  replace([x],[y],replace(
    elem([x],[y]),[z],[t]))}):

`Replace column [y] of matrix [x]
 with vector [z]':
func(mrepcol,{
  mtrans(replace(
    mtrans([x]),[y],[z]))}):

print_`matrix.txt',:
```

Next, we have another file `/prg/linalg.txt` containing user-defined functions for matrix multiplication (`mmult`), matrix application to a vector (`axb`), determinant of a matrix (`determinant`), and inverse of a matrix (`minverse`). Following is a quick demonstration of some of this utility.

```
[1]: include(`../prg/linalg.txt')
[1]: linalg.txt,

[2]: let(a,<<1,2,-1>,<2,1,2>,<-1,2,1>>)
[2]: <<1,2,-1>,<2,1,2>,<-1,2,1>>

[3]: determinant([a])
[3]: -16.0

[4]: minverse([a])
[4]: <<+0.1875,+0.25,-0.3125>,
      <+0.25,+0.0,+0.25>,
      <-0.3125,+0.25,+0.1875>>

[5]: mmult([a],minverse([a]))
[5]: <<+1.0,+0.0,+0.0>,
      <+0.0,+1.0,+0.0>,
      <+0.0,+0.0,+1.0>>
```

17 Problems and Solutions

17.1 100 Doors

There are 100 doors in a row that are all initially closed. You make 100 passes by the doors.[1]

- The first time through, visit every door and toggle the door (if the door is closed, open it; if it is open, close it).
- The second time, only visit every 2nd door (door 2, 4, 6, ...), and toggle it.
- The third time, visit every 3rd door (door 3, 6, 9, ...), etc, until you only visit the 100th door.

After the last pass, which doors are open?

Solution

As subprogram:

```
func(mult,{[x]*[y]}):

func(fill,{
  smooth(<for(
    <k,[x],[y],[z]>,{[k],})>)}):

let(doors,apply($({-1.0}),fill(1,100,1))):

func(toggle,{
  smooth(<
    for(<i,1,100,1>,{
      iff(([i]%[x])=0,{-1,},{1,})})>)}):

for(<i,1,100,1>,{
  let(doors,map([doors],mult,toggle([i])))}):

print_smooth(<for(<i,1,100,1>,{
  iff(elem([doors],[i])=1,{[i],})})>)
```

Output

```
<+1.0,+4.0,+9.0,+16.0,+25.0,+36.0,+49.0,+64.0,+81.0,+100.0>
```

[1] Rosetta Code. 100 Doors. https://rosettacode.org/wiki/100_doors

17.2 Babbage Problem

What is the smallest positive integer whose square ends in the digits 269696?[2]

Solution

As single line:

```
func(babbage,{
  iff(([x]([y]))=int(([x]([y]))),{
    [x]([y])
  },{
    babbage([x],[y]+1)
  })
})($({(([x])*(10^6) + 269696)^.5}),1)
```

Output

```
+25264.0
```

[2] Rosetta Code. Babbage Problem. https://rosettacode.org/wiki/Babbage_problem

17.3 Coconut Pyramids

Build a three sided pyramid and a four sided pyramid by stacking coconuts. Keep building them higher and higher. At some point, you will find you have used nearly the same number of coconuts for each pyramid. When the difference in the number of coconuts is 1, how big are the pyramids, i.e., how many coconuts used in each?

Solution

As subprogram:

```
let(iSq,1): let(nSq,1): let(iTri,1): let(nTri,1):

func(copy,{for(<i,1,[y],1>,{[x]})}):
func(tabfill,{copy(\ ,9-len([x]))[x]}):

print_\n`Triangle pyramid           Square pyramid\n':
print_(apply(tabfill,<`i',`n',`i',`n'>)):

100:
do:
  iff(greater([nSq],[nTri]),{block({print_():
    let(iTri,[iTri] + 1):
    let(nTri,[nTri] + [iTri] * ([iTri] + 1) / 2)})
  },{block({print_():
    let(iSq,[iSq] + 1):
    let(nSq,[nSq] + [iSq] ^ 2)})
  }):
  print_
  iff(abs([nTri]-[nSq])=1,{
    \n (apply(tabfill,<[iTri],[nTri],[iSq],[nSq]>))
  }):
loop:

print_\n`------------- last checked -------------'\n:
print_(apply(tabfill,<[iTri],[nTri],[iSq],[nSq]>))
```

Output

```
Triangle pyramid           Square pyramid
<        i,        n,        i,        n>
<     +2.0,     +4.0,     +2.0,     +5.0>
<     +6.0,    +56.0,     +5.0,    +55.0>
<    +11.0,   +286.0,     +9.0,   +285.0>
--------------- last checked -------------
<    +57.0, +32509.0,    +45.0, +31395.0>
```

17.4 Golden Ratio

Wrate a program that computes the golden ratio.

Solution

As structured program:

```
include(`../prg/sqrt.txt'):

let(acc,250):

print_
largetrim(
  (1 [l+] lsqrt(5,2.22,[acc])) [l/] 2
,[acc])
```

Output

```
+1.6180339887498948482045868343656381177203091798057628621354486227052604628189
0244970720720418939113748475408807538689175212663386222353693179318006076672635443
3389086595939582905638322661319928290267880675208766892501711696207032221043216
2695486262963
```

17.5 Haskell Tutorial

One of the best tutorials on computer programming ever published was posted to the YouTube channel *FrungyKing* on August 18, 2013. The video is called *Programming - Why Haskell is Great - 10 minutes*. The full url to the video is: `https://www.youtube.com/watch?v=RqvCNb7fKsg`

The challenge is to reproduce the tutorial using `Sxript` in place of Haskell.

Solution

As subprogram:

```
print_  let(greeting,`Hello')                                           \n:
print_  func(swedish,{[x]*`f' - `f'})                                   \n:
print_  swedish ? [greeting]                                            \n:
print_  let(personname,`Satoshi')                                       \n:
print_  swedish ? [personname]                                          \n:
print_  let(sentence,[greeting] + ` ' + [personname])                  \n:
print_  swedish ? [sentence]                                            \n:
print_  func(yell,{[x]+`!'})                                            \n:
print_  func(enthusiastically,{[x]([x]([x]([y])))})                     \n:
print_  func(very,{enthusiastically([x],[y])})                          \n:
print_  yell ? [greeting]                                               \n:
print_  yell ~ enthusiastically ~ [greeting]                            \n:
print_  yell ? swedish ? [greeting]                                     \n:
print_  yell ~ enthusiastically ~ (swedish ~ very ~ [greeting])         \n:
print_  (swedish~very~[sentence])($({very([x],())})~enthusiastically~yell)
```

Output

```
Hello
swedish
Hfeflflfo
Satoshi
Sfaftfofsfhfi
Hello Satoshi
Hfeflflfof fSfaftfofsfhfi
yell
enthusiastically
very
Hello!
Hello!!!
Hfeflflfo!
Hfffffffefffffffflfffffffflfffffffo!!!
Hfffffffefffffffflfffffffflfffffffofffffff fffffffSfffffffafffffffftfffffffoffffff
fsfffffffhfffffffi!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

17.6 Koch Fractal

Plot the Koch fractal snowflake.

Solution

As subprogram:

```
`let(path,`F')':      `line':
let(path,`FRRFRRF'): `snowflake':
let(depth,3):

func(sweep,{sub({
  let(a,[x]):let(b,[y]):
  let(c,[z]):let(r,`'):
  print_for(<i,1,len([a]),1>,{block({
    let(d,mid([a],[i],1)):
    iff([d]=[b],{
      let(r,[r]+[c])},{
      let(r,[r]+[d])}):
    print_()})}):print_[r]})}):

[depth]:do:let(path,
  sweep([path],`F',`FLFRRFLF')):
loop:

let(turtle,<0,0,0>):
let(pi,4*atan(1)):
func(crawl,{
  <elem([x],1)+
    ([y])*cos(elem([x],3)*[pi]/180),
    elem([x],2)+
    ([y])*sin(elem([x],3)*[pi]/180),
    elem([x],3)>}):
func(swivel,{
  <elem([x],1),elem([x],2),
    elem([x],3) + [y]>}):
func(fwd,{let(turtle,
  crawl([turtle],[x]))}):
func(lft,{let(turtle,
  swivel([turtle],[x]))}):
func(rgt,{let(turtle,
  swivel([turtle],[x]))}):

print_scatter(smooth(<sub({
print_for(<i,1,len([path]),1>,{
block({
iff(mid([path],[i],1)=`F',{fwd(2)}):
iff(mid([path],[i],1)=`L',{lft(60)}):
iff(mid([path],[i],1)=`R',{rgt(-60)}):
print_<reduce($({[x],[y]}),
  left([turtle],2))>,})})})>),
36,36,-1)
```

Output

As scatter plot:

```
               *** *
             * *  *  *
           *** *   *** *
           *          *
           **        * *
     *       *       *       *
   *** *   **        * *   *** *
 * *  *  * *          *  * *  *  *
** *   *** *           *** *   *** *
*                                 *
*                                * *
 *                               *
*                                * *
*                                 *
***                            *** *
 * *                          *  *
  * *                        * *
     *                       *
   **                        * *
 * *                          *  *
** *                           *** *
*                                 *
*                                * *
 *                               *
*                                * *
*                                 *
** *   *** *           *** *   *** *
 * *  *  * *          *  * *  *  *
   *** *   **        * *   *** *
     *       *       *       *
           **        * *
           *          *
           *** *   *** *
             * *  *  *
               *** *
                 *
```

17.7 Nth

Write a function/method/subroutine/... that when given an integer greater than or equal to zero returns a string of the number followed by an apostrophe then the ordinal suffix. [3]

Use your routine to show here the output for at least the following (inclusive) ranges of integer inputs: 0-25, 250-265, 1000-1025. Note: apostrophes are now optional to allow correct apostrophe-less English.

Solution

As structured program:

```
func(nth,{
  sub({
    let(a,right([x],2)):
    let(result,`th'):
    iff(right([a],1)=1,{let(result,`st')}):
    iff(right([a],1)=2,{let(result,`nd')}):
    iff(right([a],1)=3,{let(result,`rd')}):
    iff([a]-11=0,{let(result,`th')}):
    iff([a]-12=0,{let(result,`th')}):
    iff([a]-13=0,{let(result,`th')}):
    `print_[x]\'[result]':
    print_[x][result]
  })
}):

include(`../prg/vector.txt'):

print_ apply(nth,apply(unf,fill(1,25,1)))       \n\n:
print_ apply(nth,apply(unf,fill(250,265,1)))    \n\n:
print_ apply(nth,apply(unf,fill(1000,1025,1)))      :
```

Output

```
<1st,2nd,3rd,4th,5th,6th,7th,8th,9th,10th,11th,12th,
13th,14th,15th,16th,17th,18th,19th,20th,21st,22nd,
23rd,24th,25th>

<250th,251st,252nd,253rd,254th,255th,256th,257th,
258th,259th,260th,261st,262nd,263rd,264th,265th>

<1000th,1001st,1002nd,1003rd,1004th,1005th,1006th,
1007th,1008th,1009th,1010th,1011th,1012th,1013th,
1014th,1015th,1016th,1017th,1018th,1019th,1020th,
1021st,1022nd,1023rd,1024th,1025th>
```

[3] Rosetta Code. Nth. https://rosettacode.org/wiki/N%27th

17.8 Pascal Triangle

Write program that calculates the `n`th row of the Pascal triangle. Display the first sixteen rows. Display the row that begins with 1, 100, ...

Solution

As structured program:

```
include(`../prg/factorial.txt'):

func(pascal,{
  iff(let(n,[x])=0,{<1>},{sub({
  iff([n]%2=0,{block({print_():
    let(p,smooth(<for(<i,0,[n]/2-1,1>,{binom([n],[i]),})>)):
    let(q,stack([p],binom([n],[n]/2)))})
  },{block({print_():
    let(p,smooth(<for(<i,0,[n]/2,1>,{binom([n],[i]),})>)):
    let(q,[p])
  })}):
  let(r,smooth(<for(<i,len([p]),0,-1>,{elem([p],[i]),})>)):
  print_stack([q],[r])
})})}):

include(`../prg/largenum.txt'):

print_
for(<i,0,15,1>,{
  apply(largen,pascal([i])) \n
}):

print_apply(largen,pascal(100))
```

Output

(1 of 2)

```
<1>
<1,1>
<1,2,1>
<1,3,3,1>
<1,4,6,4,1>
<1,5,10,10,5,1>
<1,6,15,20,15,6,1>
<1,7,21,35,35,21,7,1>
<1,8,28,56,70,56,28,8,1>
<1,9,36,84,126,126,84,36,9,1>
<1,10,45,120,210,252,210,120,45,10,1>
<1,11,55,165,330,462,462,330,165,55,11,1>
<1,12,66,220,495,792,924,792,495,220,66,12,1>
<1,13,78,286,715,1287,1716,1716,1287,715,286,78,13,1>
<1,14,91,364,1001,2002,3003,3432,3003,2002,1001,364,91,14,1>
<1,15,105,455,1365,3003,5005,6435,6435,5005,3003,1365,455,105,15,1>
```

Output

(2 of 2, square wrapping)

```
<1,100,4950,161700,3921225,75287520,1192052400,16007560800,186087894300,19022318
08400,17310309456440,141629804643600,1050421051106700,7110542499799200,441869426
77323600,253338471349988640,1345860629046814650,6650134872937201800,306645108029
88208300,132341572939212267400,535983370403809682970,2041841411062132125600,7332
066885177656269200,24865270306254660391200,79776075565900368755100,2425192697203
37121015504,699574816500972464467800,19173532007804430507636000,49988137020347265
25205100,124108478119482865453368000,2937233982161094482396376000,66324638306863423
796047200,143012501349174257560226775,294692427022540894366527900,58071742972088
9409486981450,1095067153187962886461165020,1977204582144932989443770175,34200295
47493938143902737600,5670048986634686922786117600,9013924030034630492634340800,1
3746234145802811501267369720,20116440213369968050635175200,282588088711625741663
68460400,381165328959867279453342024000,493782357970737157473647622000,61448471214
136179596720592960,73470998190814997343905056800,84413487283064039501507937600,9
3206558875049876949581681100,98913082887808032681188722800,100891344545564193334
812497256,98913082887808032681188722800,93206558875049876949581681100,8441348728
3064039501507937600,73470998190814997343905056800,61448471214136179596720592960,
49378235797073715747364762200,38116532895986727945334202400,282588088711625741663
68460400,20116440213369968050635175200,13746234145802811501267369720,9013924030
034630492634340800,5670048986634686922786117600,3420029547493938143902737600,197
7204582144932989443770175,1095067153187962886461165020,580717429720889409486981
450,294692427022540894366527900,143012501349174257560226775,663246383068634237960
47200,29372339821610944823963760,12410847811948286545336800,499881370203472652520
5100,191735320078044305076360,699574816500972464467800,242519269720337121015504
,79776075565900368755100,24865270306254660391200,7332066885177656269200,2041841
411062132125600,535983370403809682970,132341572939212267400,30664510802988208300
,6650134872937201800,1345860629046814650,253338471349988640,44186942677323600,71
10542499799200,1050421051106700,141629804643600,17310309456440,1902231808400,186
087894300,16007560800,1192052400,75287520,3921225,161700,4950,100,1>
```

17.9 Points on a Circle

Given three Cartesian points (x_1, y_1), (x_2, y_2), (x_3, y_3), write a program that determines the center x-coordinate, center y-coordinate, and radius of a circle traced through the points. That is, find h, k, and r in the equation

$$(x-h)^2 + (y-k)^2 = r^2 .$$

Demonstrate on $(4, 1)$, $(-3, 7)$, $(5, -3)$.

Solution

As single line:

```
func(circs,{
  sub({
     let(x1,elem([x],1))
    :let(x2,elem([y],1))
    :let(x3,elem([z],1))
    :let(y1,elem([x],2))
    :let(y2,elem([y],2))
    :let(y3,elem([z],2))
    :let(alpha,(([x2])^2)-(([x1])^2)+(([y2])^2)-(([y1])^2))
    :let(delta,(([x3])^2)-(([x1])^2)+(([y3])^2)-(([y1])^2))
    :let(beta,   -2*([x2])+2*([x1]))
    :let(gamma,   2*([y1])-2*([y2]))
    :let(epsilon,-2*([x3])+2*([x1]))
    :let(rho,     2*([y1])-2*([y3]))
    :let(machine,0.0001):
    :`Some division-by-zero prevention...'
    :let(beta,   iff(greater([machine],abs([beta])),
                    {[beta]+[machine]},{[beta]}))
    :let(gamma,  iff(greater([machine],abs([gamma])),
                    {[gamma]+[machine]},{[gamma]}))
    :let(epsilon,iff(greater([machine],abs([epsilon])),
                    {[epsilon]+[machine]},{[epsilon]}))
    :let(rho,    iff(greater([machine],abs([rho])),
                    {[rho]+[machine]},{[rho]}))
    :let(k,([alpha]/[beta]-[delta]/[epsilon])
          /([rho]/[epsilon]-[gamma]/[beta]))
    :let(h,([alpha]/[gamma]-[delta]/[rho])
          /([epsilon]/[rho]-[beta]/[gamma]))
    :let(rad,sqrt(([x1]-[h])^2+([y1]-[k])^2))
    :let(result,<[h],[k],[rad]>)
    :print_[result]
  })
})(<4,1>,<-3,7>,<5,-3>)
```

Output

```
<-6.045454545454544,-3.6363636363636367,11.063770821622635>
```

17.10 Square Root Table

Write a program that generates the square root of each integer up to a cutoff value.

Solution

As structured program:

```
include(`../prg/largecalc.txt'):

func(rtgen,{sub({
  let(a,unf([x])):
  print_{$({([x][l^]2)[l-]{a}})}
})}):

func(lsqrt,{
  lnewton(
    $({unoccult(rtgen([x]))})([y])
    ,1+[y]/2,[z])}):

print_
for(<i,1,25,1>,{<unf([i]),
  lsqrt([i],1+[i]/2,20)>,\n})
```

Output

```
<1,+1.0>,
<2,+1.4142135623730950488016887242096980785696718753769480731766797379907324784621>,
<3,+1.7320508075688772935274463415058723669428052538103806280558069794519330169088>,
<4,+2.0>,
<5,+2.2360679774997896964091736687312762354406183596115257242708972454105209256378>,
<6,+2.4494897427831780981972840747058913919659474806566701284326925672509603774573>,
<7,+2.6457513110645905905016157536392604257102591830824501803683344592010688232302>,
<8,+2.8284271247461900976033774484193961571393437507538961463533594759814649569242>,
<9,+3.0>,
<10,+3.1622776601683793319988935444327185337195551393252168268575048527925944386392>,
<11,+3.3166247903553998491149327366706866839270885455893535970586821461164846426090>,
<12,+3.4641016151377545870548926830117447338856105076207612561116139589038660338176>,
<13,+3.6055512754639892931192212674704959462512965738452462127104530562271669482930>,
<14,+3.7416573867739413855837487323165493017560198077787269463037454673200351563069>,
<15,+3.8729833462074168851792653997823996108329217052915908265875737661134830919369>,
<16,+4.0>,
<17,+4.1231056256176605498214098559740770251471992253736204343986335730949543463376>,
<18,+4.2426406871192851464050661726290942357090156261308442195300392139721974353863>,
<19,+4.3588989435406735522369819838596156591370039252324449368903441381595573282031>,
<20,+4.4721359549995793928183473374625524708812367192230514485417944908210418512756>,
<21,+4.5825756949558400065880471937280084889844565767679719026072421239068684255477>,
<22,+4.6904157598234295545656301135444662805882283534117371536057018910170246327532>,
<23,+4.7958315233127195415974380641626939199967070419041293464853091144482572359074>,
<24,+4.8989794855663561963945681494117827839318949613133402568653851345019207549146>,
<25,+5.0>
```