The Shang Programming Language

Xiaorang Li

December 22, 2009

Contents

Chapter 1

Overview of Key Features of Shang

Shang is a feature-rich high-level programming language and computing environment. It unifies the essences of many diverse and perhaps conflicting programming language concepts and paradigms, such as procedural, functional programming, and object-oriented programming, etc, and offers many features including first class parameterizd functions, partial substitution, domains, conditional classes, and automatons. Shang combines the strengths of both dynamic typing and static typing.

The goal of design is to make the language very easy to read and learn, and the syntax as intuitive, concise, and clean as possible. Unnecessary keywords, counter-intuitive and complicated syntax rules, and any other kind of inelegant kludges are never introduced into the design.

Shang supports object oriented programming with a number of annovative features. Access to class member attributes can be controlled and member attributes can have domains which protect the integrity of member data and make class interfaces more expressive and readable, and often enough to completely specify the class. Class membership validator and conditional class provide two clean solutions to the well-know "ellipse-circle" difficulty associated with traditional object programming.

Shang is meant to be a gereral-purpose programming language. Yet it is equipped with built-in features for efficiently handling scientific computations, many of which are not found in other popular numerical softwares, such as sets, matrices of infinite size integers and arbitrary precision floating point numbers, internal support for matrices of special patterns such as banded matrices (besides general sparse matrices), handle parameterized functions, etc.

For executing function calls and large loops the Shang interpreter is very fast, compared to other interpreted programming languages (except when the so called Just-In-Time compiler is used; in which case the program is running in compiled mode). This is achieved before any substantial effort is made on

optimization. Our next goal is to bring the interpreter close to the speed of compiled languages.

1.1 Multi-faceted Data

Data types and classes don't form a rigid hierarchical system. A piece of data is usually multi-faceted — meaning that it is not restricted to the functionality related to a single type and can play different roles. For example, (almost) anything is a function, a function is also a set, a set is also a function, a class is also a set, etc. This often makes it possible to implement functionalities by the most convenient and concise way, yet maintains a uniform interface to the client functions.

1.2 First-class object only

Every entity including a function and a class is a value and first-class object. Functions and classes can be used anywhere a value is expected. They can be defined inside functions, can be passed to functions as input arguments, and created and returned by functions as outcomes of function calls. New functions can be created not only by writing code, but also by operations on existing functions such as addition/subtraction, multiplication/division, composition, partial calls, and function vector/matrix.

1.3 Domains

Function input and output arguments may have domains and the language interpreter automatically checks the value of the arguments and generates domain error if the value is not in domain. Shang combines the advantages of both statically and dynamically typed languages. It is as flexible as dynamically typed language, yet it can be more specific than statically typed language, because what it requires is the input argument is inside a domain, not just of a type. A domain is a set that can be defined by different ways, and can be very specific and can often completely ensure the validity of input data, while static typing is often inadequate in this respect.

1.4 Function Parameters

Functions can have parameters in addition to input/output arguments, which makes functions customizable after they are created, and simplifies the calling sequence in many cases. Functions with parameters act like a class-less objects and spawn new functions like its own constructor.

1.5 Class and Members

Shang is objected oriented and has full support for most of the OOP features including access control of member attributes and multi-inheritance. Each attribute of a class can have a domain so that public class attributes can often be used for convenience yet object integrity is still protected. This can help eliminate the need of private attributes and "setter/getters" and make the class interfaces both safe and clean. The attributes a subclass inherited from super classes may violate the validity of the member of sub class. A class may have a validator to guard against illegal actions performed by super class attributes.

Shang has also extended the traditional concept of class to *conditional class*. A conditional class is a collection of loosely connected objects. Unlike a traditional class, it doesn't "create" new members using the constructor, but issues membership to members of other classes that satisfy certain conditions. Such memberships may be cancelled once the conditions are no longer satisfied, or the member can choose to withdraw from the class voluntarily. By using conditional classes, it is possible to avoid unnecessary programming complexity, too many levels of multiple inheritance, and frequent object creations and destructions.

1.6 Vectorization

In Shang all the numerical data are consistently represented in matrix format. A scalar is just a one by one matrix. Common matrix operations are supported directly. General data values are also vectorized. Every value is a list; when it is not defined as a list, it is considered a list of one element.

1.7 Rich features for numerical computing

While being a general purpose programming tool, Shang matrices of two and multi dimensions of various storage types including infinite size integers and arbitrary precision floating point numbers for efficient numerical computing. Extensive collection of matrix functions are built-in. Many other constructs such as functions, sets, lists, and tables also make it more convenient and efficient to express computing models and algorithms.

1.8 Minimized Implicit Behavior

Programing languages that use object references exclusively provide no way to directly handle objects, and cannot do neither passing by value nor passing by reference properly. This promotes implicit behavior of programs and poses particular difficulties when implementing structures like sets which are supposed to maintain a constant value unless changed by their owner. In Shang data values instead of their references are stored in variables therefore a variable's status

never changes unless a new value assigned to it, or explicit modifying operation performed on it, and therefore implicit behaviors of programs are reduced. On the other hand, safe and effective pointers are implemented with clear and readable syntax, to provide means to build complicated data structures.

1.9 Automaton

Shang provides automaton – a special data type that is a program with its own variables, whose execution can be halted and resumed, with running status and values of local variables retained. Automatons function as computers and can communicate with each other. They can help implement complicated control flows and event-driven programs.

1.10 High Performance

Our preliminary partial tests show that Shang interpreter is probably more efficient than most other interpreted programming language for executing loops and recusive function calls. That means it can be used as a real programming language and programmers don't need to be advised to avoid recursions and loops.

1.11 The Interactive User Interface

The interactive command console supports multi-line command editing with history browsing, so that whole blocks of commands can be edited and retrieved from command history. Full colored syntax highlighting makes the command editing easier, and the display more readable and less dull.

Chapter 2

The Shang Interpreter

2.1 The interactive mode

When the Shang executable program is invoked, a Shang session is launched. The initial mode of the session is an interactive programming environment, in which Shang commands are executed and answers displayed.

In the interactive mode the interpreter displays the prompt sign "`>>`" to indicate that it is waiting for the user to enter a command.

```
>>
```

When the user types a command after the prompt and hits `Enter`, the interpreter will examine the command. If it contains no syntax error and can be carried out, the interpreter will perform some necessary calculations and show the result and display the prompt sign again.

```
>> cos(pi)
   -1
>>
```

The interpreter recognizes `pi` as a system defined global variable that stores the value of the mathematics constant π, therefore evaluates `cos(pi)` to the cosine of π. Otherwise, if it doesn't recognize a symbol, it will print an error message and wait for the next command.

```
>> cos(pi)
   -1
>> cos(PI)
   Error: line 2, symbol "PI" not defined
>>
```

A session can be viewed as a stack whose top level is the interactive mode. If user-defined functions are invoked, the session enters lower levels of the stack. The local variables defined in the interactive mode cannot be accessed on lower

levels of the stack. Each level of the stack has its own work space to store local variables and won't interfere with each other. When a function call is finished, the work space is deleted and the active level of the stack is restored to the previous level. For example, during the execution of the following commands, the two variables both named `x` belong to different levels of stack and won't interfere with each other.

```
>> x = 10;
>> f = function y -> z
            x = sqrt(1 + y^2);
            z = 1 / x;
       end
>> f(10)
    0.09950371902
>> x
    10
```

2.2 Use Shang as an arbitrary precision calculator

The interpreter can be used as a calculator. An arithmetic expression can be entered like in other programming languages, and after enter is hit, the interpreter will return the value of the expression. An expression is similar to those found in math books and can contain numbers and operators +, -, *, and / that represent addition, subtraction, multiplication respectively, and division, and parentheses can be used for grouping. For example

```
>> (1 + sqrt(5)) / 2
   1.618033989
```

The value of `a^b` is a raised to the power of b.

```
>> 81^(1/2)
    9
```

Most common elementary functions such as `sqrt, exp, log, sin, cos, tan, asin, acos`, and `atan` can be used in the expressions.

Complex numbers are supported directly. A complex number of real and imaginary parts `a` and `b` is displayed as `a+bi`, and can be entered as either `a+bi, a+bI, a+bj`, or `a+bJ`. For example

```
>> sqrt(-5)
    0 + 2.236067977i
>> (3+5i) / (2-3j)
    -0.6923076923 + 1.461538462i
```

If a constant has the `M` suffix, it is treated as a multi-precision floating point number. By default, it is stored with 128 binary digits. For example

```
>> (1 + sqrt(5M)) / 2
   1.6180339887498948482045868343656 3E0
```

2.3 Define Variables

Variables can be defined using the equal sign. For example

```
>> h = 1.25
```

creates a variable with name "h" and assign value 1.25 to it. If there is already a variable named "h", its old value will be updated to 1.25.

A variable name can be a string of letters and digits, and underscore _, but cannot begin with a digit.

2.4 Define and call functions

To define a simple one-liner function, one can use the sign `->`

```
>> f = x -> (1 - x) / (1 + x + x^2)
```

A defined function can be called in the usual way

```
>> f = x -> (1 - x) / (1 + x + x^2)
   user defined function
>> f(-1)
   -2
```

A function can have several variables

```
>> f = (r, theta) -> r * (1 + cos(theta));
>> f(2, pi)
   0
```

More complicated functions have to be defined using the keyword `function`.

2.5 Suppress the display using semicolon

Usually the result of a command is displayed after `Enter` is hit. If the command is followed by a semicolon at the end, the answer will not be displayed.

```
>> h = 1.25;
>>
```

By suppressing unwanted displays, the command window can be kept cleaner and more efficient.

2.6 Commenting

2.6.1 Single line comment

All the characters in a line following the symbol `//` are ignored. For example

```
>> // this is a comment
>> sqrt(-3)   // this is a comment as well
   0 + 1.732050808i
```

2.6.2 Multi-line comment

Comments can run multilines as well. Any characters between a pair of `/*` and `*/` are ignored.

```
>> sqrt(-3) /* square root of -3 --- this is a comment
               it will be a complex number --- this is still a comment
               the answer is double precision --- yet another comment
               we're now done commenting */

   0 + 1.732050808i
```

Shang interpreter can recognize five levels of nested comments.

2.7 Run a script

A script file is a text file that contains a sequence of Shang commands. For example, a file with name "testscript.txt" may contain the following lines

```
f = (h, r, theta) -> r^2 + h * (1 + cos(theta));
h = 3.5;
r = 5;
theta = pi / 2;
f(h, r, theta)
```

In the interactive mode, if the following command is issued

```
>> run("testscript.txt");
   28.5
>>
```

All the commands in the script file will be executed as if they were just typed in.

2.8 Line continuation

If a line is ended with three dots, then the current line and the next line will be joined by the interpreter to form a single line. This can help enter very long lines.

```
f = (h, r, theta) ->  ...
           r^2 + h * (1 + cos(theta));
```

2.9 Command editing and History

Using the up and down arrow keys the previously entered commands can be brought up for editing and entered again.

Chapter 3

Data Type

In this chapter we will introduce the different structures that can be used to represent and store data. In Shang a piece of data is anything that can be assigned to a variable. In particular, a *function* or a *class* is just like a *number* or a *string*, and can be stored in a variable. Therefore by data we refer to both static information such as numbers, matrices, and character strings, as well as functions, classes, and running programs.

3.1 Data value and attribute

3.2 Numerical Data

All numerical data are in matrix format. Therefore a scalar is also a 1×1 matrix. There is no distinction between vector and matrix either. Also supported is multi-dimensional matrix of double precision floating point numbers.

3.2.1 Scalar

A number can be entered literally in decimal, binary, or hexadecimal format. For binary and hex format, the prefixes `0b`, `0B`, `0x`, or `0X` should be used.

```
>> x = 1011.11    /* decimal format */
   1011.11

>> y = 0B1011.11  /* binary format */
   11.75

>> z = 0X1011.11  /* hex format */
   4113.0664060

>> w = -0Xabcd.ab /* hex format */
   -214375.2578
```

3.2.2 Storage type of numerical values

Numerical scalars and elements of matrices can be of the following storage types:

`double`: double precision floating point number. The value is between ±`realmax`, where `realmax` is a built-in constant. A plain constant without suffix such as `123` or `-11.75`, is always stored as a `double`.

`int`: an integer of the natural size supported by the machine. The value is between two built-in constant `intmin` and `intmax` inclusive. A numerical literal with suffix `Z`, such as `37Z` or `-98Z` is an `int` constant. Note that without suffix an integer constant is considered a `double` constant instead of an `int`.

`byte`: a small integer (one byte) with value between 0 and 256. The suffix for `byte` is `B`.

`long integer`: an integer of arbitrary size. The value can be any integer as long as the computer has enough memory to store it. A `long` constant is written with a terminal `L`, as in `325598728592935528L`.

`mpf`: software floating point number with modifiable precision. The internal binary format of such a number is

$$\pm 0.a_1 a_2 ... a_n \times 2^e$$

where a_i equals either 0 or 1, with $a_1 = 1$ unless all a_i's are zero, in which case the value of the number is zero. The value of n is controled by global variable `mpf_ndigits`, and $e + k$ is a long integer. The suffix for an `mpf` constant is `M`. For example

```
x = 3.14159265358979323846264338327950M
```

The default value of `mpf_ndigits` is 128, therefore an `mpf` may have 128 significant binary digits as opposed to 52 for `double`. The value of `mpf_ndigits`. can be set to a multiple (at least 2) of 32.

Note: none of `double`, `int`, `long`, `mpf`, or `byte` is a keyword. There are several *sets*, namely `_D`, `_Z`, `_B`, `_M`, and `_L`, which are also built-in functions. They can be used to create matrices of various storage types, or test if a scalar or the element of a matrix belongs to a particular type.

3.2.3 Matrix

A matrix is a rectangular array of numbers. it can be created with elements included in a pair of square brackets. The rows are separated by semicolons, while elements in the same row are separated by commas.

```
>> A = [1,4, 9; 2, 3, 5; -2, 5, 10]
   1   4   9
   2   3   5
  -2  5    10
>>
```

Create Matrices

Alternatively, a matrix of a required size can be created and initialized using built-in functions `zeros`, `ones`, or `rand`. The command

```
A = zeros(3, 5)
```

will return a matrix of three rows and five columns, with each element being zero. Similar usage of `ones` and `rand` will create matrices of 1's and random numbers (between 0 and 1) respectively.

These three functions can also be called with a single parameter, in which case the second parameter is assumed to be 1, and thus a column vector is created.

```
>> B = rand(5)

    0.289
    0.353
    0.154
    0.566
    0.821
>>
```

By the `dimension` of a matrix we refer to the number of rows and the number of columns. For example, the dimension of the scalar `-5` is 1×1, while the dimension of

```
   -2    3    9
    10   1    -2
```

is 2×3.

Create Even Spaced Vectors Using the Colon Operator

The symbol `:` can be used to create a row matrix whose elements are evenly spaced. The default step-size of the vector is 1, which is assumed when one colon is used.

```
>> A = -1 : 5
   -1  0  1  2  3   4   5
```

To specify a step-size other than 1, two colons are needed.

```
>> A = 3 : 0.5 : 5
   3  3.5  4  4.5  5
```

Scalar, vector, and matrix

Every numerical value is treated as a matrix. It is only for convenience that sometimes we call some special matrices scalars or vectors. There is no distinction between a scalar, a, row or column vector of length 1, or 1×1 matrix. Likewise, a row vector of five elements is the same as a 1×5 matrix, and a column vector of five elements is the same as a 5×1 matrix.

3.3 Matrix Indexing

One element or a group of elements of a matrix can be referenced, extracted, or modified by an index expression. An index expression can have a single part, two or more parts separated by commas, or two parts separated by semicolons.

Each part of an indexing expression can be either an integer scalar or a matrix of integers. An index is an integer no less than 1. Zero is not a valid index.

3.3.1 Single Index

If `A` is a vector ($n \times 1$ matrix or $1 \times n$ matrix), then `A[k]` is naturally the `k`th element of `A`. The index doesn't have to be a scalar. If `K` is a matrix itself, then `A[K]` would be a matrix of the same dimensions. For example

```
>> A = [2, 3, 5, 7, 11, 13, 17, 19, 23];
>> A[3]
      5
>> K = [1, 3, 5, 7];
>> A[K]
      2  5  11  17
>> J = [1, 3; 5, 7]
>> A[J]
    2   5
    11  17
```

Even if `A` is not a vector, it is still possible to use a single index expression to `A`. If matrix `A` has `c` columns, and $k = ic + j$, then `A[k]` refers to the element by of `A` at i-th row and j-th column. In other words, `A[k]` is the k-th element of the row vector obtained by horizontally joining all the rows of `A`.

```
>> C = rand(3,5)
    0.802     0.716     0.262     0.752     0.925
    0.65      0.489     0.327     0.859     0.655
    0.396     0.329     0.941     0.854     0.857
>> C[7]
    0.489
>> C[10]
    0.655
```

The indexing expression can be a matrix itself. In this case a matrix with the same size as that of the indexing matrix is created, whose elements are the elements of the indexed matrix at the positions specified by the elements of the indexing matrix.

```
A = rand(7)
K = 1 : 2 : 7
A[k]

A = rand(5, 3)
K = [1, 3; 5, 7]
A[k]
```

3.3.2 Two indices separated a comma

In an index expression like `A[i, j]`, we call `i` the row index and `j` the column index.

If both `i` and `j` are scalars, then `A[i, j]` refers tot he element of `A` at i-th row and j-th column.

Such an indexing expression will return a square block of the indexed matrix. For example, `A[3:5, 7:10]` returns a 3×4 matrix whose elements are all the elements of `A` that lie on row 3, 4, 5 and column 7, 8, 9, and 10, namely

$$\texttt{A[3:5, 7:10]} = \begin{bmatrix} \texttt{A[3,7]} & \texttt{A[3,8]} & \texttt{A[3,9]} & \texttt{A[3,10]} \\ \texttt{A[4,7]} & \texttt{A[4,8]} & \texttt{A[4,9]} & \texttt{A[4,10]} \\ \texttt{A[5,7]} & \texttt{A[5,8]} & \texttt{A[5,9]} & \texttt{A[5,10]} \end{bmatrix}$$

3.3.3 Two indices separated a semicolon `A[i; j]`

In an index expression like `A[i; j]`, we still call `i` the row index and `j` the column index. If both `i` and `j` are scalars, then `A[i; j]` is the same as `A[i, j]`. Otherwise, one entry of the row index `i` matches one or more entries of the column index `j`. There are three types of match.

Firstly, if the row and column indices are vectors of the same length, then the two parts form matched pairs – The result of the index expression is a column vector of elements of the indexed matrix whose row and column indices are specified by `i` and `j`, namely, the numbers `A[i[1], j[1]]`, `A[i[2], j[2]]`, ...

For example, if both `i` and `j` have five elements, then the expression `X[i; j]` refers to the following matrix

$$\begin{bmatrix} \texttt{X[i[1], j[1]]} \\ \texttt{X[i[2], j[2]]} \\ \texttt{X[i[3], j[3]]} \\ \texttt{X[i[4], j[4]]} \\ \texttt{X[i[5], j[5]]} \end{bmatrix}$$

In particular, if `A` is an 6×6 matrix, then `A[1:6; 1:6]` returns the main diagonal of `A`, and `A[2:6; 1:5]` returns the first sub diagonal of `A`.

Second, if the length of the row index equals the number of rows of the column index then each entry of row index matches a row of the column index. For example, `A[1:3; [1,2,3; 2,3,4; 3,4,5]]` would form the following matrix

$$\begin{bmatrix} \texttt{A[1,1]} & \texttt{A[1,2]} & \texttt{A[1,3]} \\ \texttt{A[2,2]} & \texttt{A[2,3]} & \texttt{A[2,4]} \\ \texttt{A[3,3]} & \texttt{A[3,4]} & \texttt{A[3,5]} \end{bmatrix}$$

If the number of columns of the row index equals the length of the column index then each column of row index matches one entry of the column index. For example, `A[[[1;2;3],[2;3;4], [3;4;5]]; [1,2,3]]` would form the following matrix

$$\begin{bmatrix} \texttt{A[1,1]} & \texttt{A[2,2]} & \texttt{A[3,3]} \\ \texttt{A[2,1]} & \texttt{A[3,2]} & \texttt{A[4,3]} \\ \texttt{A[3,1]} & \texttt{A[4,2]} & \texttt{A[5,3]} \end{bmatrix}$$

3.3.4 Using backslash \ to reference diagonals

If A is a square matrix, then

- `A[\]` returns the diagonal (as a column matrix) of `A`.
- `A[\,0]` also returns the diagonal of `A`.
- `A[\,k]` returns the k-th super-diagonal if $1 \leq k < n$.
- `A[\,k]` returns the $(-k)$-th sub-diagonal if $-n < k \leq -1$.

In the expression `A[\,k]`, `k` can be a vector as well. For example, if `A` is $n \times n$, `A[\, 1:n-1]` would return the elements of the upper triangular part of `A` (as a vector).

```
>> A = [1, 2, 3; 4, 5, 6; 7, 8, 9]

   1   2   3
   4   5   6
   7   8   9

>> A[\, 1:2]
   2
   6
   3

>> A[1,1:2] = 0

   1   0   0
   4   5   0
   7   8   9
```

If A and B are two 9×9 matrices, `A[\, 1:9]=B[\,1:9]` would copy the upper triangle of B to A.

3.3.5 $ stands for the largest index

In an index expression, the dollar sign $ stands for the largest possible value of the index. Suppose that A is an $m \times n$ matrix. Then in the single-part index expression, the value of $ is the size of the matrix, that is, $m \times n$. Then `A[$]` returns `A[m * n]`, the last element of A, and . `A[$-1]` returns `A[m * n - 1]`, the second last element of A.

If $ appears in the row index of a two-part index expression, its value is m, the number of rows of the matrix; if it appears in the column index, its value is n, the number of columns of the matrix. Therefore, `A[$, 2]` returns the element at the last row and the second column, and `A[1, $]` returns the element at the first row and last column.

The dollar sign can participate in arithmetic operations. For example, `A[$-2]` gives the third last element of A; and `A[1 : $, $ - 2]` returns the second last column of A. Note that the two $'s have different values in this expression.

3.3.6 : is equivalent to 1 : $

If an index consists of a single colon, then it is equivalent to `1 : $`. For example, `A[1, :]` returns the first row; `A[:, 3]` returns the third column, and `A[:]` returns the column vector obtained by joining all rows of A.

3.3.7 Index bound

Any elements of an index must be an integer no less than 1. Normally an index also has an upper bound, so that no reference to an non-existing element is made, unless the index expression is used as an lvalue (see next subsection). In particular, for an $m \times n$ matrix, a row index should be an integer within the range `[1, m]`, and a column index should be within the range `[1, n]`, and an index value in a single-part index expression should be within the range `[1, mn]`.

```
>> A = [1, 3, 5, 7, 11, 13, 17, 19];
>> A[9]
   Error 0 "9": index value 9 out of bound 1-8
>> A[9] = 23
   1   3   5   7  11  13  17  19 23
```

3.3.8 Index expression as lvalue

The lvalue is the expression of the left-hand side of an assignment expression. All the index expressions we have just described can be used as lvalues for modifying the contents of matrix. For examples

```
>> A = zeros(5,5);
>> A[2,3]=2.3
>> A[1:3, 1:3] = rand(3,3)
>> A[$]=100
```

Usually when an index expression is used as an lvalue, the dimension and size of the right hand side of the assignment should match that of the index expression to make the assignment possible. The only exception is when the right hand side is a scalar, then all the indexed elements of the matrix will be set to the same scalar value.

```
>> A = zeros(3,3)
   0   0   0
   0   0   0
   0   0   0
>> A[1, :] = 3
   3   3   3
   0   1   0
   0   0   1
>> A[\] = 1
   1   3   3
   0   1   0
   0   0   1
```

When updating the contents of a matrix, the indices don't have to be within the upper bounds, which makes it possible to make the size of the matrix grow. As the size of a matrix is extended, the new entries are set to zero, except for those being specified by the assignment statement. For example,

```
>> X = [1,4,9]
   1   4   9
>> X[4] = 16
   1   4   9   16
>> x[8]=36
   1   4   9    0    0    0   0   36
```

3.4 Storage types of matrix elements

When the three builtin-in functions `zeros`, `ones`, and `rand` are called to create matrix, a final optional argument can be used to specify the storage type of the elements of the matrix. The value of the argument can be _Z, _B, _L, or _M, for `int`, `byte`, `long`, and `mpf` respectively.

To create a 3×5 `int` matrix of 0's or 1's, use the command

```
x = zeros(3, 5, _Z);
```

or

```
x = ones(3, 5, _Z);
```

To create a column vector of 5 bytes of random values between 0 and 255, use the command

```
x = rand(5, _B);
```

or

```
x = rand(1, 5, _B);
```

To create a row vector of 5 binaries, use the command `binary(5)`.

3.5 Sparse Matrix

Using the regular dense storage scheme to store an $m \times n$ matrix of `double` needs $8mn$ bytes of memory. If a large number of the elements of the matrix are zeros, the dense storage scheme can be wasteful and inefficient. For large sparse matrices or matrices of special formats several more effective alternative storage formats can be chosen.

`sparse`: an $m \times n$ sparse matrix is created with `sparse(m, n)`.

`banded`: a square $n \times n$ matrix with l subdiagonals and u superdiagonals is created using `band(n, l, u)`.

`upper triangular`: a square $n \times n$ upper triangular matrix is created using `upper(n)`.

`lower triangular`: a square $n \times n$ lower triangular matrix is created using `lower(n)`.

`symmetric`: a symmetric $n \times n$ symmetric matrix is created using `lower(n)`.

All these commands give rise to matrices whose elements are all zeros. To make the matrices useful, normal assignment statements can be used to add non-zero elements.

`sparse` and `banded` matrices use efficient storage scheme and can save on memory, while `triangular` and `symmetric` matrices use the same storage scheme as dense matrix and thus woulldn't save on memory but may be more efficient when solving some linear algebraic problems.

Note that these special storage schemes are supported only for double precision floating point numbers.

3.6 Multi-dimensional matrix

A numerical matrix can also be multi-dimensional, whose each element is referenced by three or more indices. To create such a matrix, one has to use any of the three built-in functions `zeros`, `ones`, or `rand`, with three or more parameters to specify the size of each dimension. For example, the following command returns a matrix of 5 slices, with each slice having 3 rows and 2 columns.

```
>> rand(5,3,2);
```

An element or a slice of a multi-dimensional matrix can be referenced or reset using an indexing expression with appropriate number of indices. For example

```
>> x = rand(5,3,2);
>> x[1, :, :]; // the first slice of x
```

Note that the only storage type available for multi-dimensional matrices is double precision floating point number. Therefore type does not need no be specified.

If two multi-dimensional matrices `A` and `B` have the same dimensions, `A + B` and `A - B` are defined the usual way. The only other operation defined for multi-dimensional matrix is scalar multiplication.

3.7 Attributes of matrix

If `A` is an $m \times n$ matrix, then

`A.length` returns mn

`A.nrows` returns m

`A.ncolumns` returns n

`A.size` returns column vector $[m; n]$

`A.abs` returns a copy of `A` whose elements are absolute values of `A`

`A.norm` returns the vector norm (if `A` is a row or column vector), or matrix norm of `A`.

`A.sum` returns the sum of all elements of `A`

`A.max` returns the maximum of all elements of `A`

`A.min` returns the minimum of all elements of `A`

`A.range` returns `[m1, m2]`, where `m1` and `m2` are the minimum and maximum of `A`.

`A.mean` returns the average of all elements of `A`

`A.var` returns the variance of all elements of `A`

`A.stddev` returns the standard deviation of all elements of `A`

`A.rowsum` returns the sum of all each row of `A`

`A.rowmax` returns the maximum of each row of `A`

`A.rowmin` returns the minimum of each row of `A`

`A.rowrange` returns the range of each row of `A` `A.rowmean` returns the average of each row of `A`

`A.rowvar` returns the variance of each row of `A`

`A.rowstddev` returns the standard deviation of each row of `A`

`A.columnsum` returns the sum of all each column of `A`

`A.columnmax` returns the maximum of each column of `A`

`A.columnmin` returns the minimum of each column of `A`

`A.columnrange` returns the range of each column of `A` `A.columnmean` returns the average of each column of `A`

`A.columnvar` returns the variance of each column of `A`

`A.columnstddev` returns the standard deviation of each column of `A`

`A.sort()` sorts the elements of `A` in ascending order

`A.rowsort()` sorts each row of `A` in ascending order

`A.columnsort()` sorts each column of `A` in ascending order

`A.reverse()` reverses the order of all elements

`A.rowreverse()` reverses the order of all elements in each row

`A.columnreverse()` reverses the order of all elements in each column

`A.swaprows(j, k)` swaps row `j` and row `k` of `A`

`A.swapcolumns(j, k)` swaps column `j` and column `k` of `A`

`A.scalerow(j, alpha)` multiply row `j` of `A` by a scalar `alpha`

`A.scalecolumn(j, alpha)` multiply column `j` of `A` by a scalar `alpha`

`A.addrows(j, k, alpha)` add `alpha` times of row `k` to row `j` of `A`

`A.addcolumns(j, k, alpha)` add `alpha` times of column `k` to column `j` of `A`

3.8 Character String

A character string is created by enclosing a sequence of characters by a pair of double quotes or single quotes.

```
>> s = "Yankee Doodle went to town."
       Yankee Doodle went to town.
```

When using double quotes, the character \ acts as an escaping signal in order to include special characters in the string. The defined escaping sequences are the same as provided by the C programming language. The complete list is given in the table. Note that the backslash character doesn't have a special meaning in a single quoted string.

\a	alert (bell) character	\\	backslash
\b	backspace	\?	question mark
\f	formfeed	\'	single quote
\n	newline	\"	double quote
\r	carriage return	*ooo*	octal number
\t	horizontal tab	\x*hh*	hexadecimal number
\v	vertical tab		

3.8.1 Index and substring

A string can be used as if it were a one-dimensional array of characters. That is, an individual character or a substring can be accessed and modified by using an index expression.

```
>> x = "The flying pig"
   The flying pig
>> x[3:5]
   e f
>> x[1] = "E";
   Ehe flying pig
>> x[1:3] = "the";
   the flying pig
>> x[$-2:$] = "wig"
   the flying wig
```

Note that the $ sign when appearing in an index expression represents the length of the variable being indexed, therefore `x[$]` refers to the last character of the string.

ASCII values can be assigned to one or several characters of a string. For example,

```
>> x = "The flying pig"
```

```
>> x[12] = 119;
>> x
    The flying wig
>> x[12:14] = [98, 112, 103];
>> x
    The flying bug"
```

Note that the value assigned to string must be valid ASCII value; in particular, zero value is not allowed.

3.8.2 Attributes of a string

If `s` is a string, `s.length` returns the length of *s*. `s.reverse()` reverses *s*.

3.8.3 Use string as a function

When a string `s` is used as a function, the argument should also be a string `t`, and the index of occurrence of `s` as a substring of `t` is returned.

```
>> x = "The flying pig";
>> y = "pig";
>> y(x)

   12
```

3.8.4 Concatentation and other operations

A bunch of strings can be joined to make a longer one by puting them inside a pair of brackets, and separating them with commas. For example

```
>> ["Entering", " ", "Rose", " ", "County"]

   Entering Rose County
```

The same can be achieved by using the addition operator. For example

```
>> s = "Entering" + " " + "Rose" + " " + "County"

   Entering Rose County
```

If `n` is a positive integer and `s` is a string, `n * s` or `s * n` will create a new string which is `s` repeated `n` times.

```
>> s = "haha ... ";
>> 5 * s
   haha ...haha ...haha ...haha ...haha ...
```

If `s` and `t` are two strings of the same length, or one of them has length 1, then `s - t` returns a row vector of integers, which are the differences between the ASCII codes of `s` and `t`. For example

```
>> s = "bcdefg" - "a"
   1 2 3 4 5 6

>> s = "uvwxyz" - "UVWXYZ"
   32 32 32 32 32 32
```

If `s` is a string, and `x` is a integer scalar, or a vector of integers having the same length as `x`, then `s + x` returns a vector of the same length as `s` whose ASCII values are those of `s` plus the value(s) of `x`.

```
>> s = "BCDEFG" + 32
   bcdefg
```

Note `s - x` and `x - x` are defined in the same manner.

```
>> s = "bounding" - ("a" - "A")
   BOUNDING
```

When numbers or vectors are being added to or subtracted from a string, the resulting must still have characters within the range of ASCII values. Otherwise the operation is not carried out.

3.9 Regular expression

A regular expression object can be created with a tilde followed by a pair of slashes with regular expression in between. The following is a regular expression that matches one or more repeating `z`'s.

```
r = ~/zz*/
```

A few options can follow the closing slash. They are `m` for multi-line, `i` for ignoring case, `g` for finding all matches.

```
r = ~/zz*/gi
```

A regular expression can be used as either a function, or a set. To match a regular expression `r` against a string `s`, just call `r` as a function:

```
r(s)
```

The result is the starting and end indices of the match. If the `g` option of the regular expression is enabled, the result is an $n \times 2$ matrix with the indices of all the matches given as rows.

Because a function is also a set, so a regular expression can be used anywhere a set is expected, especially for specifying the domain of a function parameter, argument, or a class attribute. For example, the regular expression `/zz*/` can be considered as the set of all strings that contain `z`'s.

A regular expression `r` has the following attributes

- `r.pattern`

- `r.options`
- `r.match`

The `pattern` and `options` attributes can be accessed and reset, and the `match` attribute is a function that matches the pattern against the argument string.

3.10 Regular expression substitution

The function of a regular expression substitution is to substitute the occurrence of a pattern with a given string. A regular expression substitution object is created with a tilde followed by three slashes, with the regular pattern between the first two slashes, and the substitute string between the middle and the last. For example

```
s = ~s/zz*/oz/
```

Again, a few options can be given after the closing slash. They are `m` for multi-line, `i` for ignoring case, and `g` for finding all matches.

```
s = ~s/zz*/oz/
```

To apply a regular expression substitution, use it as function. It doesn't alter the argument string (like any Shang function), but returns an updated copy of the string, which can be assigned to the same variable if one wishes.

```
text = "Monkey in the zzoo"
s = ~s/zz*/oz/
text = s(text)
```

A regular expression substitution `s` has the following attributes

- `s.pattern`
- `s.substitute`
- `s.options`

The `pattern` and `options` attributes can be accessed and modified, and the `match` attribute is a function that matches the pattern against the argument string.

3.11 List

A list is a one dimensional array of data values of any different kinds. It is created with a sequence of elements separated by commas, and surrounded by a pair of parentheses. For instance

```
x = (2, 3, "My goldfish is evil");
```

Any data value can be a list element, including a list. For example

```
x = (2, [3, 5], x -> sqrt(x^2 + 1), "My goldfish is evil", ("a", "b", "c"));
```

Here the second element is a matrix, the third element is a function, the fourth element is a string, and the last element is a list.

To access any element, use the operator `#` followed by the index. The index value must be an integer starting from 1. For example, the third element is

```
x#3
```

Several elements of a list can be indexed at once using a vector index. The result is a still a list. For example

```
x#[1:3]
```

gives a list of the first three elements of the list, while

```
x#[1:2:9]
```

returns the list of odd-indexed elements up to the ninth.

To check the length of a list, one can use `#` followed by the list variable name

```
>> x = (2, 3, "My goldfish is evil");
>> #x
   3
```

`x.length` would achieve the same thing. However, if `x` is a vector of three elements, `x.length` would also return 3, while `# x` returns 1.

The last element of a list can be retrieved using the index `$`, and the next last using `$-1`, etc.

Note that Shang is *vectorized* on the object level. Therefore a list is *not* a so-called "container" in other languages. Any value that is not a list, is actually a list of one item. For example, the number -3 is also a list of one number -3.

```
x=-3;
y=(-3);
x == y
    1
```

Likewise, `-3`, `(-3)`, `((-3))`, and `(((-3)))` are all the same. And it's impossible to distinguish `(a, b)` from `((a, b))`.

3.11.1 Fixed length list and variable length list

A fixed length list has a fixed number of elements, while the number of elements of a variable length list can decrease or increase. A list defined normally has a fixed length, while a list defined with a trailing tilde has variable length. For example,

```
x=(100, "Kernobe", ~);
```

Note that the length of list `x` is 2, and it is equal to the fixed length list `y=(100, "kernobe")`, except that `x` can be expanded or shrinked. The tilde at the end of the list is not a special element; it just signals that the list is capable of changing length. For example

```
x=(100, "Kernobe", ~);
x#3="Schwartz";
   (100, Kernobe, Schwartz)
x#5="Dark Helmet";
   (100, Kernobe, Schwartz, [], Dark Helmet)
```

If you plan to create a list `x` by starting with an empty list and adding new elements one by one, the list should be initialized as

```
x=(~);
```

Note that both `(~)` and `()` are empty list, but new elements can only be added to `(~)`.

List plays a critical role in the operation of functions, as calling a function `f` with `f(a,b,c)` is equivalent to the multiplication operation between the function `f` and the list of arguments `(a, b, c)`.

Variable length lists have the following attribute functions

- To add an element at the end of a list `s`, one can use `s.append(x)`; to add an element at the beginning, one can use `s.prepend(x)`.
- To insert an element at the position `k`, one can use `s.insert(k, x)`.
- To delete the element at the position `k`, one can use `s.delete(k)`.
- To delete the last element, one can use `s.pop()`.
- To delete the first element, one can use `s.leftpop()`.
- To remove all repeated elements (leave only the first copy), one can use `s.unique()`.

For both fixed length and variable length lists

- To reverse the order of all elements, one can use `s.reverse()`.
- To sort all elements in increasing order, one can use `s.sort()`.

3.12 Hash Table

A hash is a set of values, with each value associated with a unique key. The value can be retrieved by providing the right key. To create a hash table, put an arrow `=>` in between each *key/value* pair, and put all pairs inside a pair of braces, separated by commas.

```
A = {"red" => "maple", "purple" => "lilac", "grey" => "ash"}
```

Each element of the table can be extracted and reset using the operator `@`.

```
>> A = {"red" => "maple", "purple" => "lilac", "grey" => "ash"};
>> A @ "purple"

   lilac
```

Note that a hash table can also act like a function, therefore to reference the entry of a table `A` with key `r`, one can use either `A@r`, or `A(r)`.

If one needs to create a finite function (whose domain is finite set), one doesn't have to write a bunch of code and can use a hash table instead.

```
>> f = {1 => 0, 2 => 0, 3 => 0, 4 => 1, 5 => 1, 6 => 1};
>> f(1)
      0
>> f(3)
      0
>> f(6)
      1
```

Values can be reset or added to the table using

```
A@r = new_value
```

Note that although both `A@r` and `A(r)` refer to the same value of the table, only `A@r` can be used as an lvalue in an assignment expression.

A hash table can have a `default` key, which matches anything that is not explicitly used as a key value. For example, if a table is created by

```
>> u = {1 => 2, 2 => 3, 3 => 4, 4 => 5, default => 0};
```

Then `u(x) = 0` for any `x` except `1, 2, 3, 4`.

3.13 Set

Set is a *concept* rather than a specific data type. It is a *facet* of any data value and virtually anything can be used as a set. For two values `x` and `s`, if any of the expressions `x in s`, `x (- s`, `s(x)`, or `s*x` evaluates to true (non-zero), then we consider that `x` to be a member of the set `s`. There several ways to create and use a set.

3.13.1 Finite Sets

A finite set can be created with a pair of braces with all the set members in between and separated by commas.

```
colors = {"red", "orange", "yellow", "green", "blue", "purple"}
faces = {1, 2, 3, 4, 5, 6}
```

To test if a value x is member of a set s, one can use

```
x (- s
```

or

```
x in s
```

or

```
s(x)
```

or

```
s * x
```

If `x` is a member of `s`, all of the above will return non-zero; otherwise `0`.

To add a new member to a set, use

```
s.add(m)
```

To remove a member from a set, use

```
s.remove(m)
```

The k-th element of a set can be extracted using

```
s[k]
```

However, the order of the members is not a property of the set. Therefore if `s[1]` is not equal to `t[1]`, it doesn't imply that `s` does not equal `t`. Providing the indexing function is for the purpose of looping over the whole set. Note also that set member can not be added or updated using the indexing expression.

3.13.2 Intervals

Interval of real numbers in mathematics can be can be represented using the keyword `to`. For example, the closed interval $[0, 1]$ can be created by

```
>> s = 0 to 1;   // interval [0, 1]
```

Note that since any single value is the same as a list whose only entry is that value. Therefore, `0 to 1` is equivalent to `(0 to 1)`, and we can write

```
>> s = (0 to 1);  // interval [0, 1]
```

Open and half open intervals are created using suffix `+` and `-`. For example

```
>> s1 = (0 to 1);   // interval [0, 1]
>> s2 = (0+ to 1-);   // open interval (0, 1)
>> s3 = (0+ to 1);   // half open interval (0, 1]
>> s4 = (0 to 1-);   // half open interval [0, 1)
>> s5 = (0 to inf);   // interval [0, inf)
>> s6 = (-inf to 1);   // interval (-inf, 1]
>> s7 = (-inf to inf);   // interval (-inf, inf) equivalent to _R
```

3.13.3 Define a set using function

Any function `f` can act like a set. If the logical value of `f(x)` is true, then `x` is considered a member of `f`, otherwise it is not. For example, the set of all the (floating point) numbers between `0` and `1` can be defined by

```
>> S = x -> (x >= 0 && x <= 1)
>> 0.5 (- S
     1
>> 1.5 (- S
     0
```

3.13.4 Set operations

- To check if `s` is a subset of `t`

  ```
  s (= t
  ```

- To check if `s` is a true subset of `t`

  ```
  s (< t
  ```

- To check if `s` is a super set of `t`

  ```
  s =) t
  ```

- To check if `s` is a true super set of `t`

  ```
  s >) t
  ```

  To calculate the union of two sets:

  ```
  s \/ t
  ```

- To calculate the intersection of two sets:

  ```
  s /\ t
  ```

- The following gives the intersection of `s` and the complement of `t`

  ```
  s \ t
  ```

All the above three operations can be performed on any objects as well as finite sets. If both operands are finite sets, the outcome is also a finite set. If any operand is not a finite set, the result is usually a function that can be used as a set. For example, if

```
u = s \/ t
```

then `u` is a function such that `u(x) = 1` if `x` is a member of either `s` or `t`, and zero otherwise.

The interpreter is capable of determining if `s` is a subset of `t` only when `s` is a finite set. It is done by testing each element of `s`. If `s` is defined by a function, such computation is impossible and would not be attempted. The same applies to other set operations that involve infinitely many steps of computations.

3.14 Stack

Stack is a type of collective data that stores data in a "first in, last out" manner. "stack" is not a keyword, but a built-in function name. To create an empty stack, call the stack function

```
s = stack();
```

To push an item x into the stack,

```
s.push(x);
```

To pop the top item out of the stack

```
x = s.pop();
```

To check the height of the stack

```
s.height;
```

3.15 Queue

Queue is a another type of collective data that stores data in a "first in, first out" manner. To create an empty queue, call the queue function

```
q = queue();
```

To add an item x into the queue,

```
q.enqueue(x);
```

To remove the next available item in the queue

```
x = q.dequeue();
```

To check the length of the queue

```
s.length;
```

3.16 Structure

A structure is a group of attribute values with each one associated with an identifier. It is very easy to create by using a pair of braces. Each attribute is declared and initialized by assigning the value to the identifier, and all attributes are enclosed in a pair of braces. For example

```
dims = {length = 15, width = 10, height = 9};
```

Here a structure with three attributes is created. Note that each part of the definition corresponds to a structure attribute. The identifier on the left side of the assignment sign declares the name of an attribute, and will not be confused with any local variable in the surrounding scope. The expression to the right of the assignment sign is evaluated in the surrounding scope. For example

```
length = 17
height = 28;
dims = {length = 15, width = length, height = height};
```

In definition of the `dims` structure, when `width = length` is processed, `width` declares a new structure attribute, where `length` is the value of the local variable `length` and has nothing to do with the previously defined attribute of the same structure, which is also named `length`. In `height = height`, the first height declares a new attribute, while the second will be the value of the local variable `height`.

An attribute of a structure can be accessed, added or reset using the dot operator `.`

```
>> dims = {length = 15, width = 10, height = 9};
>> dims.height
   9
>> dims.height = 12
>> dims.height
   12
```

Note that since function is also a data type, attributes of a structure can be functions as well. For example

```
>> region = {
              shape  = "circular",
              center = [2, -3],
              radius = 15,
              verify = (x, y) -> (x - 2)^2 + (y + 3)^2 <= 225
          };

>> region.verify(3, 2)
   1
```

One obvious use of structure is to return more than one values in a function. For example

```
dstats = function x -> s
              n = length(x);
              min = max = sum = x[1];
              for k = 2 : n
                  sum += x[k];
                  if x[k] < min
                       min = x[k];
                  end
                  if x[k] > max
                       max = x[k];
                  end
              end
              mean = sum / n;
              sd = 0;
              for k = 1 : n
                   sd += (x[k] - mean)^2;
              end

              sd = sqrt(sd / (n - 1));

              s = {
                     n = n, min = min, max = max,
                     mean = mean, sd = sd
                  };
          end
 x = rand(15);
 dstats(x)
```

In order to make a copy of a structure, we only need to assign the structure to another variable.

```
>> dims = {length = 15, width = 10, height = 9};
>> newdims = dims;
>> newdims.width = 30;
>> dims.width
   10
>> newdims.width
   30
```

A structure encapsulate several pieces of data and behaves like a class member. One may regard it as an object without a class, or a quicker way to create an object. Structures lack the more advanced features of classes, but are often sufficient and more convenient. Another important use of these classless objects is to provide 'candidates' for conditional classes (chapter 8). They can "apply" for memberships of conditional classes and then be able to use the rich functionalities offered by the classes.

3.17 Other Data Types

In most programming languages, programming structures such as functions and user defined types are treated differently from normal values and are not considered data types. In Shang, any entity has a value and can be stored in a variable. Apart from the types discussed above, the following are all data types:

- functions (user defined, built-in, pseudo)
- classes
- members of classes
- automatons (running functions)

They can be assigned to variables, and be entries of lists, and be passed to functions as arguments values. Functions are discussed in 5.9, classes and members are discussed in 6.16.4, and automatons in 11.4.3.

Chapter 4

Operators

An expression is a sequence of constants and variables combined by operators that produces a new value. The constants and variables are called the *operands* of the operator. Many operators take one or two operands, where other operators may take more. The construction of expression is recursive, therefore an operand can be an expression itself. For example, in expression `a + b / c`, the second operand of `+` is an expression `b / c`.

4.1 Arithmetic Operators

Arithmetic operators manipulate numerical data. Most of them are the usual ones learned in elementary math, and work the familiar way. However, they take not only numbers but vectors, matrices, and other data as well, and there are a few operators designed for matrices only.

4.1.1 Addition and Subtraction: `+`, `-`

Numerical Operands

The expression `x + y` produces the sum of `x` and `y` when they are two scalars (either doubles, integers, or bytes).

If `A` and `B` are matrices of the same dimensions, `A + B` is a matrix of same dimension whose entries are the sums of the corresponding entries of `A` and `B`.

If either `A` or `B` is scalar (`1 x 1` matrix), then the scalar is added to each entry of the other matrix.

If `A` is an $m \times n$ matrix and `B` is $m \times 1$ row vector, when `A+B` is calcuated, the kth element of `B` is added to each element of the kth row of `A`.

Similarly, if `A` is an $m \times n$ matrix and `B` is $1 \times n$ column vector, when `A+B` is calcuated, the kth element of `B` is added to each element of the kth column of `A`.

If the dimensions of `A` and `B` don't match in any of the ways described above, `A + B` will cause an error. For example

```
x = [-2.5, 3; 9, 7]
    -2.5    3
       9    7
y = [1, -2; 3, -5]
       1    -2
       3    -5
z = x + y
       -1.5  1
         12  2
```

If `A` and `B` are of the same storage type, then `A + B` is of the same type. Otherwise the type of the result is same as the one that needs more storage space. For example, if a double matrix and a integer matrix are added, the result is a double matrix.

Subtraction `A - B` is defined in the similar manner.

Functional Operands

If either of `A` and `B` is a function, `A + B` or `A - B` will be a new function. For example

```
>> f = sin + cos;
   f(pi / 4)
        0
>> h = x -> sqrt(x^2 + 1);
>> h1 = f + h;
>> h1(0)
   2

>> g = 2 - sin;
>> g(pi/2)
   2.141592654
```

Note that the definition of `g` is $g(x) = 2x - \sin x$ instead of $g(x) = 2 - \sin x$. Therefore $g(\pi/2) = 2(\pi/2) - \sin(\pi/2) = \pi - 1$.

List Operands

If `M` and `N` are two lists of the same length, then `M + N` is a list of the same length, whose elements are the sums of the corresponding entries of `M` and `N`.

```
>> M = (3, 5, 9, [2, -3]);
>> N = (1, -7, 0, [-2, 5]);
>> M + N
   (4, -2, 9, [0, 2])
```

The subtraction of one list from another list is defined similarly. If the lengths of the two lists are different, or if any two corresponding elements can not be

added, then the addition of the two lists is undefined and attempt to it will cause an error.

Note that in some languages, adding two lists means concatenating them to make a longer list. In Shang, the operator for concatenating two lists is `>-<`.

4.1.2 Multiplication: `*`

If `A` and `B` are two scalars, then `A * B` is the product of `A` and `B`.

If one of `A` and `B` is a scalar and the other is a matrix, the result is a matrix of the same size, whose entries are the products of the value of the scalar and the entries of the matrix. For example

```
>>   x = 2;
>>   y = -9;
>>   x * y
      -18
>>   A = [3, 5; 2, -8];
>>   3 * A
       9     15
       6    -24

>>   A * 7
       21    35
       14   -56
```

If `A` and `B` are non-scalar matrices, then `A * B` is the matrix multiplication of `A` and `B`. This is defined only when the sizes of `A` and `B` match, i.e., the number of columns of `A` equals the number of rows of `B`. For example

```
>>   A = [3, 5; 2, -8; 1, 9];

       3   5
       2  -8
       1   9

>>   B = [2, 1, 7; -2, 0, 8]

       2   1   7
      -2   0   8

>>   A * B
       -4   3   61
       20   2  -50
      -16   1   79

>>   B * A
       15  65
```

```
      2  62
```

Attempt to multiply matrices whose sizes do not match will cause an error.

If `A` and `B` are two lists with the same length n, then `A * B` is the dot product `(A#1) * (B#1) + (A#2) * (B#2) + ... + (A#n) * (B#n)`.

If `B` is a list with length n, and `A` is a list whose length is m and whose each element is again a list with the same length n, then `A * B` is the list of length m: `((A#1)*B, (A#2)*B, ..., (A#n)*B)`.

4.1.3 Division: /

If both `A` and `B` are scalars, then `A / B` is the quotient of `A` and `B`. No matter they are doubles or integers, the quotient is always a floating point number of double precision. If `A` is not equal to zero, but `B` is, then `A / B` is `inf`. If both `A` and `B` are zero, then `A / B` is `nan` (meaning *not a number*).

If one of `A` and `B` is a scalar, while the other is a matrix, then `A / B` is a matrix of the same size, with each element being `A` divided by the each entry of `B`, or each entry of `A` divided by `B`, depending which one is a scalar.

If both `A` and `B` are matrices, then `A / B` is the (numerical) solution of matrix equation `X B = A`. The matrix `B` must be an `n` $\times$ `n` non-singular square matrix and `A` must have `n` columns. Note that `X B = A` is solved numerically, so the solution can only satisfy the equation approximately.

```
>>  A = [3, 2, -5];
>>  B = [1, 2, -3; 2, 1, 5; 0, -1, 3];
>>  X = A / B
      11  -4  16
>>  X * B
      3    2  -5
```

4.1.4 Back Division for solving linear system: \

Numerical Operands

If both `A` and `B` are matrices, then `A \ B` is the (numerical) solution of matrix equation `A X = B`. The matrix `A` must be an `n` $\times$ `n` non-singular square matrix and `B` must have `n` rows. For example

```
>>  A = [1, 2, -3; 2, 1, 5; 0, -1, 5];
>>  b = [3; 1; -2];
>>  x = A \ b
     0.75
     0.75
    -0.25

>>  A * x
    3
    1
```

```
   -2
>> norm(A*x-b)
    2.220446049e-16
```

Note that Ax is only approximately equal to b.

Set Operands

If both `A` and `B` are finite sets, then `A \B` is the difference between set `A` and `B`:

$$A \backslash B = \{x \in A, x \notin B\}$$

For example

```
>> A = {3, -5, "abc", 9, 10, -7};
>> B = {9, -3, "abc", -7, -3};
>> A \ B
    {3, -5, 10}
```

4.1.5 Element-wise multiplication and division: `.*`, `./`

If `A` and `B` are two matrices of the same size, then `A .* B` and `A ./ B` are the matrices of the same size, whose entries are the products/quotients of the corresponding entries of `A` and `B`. For example

```
A = [1, 2, 3, 4, 5];
A .* A
    1   4   9  16  25
A = [1, 2; 3, 4];
B = [1, 1; 2, 2];
A .* B
    1     2
    6     8
A ./ B
          2
   1.5    2
```

4.1.6 Power: `^`

If `b` and `r` are two scalars, then `b^r` is `b` raised to the power of `r`: b^r.

If the base `b` is negative and the exponent `r` is a fraction, or either `b` or `r` is complex, the result is the principal value of the (multi-valued) power b^r. For example

```
2 ^ (5)
    32
3 ^ (-2)
    0.1111111111
```

```
9 ^ (1/2)
    3
(-3) ^ (1/2)
    0 + 3i
```

If `A` is a square matrix, and

- `n` is a positive integer, then `A^n` is the product of `n` copies of `A`:

$$A^n = AA \cdots A$$

- `n=0`, then `A^0` is the identity matrix

$$A^0 = I$$

- `n = -1`, then `A^(-1)` is the inverse matrix of `A`
- `n` is negative integer, then `A^n` is the inverse of `A^(-n)`

$$A^n = (A^n)^{-1}$$

4.1.7 Element-wise Power: .^

If `A` and `B` are two matrices of the same size, or either one is a scalar, then `A .^ B` is `A` raised to power of `B` element-wise.

For example, if want to square each entry of a vector

```
>> A = [1, 2, 3, 4, 5];
>> A .^ 2
    [1, 4, 9, 16, 25]
>> B = [-2, -1, 0, 1, 2];
>> A .^ B
    1         0.5       1         4         25
```

4.1.8 Modulus

If `p` and `q` are two numbers (of any storage type), and `p = n * q + r`, where `p` and `r` have the same sign, `n` is an integer, and `|r| < |q|`, then `p % q = r`. For example

```
>> 5 % 3
   2
>> 5 % (-3)
   2
.> (-5) % 3
   -2
>> (-5) % (-3)
   -2
```

```
>> r = 10 % pi
   0.5752220392
>> (10 - r) / pi
   3
```

If one of `p` and `q` is a matrix and the other is a scalar, or both are matrices of the same storage type, then `p % q` is defined as a matrix, whose entries are the element-wise modulus of the entries of `p` and `q`.

4.2 Unary + and -

4.2.1 Prefix + and -

If `x` is a scalar or matrix, then `+x` is the same as `x`, where `-x` is a matrix of the same size as `x`, with each element being the negative of the corresponding element of `x`. For example

```
>> x = [1, 5; -5, 1]
>> -x
     -1    -5
      5    -1
```

4.2.2 Postfix + and -

For any real scalar `a`, `a+` and `a-` have the same value as `a`, except they are used for constructing intervals.

`(a to b)` is open interval $[a, b]$.
`(a+ to b-)` is open interval (a, b).
`(a+ to b)` is half open interval $(a, b]$.
`(a to b-)` is half open interval $[a, b)$.
For example

```
>> 3+
   3

>> 3-
   3

>> 3 to 5
   [3, 5]

>> 3+ to 5-
   (3, 5)

>> 3 to 5-
   [3, 5)
```

4.3 Transpose: '

`x'` is the complex transpose of `x`. If `x` is a real scalar, then `x'` is the same as `x`. If `x` is a complex scalar, then `x'` is the complex conjugate of `x`. If `x` is an $m \times n$ matrix of numbers, then `x'` is the complex transpose of `x`, i.e., an $n \times m$ matrix whose element on row `i` and column `j` is the conjugate of the element of `x` on row `j` and column `i`.

```
>> A = [1, 2; -3, 5]
     1      2
     -3     5
>> A'
     1     -3
     2      5

>> v = [1, -5, 3, 2]
     1  -5   3   2
>> v'
      1
     -5
      3
      2
>> U = [2, 3; 2-i, 5-3i]
        2+0i                 3+0i
        2-1i                 5-3i
>> U'
        2-0i                 2+1i
        3-0i                 5+3i
```

4.4 Relational Operators

If `x` and `y` are two real scalars, then

- the value of `x == y` is 1 if `x` is equal to `y`, and 0 otherwise.
- the value of `x < y` is 1 if `x` is less than `y`, and 0 otherwise.
- the value of `x > y` is 1 if `x` is greater than `y`, and 0 otherwise.
- the value of `x <= y` is 1 if `x` is less than or equal to `y`, and 0 otherwise.
- the value of `x >= y` is 1 if `x` is greater than or equal to `y`, and 0 otherwise.
- the value of `x != y` is 1 if `x` is not equal to `y`, and 0 otherwise.

These operations are also defined if either one of `x` and `y` is a scalar and the other is a matrix, or both are matrices of the same size. In that case, when the relation of interest is true for any two corresponding elements, the value of the operation is 1. For example

```
>> 3 > -5
   1
>> -2 >= 0
   0
>> x = [1, 2, 3, 5]
>> x >= 1
   1

>> y = [2, 2, 3, 7]
>> x < y
   0
>> x <= y
   1
```

4.5 Logical Operators

Logical operators manipulate data and return a `true` or `false` value. These operators are often necessary in flow control structures where whether a condtion is true or false needs to be tested.

4.5.1 Logical Value

Virtually any data item has a true/false value. In particular, the following objects have *false* logical value

- The scalar 0
- The empty matrix or empty list `[]`
- A matrix whose entries are all zero
- The empty string `""`

Anything else would have *true* logical value.

There is no special type for logical values. The result of any relational or logical operation is either 1 or 0. But if one wish, one can always define two global variables `true` and `false`

```
global.true = 1;
global.false = 0;
```

and always use `true` and `false` for logical values.

4.5.2 Logical `and`: `&&`

`A && B` is true if both `A` and `B` are true, and false otherwise. The interpreter will evaluate `A` first. If it is false, `B` will not be evaluated. For example

```
>> x = ("abc", "def", "ghi", "jkl");
>> k = 3;
>> if (k >= 1 && k <= #x)
       play(x # k);
   end
```

4.5.3 Logical `or`: `||`

`A || B` is true if either `A` or `B` is true, and false otherwise. The interpreter will evaluate `A` first. If it is true, `B` will not be evaluated.

```
>> x = ("abc", "def", "ghi", "jkl");
>> k = 3;
>> if (k < 1 || k > #x)
       "Invalid selection"
   end
```

4.5.4 Logical `not`: `!`

`!A` is true if `A` is false, and vice versa. For example, the following piece of code finds out the count of items in a list `x` that don't `qualify` (assuming that the values of `x` have the `qualify` attribute):

```
>> count = 0;
>> for k = 1 : #x
     if ! x[k].qualify()
          ++count;
     end
   end
```

4.6 Assignment Operator: `=`

4.6.1 Lvalue: variable name

The assignment operator `=` can be used to declare and initialize a local variable, or update the value of an existing variable. The operand on the left side of the `=` sign is called the *lvalue*. The most common lvalue is an identifier. If it is an identifier that does not coincide with any local variable's name, a new local variable with this name will be created and initialized. Otherwise the existing variable by that name will be reset to the value given on the right hand side of `=`.

When assignment `a = b` is carried out, the variable `a` obtains a *copy* of `b`, and the two values are now independent. For example

```
>> b = (2, 3, 5);
>> a = b;
>> a # 2 = -3;
```

```
>> a
   (2, -3, 5)
>> b
   (2, 3, 5)
```

Two variables can never refer to the same object, or in other words, one object cannot have two different names. This is quite unlike programming languages that store *references* instead of values in variables.

4.6.2 Multiple Assignments

The lvalue can be a list of several variable names. In this case the right hand side of the assignment operator must also be a list of the same length. The effect of the assignment is the each element of the list on left hand side assigned to the corresponding variable name on the left hand side list.

```
>> data = (2, 3, "point");
>> (x, y, label) = data;
>> x
     2
>> y
     3
>> label
     point
```

4.6.3 Other Lvalues

Besides identifiers that are names of local or global variables, other expressions that can appear on the left side of = include

- Matrix indexing expression

```
A[3] = 5;
A[1:3] = [-1, 0, 1];
A[1:3, 1:3] = rand(3,3);
A[1:3; 1:3] = [3,3,3];
```

- List indexing expression

```
A # 3 = "xx xxx";
```

- Hash table entry

```
A @ "ppp" = 3;
```

- Function parameter, structure, or class member attribute

```
A.name = "flee";
```

Note that in the above

- `A` can be either the name of a variable, or one of the lvalues listed above. For example, `A.name[1] = "F"`.
- For function parameter or class member attribute, the value on the right side of `=` must be in the domain of the parameter or attribute.

4.6.4 Compound Assignments

In the expression

```
A = A + 3;
```

the variable name `A` appears twice. The expression reads "add `3` to `A`", and can be written in a condensed form

```
A += 3;
```

Note that in `A += B`, `A` must be a variable or lvalue, while `B` can be any expression. `A += B` is a valid statement as long as `A + B` is defined. For example

```
>> x = 35;
>> y = 6;
>> x += y + 1;
>> x
    42
>> A = [3, 5, 9];
>> B = [2, 0, -2];
>> A += B .^ 2;
>> A
    7   5   13
```

There are compound assignment operators for other arithmetic operators, defined in the same way. They are listed in Table (4.1).

4.7 Increment and Decrement Operators

It is often needed to add 1 to or subtract 1 from a variable. The expression for doing this is

```
x = x + 1;
y = y - 1;
```

Table 4.1: Table of Compound Assignment Operators

Operation	Equivalent
`A += B`	`A = A + B`
`A -= B`	`A = A - B`
`A *= B`	`A = A * B`
`A /= B`	`A = A / B`
`A %= B`	`A = A % B`
`A \= B`	`A = A \ B`
`A ^= B`	`A = A ^ B`
`A .*= B`	`A = A .* B`
`A ./= B`	`A = A ./ B`
`A .^= B`	`A = A .^ B`

or

```
x += 1;
y -= 1;
```

The expression `+= 1` has a further shortened form of syntax, which is `++`. It can be put in front of or after the variable, i.e., `++A`, or `A++`. The effects of `++A` and `A++` on the variable `A` are same, but the values of the two expressions are different, as described below

- `++A` and `--A`

  `++A` will add `1` to the value of `A`, and return the updated value of `A`. `--A` will subtract `1` from the value of `A`, and return the updated value of `A`.

- `A++` and `A--`

  `++A` will add `1` to the value of `A`, and return the original value of `A`. `--A` will subtract `1` from the value of `A`, and return the original value of `A`.

If `++A` and `A--` are two stand-alone expressions, their effects on the variable `A` are same and their own values are discarded, so they are equivalent. But if they are parts of a larger expression, their values will be different. For example:

```
>> n = 1;
>> p = ++n
   2
>> // now n is 2

>> q = n++;
>> // q is 2
>> // but n is now 3
>> q
```

```
    2
>> n
    3
```

4.8 Other Operators

There are quite a few other operators that are defined for particular data types. The detailed descriptions of these data types and operators are given in other chapters. The following is a summary of the usage of those.

4.8.1 Attribute Retrieval

If value `v` has an attribute `a`, then the value of the attribute can be retrieved by

```
v.a
```

The attribute name must be an identifier.

Most system defined data types have some built-in attributes. For example, matrices have attributes `length`, `norm`, `nrows`, `ncols`, etc..

4.8.2 Matrix

Matrix element and submatrix indexing operator `[]`. The syntax is

```
A[index]
```

where `A` is a matrix of any numerical type. The `index` can have a single part, or two parts separated by a comma, or two parts separated by a semicolon, or more (3+) parts separated by commas.

1. A single index. The index must be a matrix of positive integers. The value of the indexing expression is a matrix.

2. Two indices separated by a comma. This will extract a square block of the matrix, except when the first index is a slash \, where the result is one or more diagonals of the matrix.

3. Two indices separated by a semicolon.

   The value of the indexing expression is a column vector

4. Three or more indices separated by commas. This can only be used on multi-dimensional matrix.

Colon Operator for creating evenly spaced vector

If `a` and `b` are two real scalar, `a : b` returns a row vector of numbers `a`, `a+1`, `a+2`, up to `b`, or the largest number `a+n` such that `a+n <= b`. If `a<b+1`, then `a : b` returns empty matrix.

```
>> v = 1 : 5
   1  2  3  4  5
>> v = 1.2 : 5.3
   1.2  2.2  3.2  4.2  5.2
>> v = 3 : -1
   []
```

If a<b are two real numbers, and h>0, then a : h : b returns a row vector of numbers a, a+h, a+2h, up to b, or the largest number a+nh such that a+nh <= b.

If a>b, and h<0, then a : h : b returns a row vector of numbers a, a+h, a+2h, down to b, or the smallest number a+nh such that a+nh> b. a : b returns empty matrix.

```
>> v = 1 : 2 : 10
   1  3  5  7  9
>> v = 1.2 : -1 : -5
   1.2   -0.2  -1.2  -2.2  -3.2  -4.2
```

4.8.3 List Indexing

The k-th element of a list s

```
s # k
```

k must be an integer no less than 1 or a vector whose entries are no less than 1.

The length of a list is obtained by

```
#s
```

4.8.4 Function Parameter, Structure and Class Member Attribute

The value of a parameter, an attribute of a structure or a class member is referenced by

```
A.name
```

where A is an expression whose value is a function, a structure, a class, or a class member, and name is an identifier.

4.8.5 Hash Entry

The value of a hash table entry is accessed using the operator @

```
A@key
```

where A is an that evaluates to a hash table, and key is an expression whose value is a key of the table.

4.8.6 Set

To test if `x` belongs to a set `S`, we use either

```
x in S
```

or

```
x (- S
```

To test if `x` is an element of a matrix or list `S`, one can only use

```
x in S
```

To test if `S` is *subset, true subset, superset*, or *true superset* of a set `T`, use

```
S (= T
S (< T
S =) T
S >) T
```

respectively.

4.8.7 Pointer

To obtain a pointer that points to `A`, one can use

```
p = >> A
```

or

```
p =>> A
```

To retrieve the value that a pointer `p` points to, one can use

```
x = p>>
```

To reset the value pointed to by `p`, one can use

```
p>> = new_value
```

or

```
p >>= new_value
```

4.8.8 Function

To call a function `f` with argument `x`, one can use

```
f * x
```

or

```
f(x)
```

Operator	Associativity
`::`	
`>>`	
`.`	
`[], {}, ()`	
`#, @`	left to right
`'`	
`!, >>, ++, --`	
`^, .^`	
`&*, &/`	
`*, /, %, .*, ./`	left to right
`+, -`	left to right
`/`	
`>-<, <->, <-<, >->, =>`	
`to, ==, >, <, !=, >=, <=`	
`&&, in, (-, -), (=, =), (<, >)`	
`||`	
`<>`	
`->`	
`=, +=, -=, *=, .*=, %=, /=, ./=, ^=, .^=, >>`	right to left

4.9 Precedence and Associativity of Operators

Operators have a set order of precedence during evaluation. Items enclosed in parenthesis are evaluated first and have the highest precedence.

When several operators with equal precedence appear in the same statement, they are evaluated according to their associativity. For example, `+` and `-` have the same level of precedence and associate from left to right, therefore `a-b+c` is evaluated as `(a-b) + c`. The assignment operator `=` associates from right to left, therefore `a=b=c=89` is evaluted as `a = (b = (c = 89))`.

The following chart shows the order of precedence with the items at the top having highest precedence.

Chapter 5

Flow Control Structures

An *expression* consists of constants, variables, and operators. An expression followed by a return or a semicolon is a simple *statement*. A sequence of simple statements would be executed one by one by the interpreter. Sometimes we want some statements to be executed repeatedly, or we want some statements to be executed only when certain conditions are met. To achieve this we need the flow control structures which specify the order of execution of statements.

5.1 `if` statement

An `if` statement can appear anywhere a statement is expected. Its simplest form contains two key words `if` and `end`.

```
if expression
   statement
end
```

It is used to expression decisions. When the flow of program execution comes to an `if` statement, the `expression` is evaluated. If the logical value of the expression is `true`, `statement` will be executed. Otherwise, `statement` is skipped and the program moves to the point after `end`. For example

```
x = rand(2);  // get two random numbers

// test if the point (x[1], x[2]) is inside the unit circle
if norm(x) <= 1
   in_circle = 1;
end
```

If one wishes to execute certain commands when the `expression` is `true`, and some other commands when its false, a variant of the `if` statement that involves three keywords `if`, `else`, and `end` can be used.

```
if expression
   statement group 1
else
   statement group 2
end
```

For example,

```
if norm(x) <= 1
   in_circle = 1;
else
   in_circle = 0;
end
```

If the first condition is tested to be false, further condition can be tested using the `elseif` keyword to determine if the second group of commands are to be executed. The following is the general form of an `if` statement.

```
if expression 1
   statement group 1
elseif expression 2
   statement group 2
elseif expression 2
   statement group 2
...  // more elseif's
else
   statement group n
end
```

If `expression 1` is true, then `statement group 1` is executed. If `expression 1` is false and `expression 2` is true, then `statement group 2` is executed. This goes on until the condition after one of the `elseif`'s is true. If none of `expression 1` to `expression n-1` is true, `statement group n` is executed.

Note that each statement group can be one or several statements, and can also have nested control structures.

In the general form, the parts that start with `elseif` and `else` are optional.

5.2 unless statement

To execute a group of commands when a condition is not true, one can use

```
if ! expression
    command group
end
```

Alternatively, one can use the structure `unless ... end`

```
unless expression
    command group
end
```

Note that the `unless` statement only has this one simple form; there can't be any `else` or `elseif` keywords following it.

5.3 for statement

The `for` statement is used to repeatedly execute a bunch of commands for a definite number of times. The often used form is as follows

```
for var = k0 : k1
    statement group
end
```

where var is a variable name, `k0` and `k1` are two expressions that should evaluate to two integers with `k0<=k1`. When the `for` statement is encountered, the variable var is set to `k0`, and if `k0` is less than or equal to `k1`, the statement group is executed once. then the variable `var` is increased by 1. If `var` is still within `k1`, the statement group is executed again. This is repeated untill `var` exceeds `k1`. The loop counter `var` is a variable that can be accessed in the group of statements. It can be either a previously defined variable, or a new name. Inside the loop, the value of the counter is updated automatically, and therefore shouldn't be altered explicitly. Any assignment to the loop counter inside the loop is ignored. For example, the following code find the sum $1+2+3+...+n$

```
s = 0;
n = 1000000;
for k = 1 : n
    s += k;
end
```

A second variant of the `for` statement

```
for var = k0 : step : k1
    statement group
end
```

In this case, the loop counter `var` receives values of `k0, k0 + step, k0 + 2 * step, ...`, until it's greater than `k1` (if `step` is positive), or less than `k1` (if `step` is negative).

Finally, there is a short form of `for` statement, in which the value of the counter is not needed in the loop, and the code is repeated for a given number of times.

```
for n
     statement group
end
```

where `n` should be an expression that evaluates to an integer, which is the number of times the `statement group` is going to be carried out.

The statement group of the `for` loop may contain simple statements or other control structures. For example,

```
A = zeros(10, 10);
for k = 1 : 10
     for j = 1 : 10
         A[k, j] = k * j;
     end
end
```

5.4 `while` statement

The `while` statement can be used to repeatedly execute a sequence of statements as long as a condition is satisfied.

```
while expression
    statement group
end
```

What it does is evaluate `expression` first. If its logic value is true, the statement group is executed. Then `expression` is evaluated again, and if it is true `statement group` is executed again. This is repeated over and over until the value of `expression` becomes false. For example:

```
s=0;
k=1;
while  k <= 100
   s += k++;
end
```

The execution of the loop commands should be able to affect the value of the `expression`, so that the loop comes to an end at some point. Otherwise, it will run forever, and the interpreter will have to be closed by the operating system.

5.5 `until` statement

The `until` statement can be used to repeatedly run a piece of code as long as a condition is *not* satisfied.

```
until expression
    statement group
end
```

It is the opposite of `while` statement. The `statement group` is executed until the logical value of `expression` becomes true.

Note that `untill` and `until` are equivalent.

5.6 do -- while statement

The `do-while` statement repeatedly executes a piece of code until a condition is no longer satisfied.

```
do
    statement group
while expression;
```

It will execute the `statement group` first and then evaluate the logical value of `expression`. If it's false, the loop is over. If it's true, the `statement group` will be executed again, until `expression` becomes false.

A `do-while` statement always carries out the sequence statements at least once.

5.7 do -- until statement

The `do-until` statement repeatedly executes a body of commands until a condition is satisfied.

```
do
    statement group
until expression;
```

This will execute the `statement group` first and then evaluate the logical value of `expression`. If it's true, the loop is over. If it's false, the `statement group` will be executed again, until `expression` becomes true.

A `do-until` statement always carries out the group of statements at least once.

Again, `do-untill` and `do-until` are same.

5.8 break and continue

There are three types of loop structures (`for`, `while`, and `do` statements). They provide structured solutions to most of the repetitive computing tasks. However, the lack of the `goto` statement makes it not easy to stop a loop in the middle. The `break` and `continue` statements can somehow remedy this.

The `break` statement breaks out of the innermost enclosing `for`, `while`, or `do` loop and continues after the loop. For example, the following code repeatedly reads a command, and executes the command until the command is "quit", in which case the loop will be terminated.

```
while 1
   command = readline();
   if command == "quit"
        break;
   end
   execute_command(command);
end
```

The `continue` statement skips the rest of the current round of the loop and continues with the next iteration.

`continue` statement is very similar to `break` statement. The difference is it doesn't break out of the loop altogether, but only breaks out of the current round of the loop. Then the interpreter continues to execute the next iteration in the loop.

In the case of a `for` loop, the loop counter is updated immediately and its value is tested to determine if a new iteration of loop is to be carried out.

For a `while` or `until` loop, the execution flow jumps up to the beginning of the loop body. For a `do-while` or `do-until` loop, the rest of the commands of the loop body are skipped and execution flow jumps to the loop condition testing.

For example, the following code repeatedly reads and processes a command. If the command is empty, the loop starts over, if the command is "quit" the loop will be terminated. Otherwise the command is executed and new command is read again.

```
while 1
   command = readline();
   if command == ""
        continue;
   end
   if command == "quit"
        break;
   end
   execute_command(command);
end
```

Because of the use of `continue`, no empty string will be submitted for execution.

5.9 `switch` statement

The `switch` statement is a multi-way decision making mechanism which tests whether an expression matches one of a number of given values, and carry out

corresponding commands accordingly. The following is the general form of a `switch` statement.

```
switch expression
       case value1
            statement group 1
       case value2
            statement group 2
       ...
       default
            statement group
   end
```

It is equivalent to the following `if` statement

```
      if expression == value1
            statement group 1
      elseif expression == value2
            statement group 2
       ...
      else
            statement group
   end
```

The `default` clause in the `switch` statement is optional.

Besides the `case` keyword, one can also use `cases`. For example,

```
   switch word
       case value
            statement group 1
       cases values
            statement group 2
    end
```

Here, if `word` is equal to `value`, `statement group 1` will be executed. If `word` is not equal to `value`, but it is a member of the set `values`, then `statement group 2` will be executed. It is equivalent to the following `if` statement

```
      if expression == value
            statement group 1
      elseif expression in values
            statement group 2
      end
```

The following code reads lines of text and count the words in them that start with `a`, `b`, `c`, and `d`.

```
T @ "a" = 0;  // set counters to zero
T @ "b" = 0;
```

```
T @ "c" = 0;
T @ "d" = 0;
T @ "?" = 0;
while 1
  line = readline();  // read a line
  words = split(line, " "); // split the line to a list of words
  for k = 1 : (# words)     // loop over all words
    switch words # k
      cases ~/^ *[aA]/;   // word starts with a or A
          T @ "a" = T @ "a" + 1;
      cases ~/^ *[bB]/;  // word starts with b or B
          T @ "b" = T @ "b" + 1;
      cases ~/^ *[cC]/;
          T @ "c" = T @ "c" + 1;
      cases ~/^ *[dD]/;
          T @ "d" = T @ "d" + 1;
      default
          T @ "?" += 1;
    end
  end
end
```

Chapter 6

Function

Functions are one of the most important features of a programming language. Our concept of function is any value f that can be called using syntax $f(x)$. Most values such as scalars, matrices, strings, regular expressions, lists, hash tables, sets, and classes can act as functions. More useful functions are created by writing a subprogram. When such a function is called, a work space of local variables is created, and input argument values are passed to the work space, and the code of the function is executed. When the execution of the code of the function is done, the local variables of the function are cleared, but the output argument value is kept and passed back to the caller.

Shang functions are powerful and very easy to use. They also have a number of new features which are not supported in most other programming languages. For example, function can have parameters which make them customizable; their arguments can have domains, so that the interpreter can automatically check if arguments are valid; functions can be added, subtracted, multiplied, and chained to make up new 'pseudo' functions; functions can be part of a matrix; a partial list of input arguments can be passed a function to create a new function whose arguments are the ones missing from the list; etc.

6.1 Function definition

6.1.1 One Liner Functions

A function that can be represented by a single formula can be defined as a single line map using the symbol `->`. For example, function $f(x) = 2x^2 - \frac{3}{x}$ can be defined by

```
>>   f = x -> 2*x^2 - 3/x;
>>   f(3)
     17
```

Here `x` is the name of the formal argument, and will not be confused with the variable named `x` in the surrounding environment (if there is any) of the

function. The right hand side of `->` is an expression involving constants, the formal argument, and global variables. The local variables of the surrounding environment are invisible on the right hand side of `->`.

If the function has more than one argument, they can be included in a pair of parentheses. For example

```
f = (x, y) -> 2*x - y;
```

If there is no input argument, use a pair of empty parentheses.

```
>>f = () -> rand(1)^ 2; //define a function with no input

>> f()    // call it
        0.1156542049
```

One-linear functions that have no output arguments are useless and therefore not allowed.

6.1.2 Functions defined by a sequence of code

The structure of a general function is as follows

```
function input_arguments -> output_arguments
         statement
         statement
         statement
         ...
end
```

The first line of a function definition is called the *header*, which starts with keyword `function`, followed by input arguments and output arguments, which are joined by an arrow `->`. The header must end with a newline, and no other statements can appear in the line of the header after the output argument. The last line of the function must be the keyword `end`. Between the header and the last line is a sequence of statements.

In the function header, `input_arguments` and `output_arguments` are either a single variable name such as x, or a list of variable names, such as `(x, y, z)`, or an empty list. Therefore a function header may look like one of the following

```
function x -> y  // one input, one output
function (x, y) -> z  // two inputs, one output
function (x, y, z) -> w  // three inputs, one output
function x -> (y, z)     // one input, two outputs
function x -> ()     // one input, no output
function () -> z    // no input, one output
function () -> ()    // no input, no output
```

In Shang, anything is also the list containing itself. Therefor `x = (x)`, so it's ok to write `function x -> y` as `function (x) -> (y)`

Note that a variable name in the argument list may optionally contain the *default* argument value (See 6.11), and the *domain* (See 6.12) of values of the argument. Therefore, a function header may look like this

```
function (x = 0, y = 1) -> z  // input arguments have default values
function (x in _R, y _in _R) -> z  // input arguments have domains
```

For now, we will focus on the simplest case – an argument declaration only contains the argument name.

A function definition will evaluate to a value of type `function`. It doesn't automatically have a name. To be able to call the function later, usually we should assign the function definition to a variable, such as

```
>>  sumto = function n -> s
              s = 0;
              for k = 1 : n
                  s += k;
              end
          end
```

The above defines a function which takes a single argument `n` and calculates the sum of `1, 2, ..., n`. The function is assigned to the variable `sumto`. Now we can call the function by using name `sumto`

```
>> sumto(100)
     5050
```

6.2 Local variables

A function definition starts a new local variable scope in which the local variables of the scope surrounding the function definition are invisible.

Any variable that is created for the first time (by using assignment statement `variable_name = value`) in the body of a function definition is a local variable, and can only be accessed by the statements in the body of the current function definition. Local variables are even not accessible inside the body of a function definition contained in the current function body. For example

```
f = function x -> y
         var = 35;
         f = function u -> v  // this f is not the previous f
             v = u + var;  // bad, since var is not defined
             ....
             ....
         end
    end
```

The function input and output arguments automatically become local variables of the function. When the function is called, the input arguments receive the values passed by the caller, where the output variable is initialized to `null`.

The assignment operator = can be used to declare and initialize a local variable, or update the value of an existing variable. The operand on the left side of the = sign is called the *lvalue*. If the lvalue is an identifier, it must refer to a local variable. If a local variable by that name does not exist yet, a new local variable with this name will be created and initialized. Otherwise the existing variable by that name will be updated. Here we have an example function:

```
// binary search for target in a list x
binsearch = function (x, target) -> index
                n = #x;
                left = 1;
                right = #x;
                while left < right
                     // round down using floor
                     center = floor(left + (right - left) / 2);
                     if x # center < target
                          right = center;
                     elseif x # center > target
                          left = center + 1;
                     else
                          index = center;
                          return;
                     end
                end
           end
```

It has local variables `x`, `target`, `index`, `n`, `left`, `right`, `center`, among which `x` and `target` are input arguments, `index` is output argument, and the rest are variables created inside the function body.

When a function definition such as

```
f = function x -> y
        ...
        ...
    end
```

is executed, the function definition is processed, and the code is compiled to instructions in some internal format. But the local variables do not exist until a call to the function is made. When a function is called, a stack is initialized that contains all the local variables of this function. Different calls to the same function will have different variable stacks and therefore different copies of the local variables.

Upon termination of the function call, all the local variables except for the output argument will cease to exist, the memory they occupy is recycled, and the stack is destroyed. The value of the output argument will be kept and returned to the caller. If the caller chooses to ignore the return value, the Shang interpreter will claim the memory used by the return value – the programmer never needs to worry about 'garbage collection'.

6.2.1 Global Variable

As we have seen, local variables can only be accessed by the statements in the function body in which the variables are defined, and can be restrictive sometimes. For example, the following code will not work

```
taylor_sin = x -> x - x^3 / 6 + x^5 / 120;

test_fun = function x -> d
                d = abs(sin(x) - taylor_sin(x));
               // bad -- taylor_sin is invisible here
           end
```

since `taylor_sin` is a local variable in the scope outside the body of `test_fun` and therefore can not be accessed inside. One solution to this is passing the value of `taylor_sin` to the function

```
test_fun = function (x, f1, f2) -> d
                d = abs(f1(x) - f2(x));
           end
test_fun(1.25, sin, taylor_sin)
     0.0009258624152
```

However, this may make the function calls more complicated. Sometimes there are certain important data values that many functions need to use and share. It would be convenient to store such a value in a public area that can be accessed inside any function. Such a variable is called a `global` variable.

The Shang interpreter maintains a global variable that can be accessed anywhere. The name of this global variable is `global`. It is a structure whose attributes can be used as if they were independent variables. For example, to create and initialize or modify a global variable named `volume`, one can do

```
global.volume = 75
```

To reference a global variable, the keyword and the dot `global.` can be omitted if the surrounding scope does not have a local variable with the same name. For example

```
global.taylor_sin = x -> x - x^3 / 6 + x^5 / 120;

test_fun = function x -> d
                d = abs(sin(x) - taylor_sin(x));
                /* this will be ok */
           end
test_fun(1.25)
     0.0009258624152
```

Note that

- in the body of `text_fun`, `taylor_sin` refers to the global variable, since no local variable by the same name exists. Otherwise we must use `global.taylor_sin`.
- `sin` is already a global variable (i.e., an attribute of the `global` structure), which stores the built-in function `sin`.

Function definitions are usually stored in global variables. In the case of C and most other programming languages, functions are special structures and are not stored in variables, but are globally accessible. However, global variables that store other forms of data should not be used excessively. They should not be used in place of normal function argument passing. Too many global data will make the program hard to understand and problems hard to diagnosed.

A global variable can have a *domain* which limits the values that can be assigned to the variable to a set. The domain is optionally declared when the variable is first assigned a value using the keyword `in`. For example, a global variable defined as follows can only take one of the three values `0`, `1`, or `2`.

```
global.u = 1 in {0, 1, 2};
```

After this, a command `global.u = 3` will fail and cause an error, since `3` is not in the domain of `global.u`. Note that domain can only be specified once. If domain is not specified when the variable is created, it has a default domain `_ALL`, which is the set of everything.

6.3 Return value of a function

The return value is the value of the output argument when the function call is finished. So no explicit "return value" statement is needed. Since the output argument is a local variable, it is initialized to `null` in the beginning of the function call. If output variable is never reset during the function call, then `null` will be returned. Of course, usually it should be assigned a value.

If a function does not intend to return any value, then there is no need to specify the output argument variable. The header of the function should look like

```
f = function x -> ()
        ...
        ...
    end
```

In this case, the function call `f(x)` still returns the `null` value back to the caller. (the `null` value is equal to an empty list, or an empty matrix, represented the symbol `()`, or `[]`).

6.4 `return` statement

The `return` statement can be used to terminate the execution of the function immediately. For example, the following function enters a loop and repeat

some commands untill `done` is true. It then executes the `return` command and terminates the function call.

```
f = function  x -> y
           done = 0;
           ...
           while 1
                ...
                if done
                     y = some_value;
                     return;
                end
                ...
           end
    end
```

The return statement doesn't take any argument. To return a certain value, one may assign the value to the output argument prior to the return statement.

6.5 Calling a function

To call a function f with argument value x, one can write either

```
f(x)
```

or

```
f * x
```

In other words, you can view "f of x" as "multiply x by f". Conversely, whenever `A * B` is defined, one can use `A` like a function.

If `f` takes more than one input argument, one can either call the function with the correct number of arguments, or call it with a single argument, which is a list whose length is equal to the number of input arguments of the function definition. For example

```
f = (x, y, alpha) -> (x^alpha + y^alpha)^(1 / alpha);
s = (3, 4, 2);
f(s)   // equivalent to f(3, 4, 2), and f * s as well
```

On the other hand, if the function definition has only one input argument, it's still possible to call it with more than one actual arguments, in which case, the arguments will be wrapped as a list and the list is passed to the function. This provides a simple way to implement variant argument list.

```
f = function x -> s
              s = 0;
              for k = 1 : #x
```

```
                s += x # k;
            end
    end

f(2, 3, 5, 7, 11)
    28
```

In general, no matter how the function `f` is defined, the signature can be represented as `f = function x -> y`, where `x` and/or `y` can be single variable, list of variable, or empty list.

6.6 Pass Functions as Input and Output Arguments

The definition of a function can appear anywhere a data value is expected. Function is a data type and can be stored in a variable, therefore there is no trouble passing a function as input argument, or returning a function as an output argument. For example

```
>> f = g -> abs(g(0));
>> f(sin)
    0
>> f(cos)
    1
```

The one-linear definition of a function can be used directly

```
>> f = g -> abs(g(0));
>> f(x -> sqrt(3 + x^2))
    1.732050808
```

```
>> select_func = function code -> f
                  if code == 's'
                       f = sin;
                  elseif code == 'c'
                       f = cos;
                  elseif code == 'a'
                       f = sin + cos
                  else
                       f = x -> sqrt(1 + x^2);
                  end
              end
>> f = select_func('a');
>> f(pi/4)
     1.414213562
```

Also, it is very common and easy to use nested functions — function definitions inside function definition. For example

```
create_func = function d -> func
                  ...

                  func = function x -> y
                            y = sqrt(x^2 + 1);
                        end
              end
```

When `create_func` is called, it creates a function `f` and return it. Note that inside the definition of `func`, the value of input argument `d` is not accessible, so it may seem impossible to create a function based on the input value of `create_func`. Actually there are two ways to get around this – parameterized functions (See 6.8) and partial substitution (See 6.10).

6.7 Argument Passing

All input arguments are passed by value. Therefore assigning values to the input arguments inside the function definition will not alter the original values of the arguments of the caller.

```
>>  f = function x -> y
               x = x^2 + 1;
               y = sqrt(x);
        end
>>  p = 25;
>>  f(p)
      25.01999201
>>  p
      25
```

If it is desired to change the value of a local variable using a function, one can either assign the result of the function call to the variable, or pass a pointer (See 8.2) to the function.

6.8 Return Multiple Output Arguments

If the output argument in the function header contains a list of names, then each of these name will become a local variable for the function. At the end of the function call, a list will be built using these output variables and returned to the caller. For example

```
>> summary = function x -> (mean, min, Q1, median, Q3, max)
            x = sort(x);
            n = length(x);
            s = x[1];
            min = x[1];
```

```
            max = x[1];
            for k = 2 : n
                 if x[k] < min
                      min = x[k];
                 end
                 if x[k] > max
                      max = x[k];
                 end
                 s += x[k];
            end
            mean = s / n;

            global.findmedian = function y -> md
                 N = length(y)
                 if N % 2
                      md = y[(N + 1) / 2];
                 else
                      md = (y[N / 2] + y[N / 2 + 1]) / 2;
                 end
            end

            median = findmedian(x)

            if n % 2
                 Q1 = findmedian(x[1 : (n + 1) / 2]);
                 Q3 = findmedian(x[(n + 1) / 2 : n]);
            else
                 Q1 = findmedian(x[1 : n / 2]);
                 Q3 = findmedian(x[n / 2 + 1 : n]);
            end

       end
>> s = summary([3, 5, 2, 1, 9, 10, 22])
     (7.42857, 1, 2.5, 5, 9.5, 22)
```

The function returns a single value, which is list of all output arguments. We can use a multi-assignment statement to pass all the list assignments to individual variables of the caller

```
(mean, min, Q1, median, Q3, max) = summary([3, 5, 2, 1, 9, 10, 22]);
```

6.9 Function Parameters

Sometimes the terms `function argument` and `function parameter` are used interchangeably. In Shang language, they are different. Apart from input and

output arguments, a function may have a number of *parameters*. These are attributes of the function that can be used to modify the behavior of the function after the function is created. A parameterized function can be viewed as either a family of functions, from which one can pick given the parameter value, or a function with a state that can be modified.

6.9.1 Parameter Syntax

To specify a list of function parameters, one can include them in a pair of square brackets and place the brackets right in front of the input arguments list in the function header. The items in the parameter list are separated by commas. The syntax of the simplest form of a parameter item is

```
parameter_name = intial_value
```

where `parameter_name` is a name used to identify the parameter, `initial_value` is the initial value of the parameter

For example, the following defines a function $f(x) = \alpha x + \beta$, with two parameter α and β.

```
f = function [alpha = 3, beta = 5] x -> y
      y = alpha * x + beta;
    end
```

A parameterized function behaves like a class member that has attributes that can be accessed using the dot operator. To refer to the value of a parameter `alpha` of function `f` outside the function body (in a scope where `f` is visible), one can use `f.alpha`. For example;

```
>> f = function [alpha = 3, beta = 5] x -> y
       y = alpha * x + beta;  // alpha is the value of the parameter
     end
>> f.alpha
   3
>> f.beta
   5
```

To refer to the value of parameter `alpha` inside the function definition body, one can simply state the parameter name, like in the above example. However, if a local variable has the same name, the local variable has higher precedence. In this case, `alpha` refers to the name of the local variable, while to access the parameter named `alpha`, one needs to use `this.alpha`. Of course, `this.alpha` always refers to a function parameter, whether there is a local variable `alpha` or not. So the above function can also be written as

```
f = function [alpha = 3, beta = 5] x -> y
      y = this.alpha * x + this.beta;
    end
```

Inside a function body, the keyword `this` refers to the function itself. The following example illustrates the difference between a local variable and a parameter.

```
>>   f = function [alpha = 3, beta = 9] () -> y
            alpha = -5;   /* now alpha is a local variable */
            y = [alpha, this.alpha, beta, this.beta];
     end
>>   f()
       -5    3    9    9
```

6.9.2 `public` parameter

If the parameter is specified with no modifier, or with the keyword `public`, it is a public parameter. A `public` parameter is like a readonly local variable for the function, which can be accessed but not modified by the function itself. But its value can be extracted and reset in the scope where the function is visible. For example

```
>> f = function [public alpha = 3, public beta = 5] x -> y
         y = alpha * x + beta * y;
   end

>> f.alpha
   3
>> f.alpha = -5;
>> f.alpha
   -5
>> f.beta = 10;
>> f.beta
   10
```

It's impossible to change the value of a `public` parameter inside the function body. For example, if we do

```
>> f = function [public alpha = 3, public beta = 5] x -> y
            alpha = 10;
            ...
       end
>> f.alpha
        3
```

we merely creates a local variable whose name is `alpha` and value is 10. And

```
>> f = function [public alpha = 3, public beta = 5] x -> y
            this.alpha = 10;
            ...
       end
```

will result in a compile error. The purpose of this restriction is to reduce the implicit behavior of a function. The owner (the surrounding scope) of a function can always predict the behavior of a function as long as it only has public parameters because the function itself cannot change its public parameter values.

6.9.3 `private` parameter

If a parameter is declared with keyword `private` in front of the parameter name, it is a private parameter. A private parameter is accessible but not modifiable outside the function definition body, but can be reset within the function during function execution. For example, in the following function, the `private` parameter `count` keeps a record of the times `f` is called.

```
>> f = function [private count = 0] x -> y
            this.count = this.count + 1;
            ...
   end
>> f.count
   0
>> f(1);
>> f.count
   1
>> f(2);
>> f.count
   3
>> f.count = 0; // !!! error ...
```

Inside the function body, to access a private parameter, `this` keyword must be used. Internal parameters will add to the implicit behavior of a function, and makes the function call results hard to predict, therefore they should be used only when necessary.

6.9.4 `common`, `auto`, and `readonly` parameters

Function parameters can be also declared with keywords `common`, `auto`, or `readonly` in front of the parameter identifier.

A `readonly` parameter is an attribute of the function that is accessible but not modifiable inside or outside the function. However, its value can be modified by other `common` or `auto` parameters.

`common` and `auto` parameters are both functions

A `common` parameter is an attribute of the function that is itself a function. Its value is fixed (to the initial value, which is a function). It is like a utility function which can be used to reset the values of `readonly` parameters.

An `auto` parameter is an attribute of the function that is itself a function which takes no input arguments. When the auto parameter is accessed, the function will be called automatically.

The following example will illustrate the uses of various parameters.

```
polynomial = function [private coeff,
       common  setcoeff = function c -> ()
                             parent.coeff = c;
                          end,
       auto roots = () -> polyroots(parent.coeff)] x -> y

       y = polyeval(coeff, x);
end
```

6.9.5 Parameter Domain

Reseting a parameter's value will alter the behavior of a function, therefore it is necessary that a parameter is only allowed to be set to certain values. This can be realized by specifying a *domain* for the parameter. For example

```
>> plot = function [coordinate = "cartesian" in {"cartesian", "polar"},
                   color_scheme = "bw" in {"bw", "rgb"}] (x, y) -> ()
                  ...
                  ...
       end
>> plot.color_scheme = "rgb"; // ok
>> plot.color_scheme = "bs";  // !!! error
```

The general syntax for specifying the domain of a parameter is

```
[access_control_type] parameter_name [= initial_value [in domain]]
```

where `initial_value` is the initial value of the parameter, `domain` is the domain of the parameter. `initial_value` and `domain` are evaluated in the scope surrounding the function definition. If the domain is specified, then the value assigned to the parameter must belong to the domain. Otherwise an error will occur.

The value of `domain` may be a finite set, which is defined using a pair braces, like in the above sample function `plot`. Domains can also be functions that act like sets. For example, we may use the following function to define a set of positive numbers

```
pnumbers = x -> (x > 0);
```

Then we may use this function as the domain of a parameter

```
lf = function [alpha = 1 in pnumbers, beta = 0 in pnumbers] x -> y
          y = alpha * x + beta;
     end
```

Note that the set `pnumbers` can be represented using interval `pnumbers = 0+ to inf`, or using the built-in function `_R`.

By using domains we ensure that parameters can only be set to valid values, and then the function is never in an illegal state. Note that although you don't

have to provide a domain in the function definition, every parameter always has a domain – the default domain for any parameter is the set that contains everything – `_ALL`. Since anything belongs to this set, having it as domain won't apply any restrictions to the values assigned to the parameter.

Spawn a New Function

A function is a value and can be stored in a variable. If `f` is a function,

```
g = f;
```

will create a copy of `f` and assign it to `g`. Any modification to `g` will not affect `f`. Only functions that have parameters are modifiable. The following code makes a copy of `f` and then modifies the copy.

```
f = function [alpha = 1] x -> y
        y = sqrt(x^2 + alpha^2);
    end
g = f;
g.alpha = 5;
```

Now we have two functions

$$f(x) = \sqrt{x^2 + 1}, \quad g(x) = \sqrt{x^2 + 25}$$

Any function can act as a prototype and new functions can be "spawned" directly from it. The syntax of spawning a new function is

```
f[parameter_value_1[, parameter_value_2, ...]]
```

where `parameter_value_1` is the value of the first public parameter, `parameter_value_2` the value of the second public parameter, etc. The parameter names must appear in the same order as in the original function definition, with non-public parameters skipped. For example

```
f = function [alpha = 1] x -> y
        y = sqrt(x^2 + alpha^2);
    end
g = f[5];
```

`g` will be a function that is the same as `f`, except that its parameter `alpha` has value 5.

It's possible to combine function spawning and calling and pass the values of arguments and parameters at the same time. For example

```
>> f = function [alpha = 1] x -> y
        y = sqrt(x^2 + alpha^2);
   end
>> f[5](3)
   5.830951895
```

6.10 Recursion

A function may call itself directly or indirectly. This is referred to as recursive function call, and is very common feature of most programming languages. In Shang there are three ways that recursion can be done. First, the keyword `this` can be used inside the function to call the function object itself. For example

```
fac = function n -> f
            if n <= 1
                 f = 1;
            else
                 f = n * this(n - 1);
            end
      end
```

Inside the function body, `this` refers the function `fac`, therefore `this(n - 1)` is a call to the function itself. The result of calling `fac(n)` will be the factorial of `n`, i.e., the product of `1`, `2`, ..., up to `n`. This appears to be a neat solution for doing recursion, but when calling a function itself indirectly is needed, it can't help. In this case the best strategy is to define the functions as global variables. The following is an example of indirect recursion.

```
global.f = function x -> y
              ...
              global.g(3);
              ...
           end

global.g = function x -> y
              ...
              global.f(3);
              ...
           end
```

Here the two functions `f` and `g` call each other. We call this indirect recursive function call. Note that direct recursion can be done like this as well.

```
global.fac = function n -> f
            if n <= 1
                 f = 1;
            else
                 f = n * fac(n - 1);
            end
      end
```

Note that the statement `f = n * fac(n - 1)` can be written as `f = n * global.fac(n - 1)` as well.

Finally, it's also possible to implement recursion by passing the function to itself as an input argument value (or set it to one of its parameter values). For example:

```
>> fib = function (n, f) -> s
                if n <= 1
                     s = 1;
                else
                     s = f(n - 1, f) + f(n - 2, f);
                end
   end
```

which can be called as follows

```
>> fib(10, fib)
   89
```

.

6.11 Partial Substitution

If we have a function $f(x, y) = e^{-(x^2+y^2)}\sqrt{x^2 + y^2}$

```
f = (x, y) -> exp(-(x^2 + y^2)) * sqrt(x^2 + y^2);
```

we don't have to pass the values of both arguments to `f` all at the same time. Instead, we may pass the value of `y` to `f`, and leave `x` unspecified. The result is a new function of `x`. An unspecified argument value is represented by a dot.

```
f1 = f(., 3)
```

Here the definition of function `f1` is

$$f1(x) = e^{-(x^2+9)}\sqrt{x^2 + 9}$$

We call such a function call a *partial substitution*.

In general, when calling a function, some input arguments can receive dots as values, while the rest receive actual values. The result of such a function call is a new function whose input arguments are those given dots, while the arguments of the original function which do get values passed to are no longer arguments of the new function. Here a dot means a missing or delayed value, and corresponds to an argument of the new function. Calling the new function is equivalent to calling the original function by two separate steps of argument passing. For example:

```
>> f = function (x, y, alpha) -> s
           s = (x^alpha + y^alpha) ^ (1 / alpha);
    end
```

```
>> f2 = f(., ., 2);
>> f2(3, 4)    // equivalent to f(3, 4, 2)
   5
```

A function call in which all argument values are dots will have no effect – it just evaluates to the original function. For example, `sin(.)` is the same as `sin`.

With the support of partial substitution, we can write a general function, which takes many arguments. Then for a particular application, all calls to the master function may use the same values for some arguments. Then we may create a customized version of the function by partially substituting these often used argument values into the function. The calling sequence of the customized function is more compact. For example, a plotting function may be designed to take all the following arguments

```
plot = function (x, y, coord, line_style, color,
                              axis, xlabel, ylabel) -> graph
                 .....
                 .....
       end
```

If every time we use the same choice of coordinates, line style, color setting, axis, xlabel, and ylabel, then we can do

```
myplot = plot(., ., "cartesian", "solid", "bw", "boxed",
                                  "population", "revenue");
```

to make a plotting function that takes only `x` and `y` as arguments.

Partial substitution provides an alternative way to function parameters for customizing the behavior of a function after it is created. Unlike parameterized functions in which some variable are predefined to be parameters, in partial substitutions any of the input variables can be chosen as parameters. However, in terms of performance it may be at disadvantage compared to the function parameter approach since one more abstract level is added and two argument passing steps have to be processed in order to perform a call to a partial substitution.

6.12 Default value of arguments

When defining a function the default value of an input argument can be specified using the assignment operator `=`. When calling a function, the symbol `*` should be given to each argument for which default argument value is intended. For example:

```
>> f = function (x = 3, y = 5, z = 7) -> w
       // define function w = f(x,y,z), with
       // default argument values 3, 5, 7
         ...
   end
>> f(*, 10, *)   // equivalent to f(3, 10, 7)
```

Here the default values of the 1st and 3rd arguments are passed to the function.

Function calls don't have to match the signature of the function definition exactly. To be specific, there are the following situations.

- If the function call provides fewer arguments than the function definition requires, the provided arguments will match from the begining of the formal argument list, and the missing arguments are given the default values. For example, if `f` is called by `f(3)`, it is equivalent to `f(3, *, *)`.
- If the function definition requires only one argument, and a function call receives more than one arguments, then these arguments are wrapped as a list and passed to the function.
- If the function definition requires more than one arguments, and a function call receives only one arguments, and this argument is a list of the same length as the formal argument list of the function definition, then entries of the provided argument list will be extracted and passed to the function.

Another fact is that every function argument has a *default* `default` value. The *default* `default` value of an input argument is `null` (which is the same as the empty matrix `[]`). Of course, unless the function definition code does something to handle *default* `default` value, using it directly will usually cause an error.

6.13 Domain of Function Argument

The definition of a function may specify the domains of input or output arguments. Domains are sets to which the values of arguments must belong. When a function is called, the interpreter first evaluate all the arguments passed, and then check if each is in the corresponding domain. If the domain check is passed, then the argument values as well as the program control are passed to the function. Otherwise a *domain error* is raised and the function will not be executed. When the function call is finished, the interpreter also checks if the output variable value is in the domain of the output argument. If not, the value will not be returned, and a *domain error* is caused.

The domain of an argument is specified in the function header using the following syntax

```
variable_name = default_value in domain
```

For example, the function header may look like

```
global._RP = (0+ to inf);
func = function (x = 1 in _RP,
           options = "real" in {"real", "complex") -> y in _RP
    ...
end
```

The above defines a function that takes two arguments `x` and `options`. `_RP` is the set of all positive numbers. So the argument `x` must be a positive number, while `options` must be either `"real"` or `"complex"`. The output argument value `y` must be a positive number as well. Whenever `func` is called with invalid argument values, the interpreter will complain.

```
>> func(-5, 6) // !! domain error ...
```

When function definition is processed, the default value and the domain of an argument are evaluated in the scope surrounding the function definition. Therefore they can be expressions containing the local variables of the surrounding scope, but they cannot make an reference to the other argument names of the function definition. For example

```
x = 3;
p = function (x = x) -> (y = x)
        ...
    end
```

Here in `(x = x) -> (y = x)`, the first `x` is the name of the formal argument, while the second and third `x` are the default values of arguments and refer to the value of the local variable `x` (whose value is 3) of the surrounding scope.

We may define a set, and then use it as the domain in a function definition immediately

```
Dmx = x -> (x <= -1 || x >= 2);
Dmy = x -> (x >= -1 && x <= 2);

f = function (x = 3 in Dmx, y = 0 in Dmy) -> z in _RP
           ...
    end
```

In practice sometimes the domains of two arguments are not independent of each other. For example

$$f(x,y) = \frac{1-x-y}{x^2+y^2}, \qquad 0 \le x \le 1,\ 0 \le y \le x$$

To fully represent this domain in Shang, we need to define a function of one variable, which is either a vector, or a list of `x` and `y`.

```
Dxy = v -> (v[0] >= 0 && v[0] <= 1 && v[1] >= 0 && v[1] <= v[0]);

f = function v = [0, 0] in Dxy -> w
           w = (1 - v[0] - v[1]) / (v[0]^2 + v[1]^2);
    end
```

Note that if the domain of an input argument is given, a default value must be provided. Otherwise the interpreter may find the *default* `default` value (`=`

`null`) does not belong to the domain, and have difficulty processing a call to the function that provides default argument value. On the other hand, one may specify the default value but omit the domain. Whenever the domain is not specified, a default domain, which is `_ALL`, the set of everything, will be set as the domain of the argument.

Programming languages can roughly be divided into two categories: statically typed, in which argument value must be of a certain type, and dynamically typed, in which formal arguments can be any type. Shang has both the flexibility of dynamically typed language and the rigorousness of statically typed language. It is in fact superior to statically typed language with respect to type-checking since the domain is a *set*, which can be much more expressive and specific than a *type*. Many times, being a value of certain type doesn't guarantee that the argument is a valid one. For example, if an argument `a` represent age. Statical type checking only guarantees that `a` is an integer, while in Shang, the domain of `a` can be a customized set that contain all valid ages, such as integers between 0 and 150.

6.14 Calling functions using named arguments

To pass argument values to a function, an alternative way is to use a pair of braces, and "name" the arguments. For example, instead of `f(3,9)`, one can use

```
f{x=3, y=9}
```

Here `x` and `y` must be the same formal argument names declared in the definition of the function.

Calling functions this way has two advantages. Firstly, the order of the arguments doesn't matter. For example `f{x=3, y=9}` and `f{y=9, x=3}` would be the same. Secondly, you don't have to pass a value for every argument. Any arguments not named in the braces will take default values (specified in the function definition).

Note that an expression `{x=3, y=9}` is exactly the same as the syntax for defining a `structure` with fields `x=3, y=9`. So passing arguments as named parameters is similar to passing a structure to the function.

6.15 Built-in functions

A built-in function is a function provided by the interpreter system. It is compiled from C code and can be called the same way as a user defined function. For example, `rand` is a built-in function that that generates random numbers and matrices, `length` is a built-in function that returns the length of a vector or matrix, and `cos` is the trigonometric function cosine.

```
>> x = rand(5, 1);
>> length(x)
```

```
     5
>> cos(pi/4)
     0.7071067812
```

The complete list of the built-in functions is given in the appendix.

Like user-defined functions, built-in functions can have parameters as well. For example, the built-in function `normal` which generates (pseudo) random numbers of normal distribution has five parameters `mu`, `sigma`, `pdf`, `prob`, and `quantile`. `mu` and `sigma` are the mean and standard deviation of the distribution.

```
>> normal(1)
    0.5723149278
>> normal(2)
    1.64
    -0.559
>> normal.mu
    0
>> normal.sigma
    1
>> normal.prob(-inf, 0)
    0.5
```

The two parameters `mu` and `sigma` are modifiable. Their initial value are `mu = 0` and `sigma = 1` respectively. If we want a non-standard normal distribution random number generator, we may make a copy of `normal` and reset the values of `mu` and `sigma`

```
>> my_normal = normal;
>> my_normal.mu = 10;
>> my_normal.sigma = 2;
>> my_normal(1)
      8.029350953
>> my_normal(2)
      12
      10.9
>> my_normal.prob(-inf, 10)
      0.5
```

Note that `normal` is a global variable and can only be modified using the `global` keyword, therefore `normal.mu = 10` doesn't work. But in `my_normal = normal`, `normal` does refer to the global variable `normal`, since here we are not trying to alter its value. The assignment `my_normal = normal` assigns a copy of `global.normal` to `my_normal`, which can then be modified.

It's very easy to override the definition of a built-in function. Locally defined variables have higher precedence. Once they are defined, they will replace the built-in definitions.

```
>> sin = x -> x - x^3 / 6;
>> sin(6)
     -30
```

However, an overridden built-in function is not lost. Actually all built-in functions are attributes of the global variable `builtin`. So they can always be accessed using `builtin.name`. For example

```
>> sin = x -> x - x^3 / 6;
>> sin(2.5*pi)
    -72.891530550
>> builtin.sin(2.5*pi)
    1
```

6.16 Pseudo Functions

Pseudo functions are structures that act like functions that are neither built-in nor generated from user written function code. They are created from existing functions or other data based on certain rules.

6.16.1 Operations on functions

It's possible to add or multiply two functions, subtract one from another, divide one by another, and compose (chain) two functions (user defined, built-in, or defined in any other way). The result is called a pseudo function, and can be used anywhere a function is expected.

The operators for function addition and subtraction are `+` and `-`, while the operators for function multiplication and division are `&*` and `&/` respectively. The operator for function composition is `<>`. For `f + g`, `f - g`, `f &* g`, and `f &/ g`, `f` and `g` should have the same signature, i.e., they take the same number of input arguments, and return two values that are compatible for addition, subtraction, multiplication, and division respectively.

Operator	Definition
`h = f + g`	`f(x) = f(x) + g(x)`
`h = f - g`	`f(x) = f(x) - g(x)`
`h = f &* g`	`f(x) = f(x) * g(x)`
`h = f &/ g`	`f(x) = f(x) / g(x)`
`h = f <> g`	`f(x) = f(g(x))`
`h = -f`	`f(x) = -f(x)`

For `f <> g`, the number of output arguments of `g` should be the same as the number of input arguments of `f`.

For example

```
>> f = sin + cos;
>> f(pi/4)
    1.414213562

>> p = sin &* cos;
>> p(pi/4)
    0.5

>> g = sin - cos;
>> g(pi/4)
    -1.110223025e-16

>> h = log <> sin;
>> h(pi/2)
    0
```

Note that in the above, `f` and `g` don't have to be both 'real' functions – they can be pseudo functions, or anything that can be used as functions, such as numbers or matrices. If either operand is a function, the operation will create a new pseudo function. For example

```
f = sin + 2
```

will create a function `f(x) = sin(x) + 2x`. Note that `2` as a function means `x -> 2x` instead of `x -> 2`.

6.16.2 Function Matrix

Function vectors or matrices are vectors or matrices whose entries are functions instead of numbers. When they are called, the argument is passed to each element function, the outcome is a vector or matrix built from the return values of each element function call. For example

```
f = [cos, -sin; sin, cos];
```

defines a function

$$f(x) = \begin{bmatrix} \cos x & -\sin x \\ \sin x & \cos x \end{bmatrix}$$

```
>> f = [cos, -sin; sin, cos];
>> f(pi/4)
    0.707    -0.707
    0.707     0.707
```

Note that the elements of the matrix can be data of other type that can be used as functions, and are not restricted to functions (user or built-in). As long as one element is a function, the whole matrix is evaluated to a function matrix. The following example creates a function matrix

```
f = [exp <> cos, 2; x -> sqrt(1 + x^2), exp <> (-sin)]
```

which represents the following function

$$f(x) = \begin{bmatrix} e^{\cos x} & 2x \\ \sqrt{1+x^2} & e^{-\sin x} \end{bmatrix}$$

Note that `<>` is the symbol of function composition.

In general, all the elements of a matrix function should have the same number of input arguments, and the return values should be compatible for building a matrix.

6.16.3 Everything is a function

In Shang, *function* is a *facet* of almost any data type, therefore almost anything can be used as a function. In particular,

- If `A` is a number or matrix, then `A(x) = A * x`.
- If `A` is a hash table and `x` is a key, then `A(x) = A @ x`.
- If `A` is a set, then `A(x) = 1` if `x` is in `A`, and `0` if `x` is not in `A`.
- If `A` and `B` are strings, then `A(B)` is non-zero if `A` is a substring of `B`, and zero otherwise.
- If `A`, and `x` are two lists with the same length, then

$$\texttt{A(x)} = \sum_{\texttt{k=1}}^{\texttt{\#A}} [\texttt{A\#k}][\texttt{x\#k}]$$

- If `A` is a regular expression and `B` is a string, then `A(B)` is non-zero if `B` matches `A` and zero otherwise.
- If `A` is a class, then `A(x)` is non-zero if `x` is a member of `A`, and zero otherwise.
- If `A` is an member of a class that has multiplication operator overloaded, `A(x) = A * x`.

For example, the function $f(x, y) = 5x - 3y$ can be created by `f = (5, -3)`, while the function

$$f(x) = \begin{cases} 0, & x = 1 \\ 1, & x = 2 \\ 2, & x = 3 \\ 3, & x = 4 \\ 4, & x = 5 \\ 5, & x = 0 \end{cases}$$

can be created by

```
f = {0 => 1, 1 => 2, 2 => 3, 3 => 4, 4 => 5, 5 => 0};
```

6.16.4 Turn a matrix into a function

A matrix (or a number) is already a linear function via the multiplication operation `A(x) = Ax`. Yet, it's also possible to turn the matrix indexing expressions into functions of the index.

- If `A` is a vector (a $1 \times n$ or $n \times 1$ matrix), then `f = A[.]` creates a function of one variable such that `f(x) = A[x]`.
- If `A` is matrix, then `f = A[., .]` creates a function of two variables such that `f(x, y) = A[x, y]`.
- If `A` is matrix, then `f = A[., :]` creates a function of one variable such that `f(x) = A[x, :]`, which gives the `x-th` row of `A`.
- If `A` is matrix, then `f = A[:, .]` creates a function of one variable such that `f(x) = A[:, .]`, which gives the `x-th` column of `A`.
- If `A` is matrix, and `k` is a constant (a scalar or vector that is a valid column index for `A`), then `f = A[., k]` creates a function of one variable such that `f(x) = A[x, k]`.
- If `A` is matrix, and `k` is a constant (a scalar or vector that is a valid row index for `A`), then `f = A[k, .]` creates a function of one variable such that `f(x) = A[k, x]`.

For example

```
>> s = floor(rand(5) * 10)
    5
    3
    0
    7
    2
>> fs = s[.];
>> fs(1)
    5
>> fs(3)
    0
```

Another example

```
>> s = floor(rand(5, 5) * 10)
    7         0         5         8         3
    9         0         3         7         8
    1         6         6         7         9
    2         5         5         3         2
    1         2         5         5         0

>> fr = s[., :];
```

```
>> fr(1)
   7         0         5         8         3
>> fr(3)
   1         6         6         7         9
>> fc = s[:, .];
>> fc(1)
     7
     9
     1
     2
     1
>> fc(3)
     5
     3
     6
     5
     5
```

Sometimes a structured set of data is stored most efficiently in a vector or matrix. However a client function that needs to use the data may expect a function that returns one data item given an index. In such a situation we can store the data in a matrix `A`, and pass `A[.]` whenever the client function is being called.

Chapter 7

Class and Member

A class is a set whose members are data values 'created' by the set itself. Each member of a class is a structural value whose attributes are prescribed in the class definition. A class definition includes a constructor, which is a function used to create new class members, and specifications of other member attributes. When a new class member is created by calling the constructor, the member will possess the member attributes. Member attributes can be any data type including functions. An attribute function may access or modify the other attributes of the member.

7.1 Class definition syntax

A class definition starts with the keyword `class`, and ends with the keyword `end`, with a number of attribute declarations in between.

```
class
     attribute_1;
     attribute_2;
     attribute_3;
     ....
     attribute_N;
end
```

There are two types of attributes, those for the members of the class and those for the class itself. They are called *member attributes* and *collective attributes* respectively.

To define a member attribute, one needs to specify the type of access control

```
access_control_type identifier [= default_value [in domain]]
```

where the specification of `default_value` and `domain` is optional. To define a collective attribute, one starts directly with the name of the attribute (which currently can only be one of `new`, `super`, or `title`).

7.2 Access Control of Member attributes

There are four types of access controls for member attributes, which are `public`, `private`, `common`, and `auto`.

7.2.1 `public`

A `public` attribute is accessible to and modifiable by both attribute functions of the same member and the surrounding scope of the member. For example

```
circle = class
           public radius = 1;
           auto   perimeter = () -> 2 * pi * parent.radius;
           auto   area = () -> pi * (parent.radius)^2;
         end
```

in which `radius` is a `public` attribute, and `perimeter` and `area` are two `auto` attributes. If `x` is a `circle`, then one can use `x.radius` to find its radius, and use `x.radius = new_value` to reset its radius to a new value. The new value assigned to a `public` attribute must belong to the *domain* of the attribute, which, by default, is _ALL.

Inside an attribute function, to access another attribute of the same member of the class, the keyword `parent` must be used. In the above example, if `x` is a member of `circle` class, then in the surrounding scope of `x`, the radius of `x` is `x.radius`, while inside an attribute function (such as `perimeter` and `area`) of `x` the radius of `x` is referred to as `parent.radius`.

7.2.2 `private`

A `private` member attribute is not visible outside the definition of the class, but can be accessed and modified by other attribute functions of the class member. For example

```
>>   circle = class
              private radius = 1;
              common  getRadius = () -> parent.radius;
              // access parent.radius is ok here
              common  setRadius = function x  ->  ()
                                      if x > 0
                                           parent.radius = x;
                                      end
                                  end
              auto    perimeter = () -> 2 * pi * parent.radius;
              auto    area = () -> pi * (parent.radius)^2;
            end
>>   p = circle.new();
>>   p.radius;  // Error: accessing private attribute
```

```
>>   p.getRadius()
       1
>>   p.setRadius(12);
>>   p.getRadius();
       12
```

Here `radius` is private attribute and cannot be accessed outside the class. But other attribute functions (`getRadius` and `setRadius`) can access it using the `parent` keyword.

7.2.3 common

A `common` attribute must be a function, and can not be modified, and its code is therefore shared by all members of the same class. When declaring a `common` attribute, the default value (a function definition) must be provided, and will become the attribute value of all members of the class.

In the previous example, the two functions `getRadius` and `setRadius` are common attributes. Common attributes as functions can be called outside or inside the class, but they cannot be modified. For example

```
>>  circle = class
      ...
      common  setRadius = function x  ->  ()
        if x > 0
            parent.radius = x;
         end
      end
      ...
    end
>>  p = circle.new();
>>  p.setRadius = function x -> ()
        ...
    end

    Error: modifying common attribute
```

If `setRadius` is declared as `public`, then we can still use it almost the same way, but we'll be free to assign new values to the `setRadius` attribute of any individual member of `circle`.

7.2.4 auto

An `auto` attribute is essentially a special `common` attribute. It is a read-only function, therefore the values of the attribute for all members of the class are the same as the default value given in the class definition, which is a function, and cannot be modified after the class member is instantiated. But an `auto` function must take no input argument, and therefore the header looks like

```
auto f = function () -> y
                    ...
                    ...
           end
```

Normally if `f` has no input argument, it should be called by appending a pair of empty parentheses, as in `f()`. Here suppose that `u` is class member. If `f` were a common attribute, we would have to call it by `u.f()`. The only difference made by being an `auto` is now `f` can be called by `u.f` without the empty parentheses enclosure. For example

```
circle = class
           public radius = 1;
           auto   perimeter = () -> 2 * pi * parent.radius;
           auto   area = () -> pi * (parent.radius)^2;
         end
```

If `x` belongs to the `circle` class, to find its area, one only needs to write `x.area`. But manually setting `x.area` to new value using `x.area = new_value` is not allowed, since `x.area` can only be automatically calculated using the attribute function.

7.3 Domain of Attribute

The attribute `radius` of class `circle` obviously should be a positive number. Does being `public` mean that any absurd value can be assigned to it, and the `circle` class member is still legitimate? Actually, an attribute definition may also include a *domain*, which is the set of all allowed values of the attribute. The Shang interpreter monitors every value assigned to an attribute, attempt to set an attribute to an invalid value will be stopped.

The syntax for specifying the domain of an attribute is the keyword `in` followed by a set.

```
global._RP = (0+ to inf);
circle = class
           public radius = 1 in _RP;
           auto   perimeter = () -> 2 * pi * parent.radius;
           auto   area = () -> pi * (parent.radius)^2;
         end
```

Here `_RP` is a global variable predefined to be the interval of all positive scalars. With this domain specified, now only positive numbers can be assigned to the `radius` field.

When an attribute is declared without specifying a domain, its domain will be the default value `_ALL`, which is the set of everything. Note that whenever a domain is specified, the default value of the attribute must also be specified, since the default constructor of the class will create a member of the class that

has only default values. If no default value is given, then `null` is used. But then if there is a domain defined, `null` may not be in the domain, and the constructor may return an invalid object.

Domain is most useful for `public` attributes. In the following example, each attribute of a `person` is `public`, and therefore can be accessed and modified directly. Yet, only valid values can be assigned to the attributes. In many languages, to protect the integrity of data, such attributes must be made `private`, and direct access to them disallowed. Instead, function attributes called "getters" and "setters" are used to access and modify their values (see the example in 7.2.1).

```
person = class
              public gender = "M" in {"F", "M"};
              public age = 1 in 1 : 150;
              public first = "Mark" in ~/[A-Za-z][A-Za-z]*/;
              public last = "Brown" in ~/[A-Za-z][A-Za-z]*/;
         end
```

Although domain makes most sense in the case of `public` field, a `private` field can have a domain as well.

7.4 Collective attribute

A record in the class definition that begins with no access type is a *collective attribute*. It is not created for any particular class member, but belongs to the class itself. Currently only three collective attributes can be defined, which are

- `title`: must be a character string. It is meant to give a brief description of the purpose or functionality of the class.
- `new`: must be a function. It is the constructor of the class, and used to create new members of the class.
- `super`: must be a class or a list of classes. It is the super class(es) from which the current class inherit attributes.

All the collective attributes are optional. The following is a class which has all the three collective attributes.

```
global.circle = class
      super = global.ellipse;
      title = "circle";
      new = radius -> ();

      public radius = 1 in _RP;
      auto longaxis = () -> parent.radius;
      auto shortaxis = () -> parent.radius;
      auto area = () -> pi * (parent.radius)^2;
```

```
        auto perimeter = () -> 2 * pi * parent.radius;
    end
```

7.5 The Constructor

The constructor of a class is a collective attribute named `new`. It must be a function that has no output argument. The constructor is just like a normal function, which makes no mentioning of the class. The result of calling the constructor is the creation of a class member. Any local variable of the constructor whose name matches that of a member attribute of the class, will be assigned to the corresponding attribute of the class member. Any attribute of the member whose name is not a local variable of the constructor will be given the default value.

A very simple and effective way to write a constructor is to write a one-liner function. For example

```
student = class
          public name = "xxx xxx";
          public id = "000000";
          public age = 18;
          ...
          new = (name, id, age) -> ();
          end
```

At first sight, the constructor may appear to be doing nothing at all. Actually, when the constructor is called with three input arguments, a function stack is created which has three local variables named `name`, `id`, and `age`. The values of the three local variables are the arguments passed to the constructor when it's called. Because all these three names are also names of member attributes of the class, a class member is created whose `name`, `id`, and `age` attributes are the input arguments of the constructor.

In this example the constructor simply uses the input arguments as values of member attributes without any checking. If one would like to manually check the validity of the input arguments, one can write some code in the constructor to do so. If the checking fails, error can be generated and new class member is not created. Note that if the definition of each member attribute has a domain, then even if the constructor does no explicit value checking, the domain-checking is still performed. Therefore a simple constructor like the above one may still be sufficient.

In the absence of a constructor, when the class definition is processed the interpreter will generate a default constructor. When the default constructor is called, it will create a member of the class all of whose attributes have default values. Note that the default constructor is equivalent to

```
        class
            ...
```

```
            new = () -> ();
            ...
        end
```

7.5.1 Multiple Constructors

A class definition can have only one collective attribute called `new`. However, `new` as a function can accept different numbers of arguments, which makes it possible to initialize a class member in different ways. For example

```
student = class
          public name = "xxx xxx";
          public id = "000000";
          public age = 18;
          ...
          new = function args -> ()
                    if #args == 3
                         name = args # 1;
                         id = args # 2;
                         age = args # 3;
                    elseif #args == 2
                         name = args # 1;
                         id = args # 2;
                    else if #args == 1
                         name = args;
                    end
               end
          end
```

7.6 Inheritance

When defining a new class, all the member attributes of an existing class can be inherited. The existing and the new class are called `super` and `sub` classes respectively.

The super class of a sub class is specified in the sub class definition by setting the `super` collective attribute to a value. For example:

```
global.person =   class
                    public name = "???";
                    public dob = "???";
                    public gender = "F";
                    ...
                  end

global.student =  class
                    ...
```

```
    super = person
end
```

Here `person` should be a value visible in the surrounding scope of the definition of `student` class. Usually it should be a globally defined class name. To avoid potential name clashes, one can use

```
super = global.person
```

Now `student` is a sub class of `person`, all attributes defined for `person` will be created for any member of `student` as well. For example

```
>> s = student.new();
>> s.name  // check s's name, which is inherited from 'person'
>> s.dob   // check s's dob, which is inherited from 'person'
```

There is no language facility provided for calling the constructor of the super class automatically. The sub class needs to write a constructor to handle the initialization of the inherited attributes.

7.7 Multiple Inheritance

When defining a class, the attributes of more than one classes can be inherited, and each of such a class is called a super class. To realize this, the `super` collective attribute in the class definition should be set to a list of classes.

```
super = {s1, s2, s3}
```

7.8 Attribute name clash

When defining a class, attributes must have different names. However, an attribute inherited from a super class may have the same name with an attribute defined in the class itself. And attributes with same names may be inherited from different super classes. In Shang the attribute defined in a class definition always has higher precedence than inherited attributes. For example, if class `z` has a member attribute `bark` but also inherits a member attribute `bark` from a super class. If `m` is a member of `z`, then `m.bark` refers to the attribute defined in `z` not in the super class. But if `z` doesn't have a `bark` attribute, `m.bark` of course refers to the attribute of the super class.

If there is a name clash, the keyword `as` can be used to clarify which class is referred to. For example, if `m` belongs to both `S` class and `T` class, and both classes have member attribute `fern`, then one can use `(m as S).fern` or `(m as T).fern` to eliminate the ambigulty.

7.9 Validation of member attribute modification

Normally a class behaves the way it's designed to. However, if it inherits from a super class, in theory, its members automatically have all the member attributes of the super class, and some of these attributes may reduce a sub class member into illegal state. For example, consider a circle class that inherits from a super class ellipse. The two axes of an ellipse `x` are called `x.a` and `x.b`. `x` is considered an ellipse if `x.a` is equal to `x.b`. However, `x` as an ellipse may be able to change the values of `a` or `b` so that `x.a` is no longer equal to `x.b`. This will cause a circle to enter an illegal state.

In Shang, a class may have a common attribute `validate` which verifies that the state of the class member is valid. Whenever an attempt to modify a class member is made, the modification will be temporarily made and then `validate` is implicitly called. If it returns non-zero value, the validation is passed and the modification will take effect. Otherwise, the modification will be reversed.

Note that `validate` is called whenever an update of class member state is made. If `x.a` must equal to `x.b` in order for `x` to be valid, a simultaneous assignment `(x.a, x.b) = (v, v)` must be done in order to pass validation.

```
global.ellipse = class
        title = "ellipse";
        public a = 1 in (0 to inf);
        public b = 5 in (0 to inf);
        auto longaxis = () -> max(parent.a, parent.b);
        auto shortaxis = () -> min(parent.a, parent.b);
        auto area = () -> pi * parent.a * parent.b;
        new = (a, b) -> ();
end

circle = class
        super = global.ellipse;
        title = "circle";
        auto radius = () -> parent.a;
        auto longaxis = () -> parent.radius;
        auto shortaxis = () -> parent.radius;
        auto perimeter = () -> 2 * pi * parent.radius;
        common set_radius = r -> ((parent.a, parent.b) = (r, r));
        common validate = () -> (parent.a == parent.b);
        new = (a, b) -> ();
end

u = circle.new(3, 3)
u.a = 5; // won't work
```

Note that without the `validate` function, `u.a = 5` will change `u` to an ellipse with `a = 5, b = 3`.

7.10 Class as a Set

A class is naturally also a set that contains all its members. When the class is created, it is an empty set until the constructor is called and a member is added to the set. As a set, a class can be used as the domain of a function parameter, argument, or a class member. For example,

```
>>  Dog = class
              .....
              .....
        end

>>  Spot = Dog.new();

>>  Spot in Dog   // check if Spot is a member of the dog class
          1
>>  Person = class
              ...
              public dog in Dog;
              ...
    end
```

Here a member of the `Person` class has an attribute `dog`, which has to be a member of the `Dog` class.

7.11 Operator Overloading

Operators `+, -, *,` \, etc., can be performed on built-in numerical data types (as well as some other types). Classes are meant to specify a representation of user-defined data types. So what operators can be applied to members of user defined class? Firstly, they have attributes that can be accessed by the operator "`.`". Besides the dot operator, if *addition, subtraction, multiplication, division* and other operations are appropriate for the concept implemented by the classes, they can be made available to be used on the class members directly. This mechanism is called *operator overloading.*

The operators that can be overloaded are `+, -, *, /,  .*, ./, ^ , [],` `.`, including the dot operator for attribute selection. To overload an operator, a member attribute with a specific name needs to be defined. The operator and the attribute names are given in table (7.1).

If class `s` is defined as follows

```
s = class
        common _sum = y -> z
         ...
              end
```

Table 7.1: Overloaded Operators

Operator	Attribute Name
+	_sum
-	_sub
*	_mul
/	_divide
\	_backdivide
[]	_sqbracket
[,]	_sqbracket2
^	_power
.*	_dotmul
./	_dotdivide
.	_field

```
        common _sqbracket = i -> z
         ...
               end

        common _dotmul = y -> z
         ...
               end

        common _field = str -> z
         ...
               end
    end
```

Then, if `x` is a member of class `s`, `x+y` is the outcome of `x._sum(y)`, `x[i]` is the outcome of `x._sqbracket(i)`, `x[i, j]` is the outcome of `x._sqbracket2(i, j)`, `x .* y` is the outcome of `x._dotmul(y)`, and `x.str` is the outcome of `x._field(str)`.

The following example uses overloading to define a sparse matrix class. Note that in Shang does have built-in support for sparse matrix, which much more efficient and feature rich.

```
sparsemat = class
     private index = [1, 1];
     private y = 0;

     readonly nrows = 10;
     readonly ncolumns = 10;
     readonly size = 100;
     readonly nzn = 0;
```

```
new = function (nrows, ncolumns) -> ()
            size = nrows * ncolumns;
      end

common _subasgn = function (n, y) -> ()
          if n >= 1 && n <= parent.size
               indx = [fix((n - 1) / parent.ncolumns) + 1, (n - 1) % parent.ncolu
               for k = 1 : parent.nzn
                     if parent.index[k,:] == indx
                           if y == 0
                               parent.index[k,:] = parent.index[parent.nzn, :];
                               parent.y[k] = parent.y[parent.nzn];
                               --parent.nzn;
                           else
                               parent.y[k] = y;
                           end
                           return;
                     end
               end

               ++parent.nzn;
               parent.index[parent.nzn, :] = indx;
               parent.y[parent.nzn] = y;
          else
               panic("index out of bound");
          end

       end

common _subasgn2 = function (idx1, idx2, y) -> ()
          if idx1 >= 1 && idx1 <= parent.nrows && idx2 >= 1 && idx2 <= parent.nco
               indx = [idx1, idx2];
               for k = 1 : parent.nzn
                      if parent.index[k,:] == indx
                           if y == 0
                               parent.index[k,:] = parent.index[parent.nzn, :];
                               parent.y[k] = parent.y[parent.nzn];
                               --parent.nzn;
                           else
                               parent.y[k] = y;
                               return;
                           end
                      end
               end
```

```
                ++parent.nzn;
                parent.index[parent.nzn, :] = indx;
                parent.y[parent.nzn] = y;
           end
       end

common _sqbracket = function n -> y
          if n >= 1 && n <= parent.size
                y = 0;
                indx = [fix((n - 1) / parent.ncolumns) + 1, (n - 1) % parent.ncolumns + 1];
                for k = 1 : parent.index.nrows
                      if parent.index[k,:] == indx
                            y = parent.y[k];
                            return;
                      end
                end
                y = 0;
          else
                panic("index out of bound");
          end

end


common _sqbracket2 = function (i, j) -> y
          if i >= 1 && i <= parent.nrows && j >= 1 && j <= parent.ncolumns
                y = 0;
                for k = 1 : parent.index.nrows
                      if parent.index[k,:] == [i,j]
                            y = parent.y[k];
                            return;
                      end
                end
                y = 0;
          else
                panic("index out of bound");
          end

end

common _mul = function x -> b
     if parent.ncolumns == x.nrows && x.ncolumns == 1
          b = zeros(parent.nrows, 1);

          for k = 1 : b.nrows
                v = 0;
```

```
                    for j = 1 : parent.ncolumns
                         v += parent[k, j] * x[j];
                    end
                    b[k] = v;
               end
          end
     end

     common _sum = function x -> b
          if x in type(parent) && parent.nrows == x.nrows && parent.ncolumns == x.ncolu
               T = type(parent);
               b = T.new(parent.nrows, parent.ncolumns);
               for k = 1 : x.nrows
                   for j = 1 : x.ncolumns
                        b[k, j] = parent[k, j] + x[k, j];
                   end
               end
          end
     end
end
```

7.12 Acquired Attributes and structure

A member of a class can acquire an attribute that is not a class attribute declared in the class definition. Such an attribute is called an *acquired attribute*. To add such an attribute to a member, we only need to use an assignment statement

```
A.attrib_name = value;
```

For example

```
x.color = "red";
```

Acquired attribute definition may also carry a domain.

```
x.color = "red" in {"red", "gree", "blue"};
```

Specification of domain can occur only at the first time the attribute is defined (the first time a value is assigned to the attribute). If domain is not given, the attribute will have the default domain, which is the set `_ALL`. Subsequent assignment to the attribute can no longer specify a new domain, and the assigned value must belong to the domain of the attribute.

A *structure* is an object which doesn't belong to any class. It therefore has no class attribute and only has acquired attributes. To define such an object, just enclose all attributes in a pair of braces.

```
solution = {x = 3.8, y = -5.2};
```

New acquired attributes can be added to the object and updated at any time, and domain can be specified when it's added.

It is worth mentioning at this time that the keyword *global* is a actually structure that can be accessed in any scope. The attributes of *global* function as global variables.

Chapter 8

Conditional Class

Traditional classes are suitable for representing essential and static identities of objects, but they may be too rigid and inflexible for describing nonessential and volatile characteristics of objects.

A conditional class is a collection of loosely connected objects. Unlike a traditional class, it doesn't "create" new members using the constructor, but issues membership to members of other classes that satisfy certain conditions. Such memberships may be cancelled once the conditions are no longer satisfied.

By using conditional classes, it is possible to avoid unnecessary programming complexity, too many levels of multiple inheritance, and frequent object creations and destructions.

8.1 Defining a Conditional Class

The definition of a conditional class is similar to a regular class, except that

- the keyword `conditional` should be used instead of `class`,
- there are only `common` member attributes, but no `public`, `private`, or `auto` member attributes.
- there are no `new` or `super` collective attributes. Instead, a conditional class can have collective attributes named `title` and `database`. `title` is a string and meant to be a brief description of the class, and `database` is supposed to be a centralized database that stores the information of the members. It needs to be a pointer so that member functions may alter its value.

8.2 Utility Attributes

Three `common` member attributes `join`, `validate`, and `withdraw` are special utility functions of the class, and should be defined according to the following

specifications.

- `join`: this is a function that processes applications for memberships of the class. It takes only one argument, which is the conditional class. Any object `o` can try to join any conditional class `cl`, by calling `obj.join(cl)`. If `obj` satisfies the conditions of the class `cl`, the application is approved, and the value `1` is returned. The `join` function may modify the object, and may write a record to the database of the conditional class.

- `validate`: this is a function that verifies that a certain object is a member of the class. This function is automatically called whenever an object acts as a member of the conditional class, and tries to use a service function of the class. `validate` takes one argument, which is the conditional class, and returns a logical value. Having `join`ed the class does not guarantee an object remains a member of the class forever, so a verification is always necessary.

- `withdraw`: this is the function that performs some tasks on a member of the class when it withdraws from the class. It also takes one argument (the conditional class).

Besides the utility attributes, a conditional class may have a number of `common` or `auto` member attributes. These are the member services provided by the class. Note that the first argument of any of the attribute function is always the class itself, so that there is never a need to resolve name conflict between different conditional classes. When an object tries to use any of the member services, `validate` will be called. Only upon success of `validate`, the service can be provided.

The following code defines an `ellipse`, a regular class and `circle`, a conditional class.

The `validate` function of `circle` tests if a value `x` is an `ellipse` and if `x.a` is equal to `x.b`.

The `circle` class has a member attribute function `perimeter`. To call it, we need to specify the conditional class, e.g., `x.perimeter(circle)`.

The `validate` function is automatically called when the systems tries to evaluate `x in circle` or `x.perimeter(circle)`.

```
global.ellipse = class
    public a = 4 in _R;
    public b = 3 in _R;
    auto  area = () -> pi * parent.a * parent.b;
    new = (a,b)->();
  end


global.circle = conditional
    common validate = function sys -> r
```

```
       if parent in global.ellipse && parent.a == parent.b
          r = 1;
       else
          r = 0;
      end
    end
    common perimeter = sys -> 2*pi * parent.a;
end

c2 = ellipse.new(9,9);
c2.perimeter(circle)
c1 in circle
```

The following defines a conditional class called `ratclub`. In order to join the `ratclub` class, an object must belong to the the `Person` class. Upon joining the `ratclub` class, each new member is giving a `ratid`, and some information of the member is recorded on file. Each time a member service is request, the recorded information will be verified. If the verification fails, the request for service is denied.

```
global.Person = class
     public name = "John Smith";
     public gender = "M" in {"M", "F"};
     public phone = "911";
     public birthdate = "06/01/1985";
     public address = "123 Ashgrove cres., Stonycreek, ON";
     private id = 92052;
     public changeid = function newid -> ()
                          parent.id = newid;
               end
     common alterid = function newid -> ()
                     parent.id = newid;
                end
     auto getid  = () -> parent.id;
end;


global.ratclub = conditional
   database = newpointer(~);
   common join = function sys -> r
            if parent in global.Person
                parent.ratid = # (sys.database)>> + 1;
                (sys.database) >> # (parent.ratid) = {name =
     parent.name, phone = parent.phone, address =
     parent.address, gender = parent.gender, birthdate =
     parent.birthdate, points = 500};
```

```
                r = 1;
            else
                r = 0;
            end
     end

   common validate = function sys -> r
             ratid = parent.ratid;
             record = (sys.database) >> # ratid;
                 if (record.name == parent.name && record.gender ==
        parent.gender && record.birthdate == parent.birthdate
        && record.address == parent.address &&
        record.phone==parent.phone)
                 r = 1;
            else
                r = 0;
            end
      end

    common contribute = function (sys, x) -> r
                     ratid = parent.ratid;
                     ((sys.database) >> # ratid).points += x;

                     r = 1;
             end


    common check_points = function sys -> r
                     ratid = parent.ratid;
                     dbp = sys.database;
                     db = dbp>>;
                     record = db # ratid;
                     r = record.points;
             end

end

x = Person.new();
x.join(ratclub);
x.check_points(ratclub)
```

Chapter 9

Pointers

A pointer is piece of data that contains the information about how to access another piece of data. One may say that a pointer is the abstract *address* of a variable. However, the implementation of pointer has no direct link with the physical memory address of the variable.

When a pointer `p` contains the address of a variable `x`, we say that `p` points to `x`. If two pointers have equal values, they would point to the same variable. This makes it possible for different variables to share data, and create dependencies among different modules of programs.

Pointer is supported in a manner so that programmers can build intricate data structures when necessary, and yet are not forced to use pointers for normal tasks. Pointers are safe since there is no way to do pointer arithmetic, and invalid pointers are detected by the interpreter to avoid interpreter failure. More importantly, since it is not necessary to use pointers all the time, implicit program behaviors can be minimized.

9.1 Pointer Syntax

The operator for creating and deferencing a pointer is `>>`. To create a pointer that points to `x`, one can use

```
p = >>x
```

One can combine the two operators `=` and `>>`

```
p =>> x
```

One may read the above as "let `p` point to `x`"

To retrieve the value pointed to by `p`, one can use

```
v = p>>
```

One may read the above as "Let `v` be the value that `p` points to"

To reset the value that `p` points to

```
p>> = 3
```

or you may use

```
p >>= 3
```

There are two types of pointers. The first is created using

```
p =>> x
```

Here `p` is like the address of a local variable `x`, and through `p >>=` one can reset the value of `x`. Such a pointer cannot be returned as function return value, otherwise, referencing the return value will result in an error.

If `p` doesn't exist, or `p` is not a pointer, then doing

```
p >>= 3
```

will create a pointer that points to a piece of anonymous data. This is similar to the `malloc` function in C. Such a pointer can be return by a function. Here the lvalue must be a variable.

There is also a built-in function `newpointer` that can be used to create a new pointer pointing to given data. For example `p >>= 3` is equivalent to

```
p = newpointer(3);
```

9.2 Pointers and Function Arguments

Shang passes arguments to functions by value, so the called function cannot directly alter the value of an argument in the calling function since what it receives is only a copy of the variable of the caller. For example, if one wishes to write a function to interchange the values of two variables, the following will not work

```
A = "AAA";
B = "BBB";
swap = function (a, b) -> ()
              temp = a;
              a = b;
              b = temp;
        end
swap(A, B);
A
B
```

since what `swap` receives are the copies of `A` and `B`.

However, if pointers to local variables are passed then it's possible to modify the values of the local variables of the calling function. For example, the `swap` function can be implemented the following way

```
A = "AAA";
B = "BBB";
swap = function (a, b) -> ()
              temp = a>>;  // temp gets the value a pointing to
              a>>= b>>;   /*  the location a pointing to gets
                              the value b pointing to
                           */
              b>>= temp;  /* the location b pointing to gets
                             the value of temp
                           */
        end
swap(>>A, >>B);            // pass pointers to A and B to swap
A
B
```

When `swap(>>A, >>B)` is being executed, a copy of `>>A` and a copy of `>>B` are passed to `swap`. However, those two copies still point to `A` and `B` respectively, which are values of the local variables of the caller. Therefore, in function `swap`, when updating the values these pointers pointing to, the values of the local variables of the caller are altered.

9.3 Linked List

There are many ways to define a linked list. First we may select a domain for the data (although it's not necessary) we want to store on the list. Then we can define a class for the list node. For example

```
global._RP = (0 to inf);

global.list_node_type = class
     public data = 1 in _RP;
     public next = [] in newptrnset(global.list_node_type);
     new = (data, next) -> ();
end
```

Now to create a list of nodes and set the data we may do

```
head = list_node_type.new(0,[])
head.next = newpointer(list_node_type.new(1,[]));

p = head.next
p>>.next = newpointer(list_node_type.new(2,[]));

p = p>>.next;
```

```
p>>.next = newpointer(list_node_type.new(3,[]));

p = p>>.next;

p>>.next = newpointer(list_node_type.new(4,[]));
```

9.4 Returning a Pointer

A function may return a pointer as the outcome of a function call. For example

```
create_array = function () -> p
                      p >>= ("a", "b", "c");
               end
array = create_array();
array >> # 3
```

A returned pointer provides piece of data that does not belong to the local storage of any function. Therefore several functions can work on the same data by obtaining pointers to the data.

When a function call is terminated, the values of all the local variables are lost. Therefore pointer to a local variable should not be returned. For example, the following function

```
create_array = function () -> p
                      array = ("a", "b", "c");
                      p = >>array
               end
```

is bad since the return value will not be usable. The function will compile fine. But when the return value is accessed, an error will occur

```
array = create_array();
array >> # 3
```

9.5 Pointer and Class Member

A function may take a class member as argument, it cannot modify the values of the attributes of the class member. For example, if there is a `person` class, whose members have attribute `hairlength`, the following function won't work

```
cut_hair = function p -> r
               p.hairlength /= 2;
           end
```

since the function `cut_hair` will duplicate the argument passed to it and only change the `hair_length` of the duplicate. If we do want to alter the status of a class member, we can pass a pointer to the class member to the function.

```
cut_hair = function p -> r
                p >>. hairlength /= 2;
           end
p = person.new();
cut_hair(>>p);
```

9.6 Pointer and Program Efficiency

Pointers can be used to build structures like linked lists and trees, and share data among functions and modules. Otherwise pointers can be avoided and there is little need to use pointers merely for the sake of efficiency. In fact, using pointer can often make the program less efficient as Shang pointers are not really memory addresses and are implemented in an abstract manner to ensure safety and generality.

When a function is called, the value of the argument is passed to the function but the copy is not made until the function call attempts to alter the value of the argument. So if the memory usage and performance cost of copying function arguments has been a concern, one may just design the function such that the argument values are not overwritten.

Similarly, assigning `a` to `b` does not cause a copy of `a` made immediately. Although `a` to `b` are supposed to be independent data, they share the same storage until one is being changed. Even when such an event occurs, the copying process is still likely to be very efficient, especially when the variable is a list, table, class, function, class or object, in which case, each component of the copy is just a temporary link to the component of the old variable, except the part that is being updated.

9.7 Use pointer to emulate reference

In the Java programming language, if `p` is reference to an object, then `q = p` will make a reference to the same object, and statement like `q.a = 0` will alter the status of `p` as well. This is not the case in Shang, as `q = p` will make a duplicate of `p`. If one wishes to emulate the behaviors of object references in Java, one can design a front end class which has an attribute that is a pointer pointing to the real data. Thus copies of the front end refer to the same data. For example

```
global.person_data = class
   public gender = "M" in {"F", "M"};
   public age = 1 in 1 : 150;
   public first = "Mark" in ~/[A-Za-z][A-Za-z]*/;
   public last = "Brown" in ~/[A-Za-z][A-Za-z]*/;
   new = (gender, age, first, last) -> ();
end
```

```
global.person_ref = class
   private data_p;

   auto gender = () -> (parent.data_p) >>. gender;
   auto age = () -> (parent.data_p) >>. age
   auto first = () -> (parent.data_p) >>. first
   auto last = () -> (parent.data_p) >>. last

   new = function (gender, age, first, last) -> ()
      data_p = newpointer(global.person_data.new(gender, age, first, last));
   end

   common set_gender = function x -> ()
                          (parent.data_p) >>. gender = x;
                       end

   common set_age = function x -> ()
                          (parent.data_p) >>. age = x;
                       end

   common set_first = function x -> ()
                          (parent.data_p) >>. first = x;
                       end
   common set_last = function x -> ()
                          (parent.data_p) >>. last = x;
                       end
end

p1 = person_ref.new("M",25,"Derek", "Burke");
p1.set_age(29);
```

Now `p1` acts as a reference to the actual person data. Copies of `p1` are like aliases of the same person. If a copy of `p1` is updated, the original `p1` will undergo the same change.

```
p2 = p1;
p2.set_age(35);
p2.age
    // ans is 35
p1.age
    // ans is 35 also
```

Chapter 10

Errors and Exception Handling

10.1 Parsing Errors and Run-time Errors

Parsing errors are those detected by the interpreter when examine the code and translating it into commands executable by the computer. Most such errors are related to syntax. When a parsing error is encountered the interpreter generates an error message and stips the parsing of the rest of the program. For example

```
>> x = [3, 5,]
   Error: unexpected closing bracket
```

A program may have passed the phase of parsing, but during execution may still run into an abnormal event that must cause the program to abort. Such an error is called a run-time error or exception. Typical run-time errors include out-of-bound array indices and function domain errors. For example

```
>> s = function x -> y
     y = x[1] + x[2];
   end
>> s(3)
   Error: index out of bound
```

When the above piece of code is processed, the interpreter doesn't find any syntax error. The expression `s(3)` appears to be ok since it's very hard for the interpreter to note the fact that function `s` expects a matrix with length at least 2. The detection and handling of such an error is delayed untill run-time.

A function may specify a domain, in which case, the interpreter may check domain error during parsing time and detect such errors. In general it's difficult to tell what kind of errors are caught at which time unless user programs are meticulously designed to ensure that all invalid argument values are caught by the interpreter and not passed to the function.

When an error occurs during the execution of a program, the default action of the interpreter is to display some error message and information about where the error occurs, and then execution of the commands and functions is terminated. The interpreter returns to the interactive level and waits for new commands.

Alternatively, a user program may try to 'catch' and handle an error, and continue to do other tasks from the point where the error is caught. This way the program flow is not interrupted, and the interpreter is not brought all the way down to the user interactive level.

10.2 Raising Exceptions

Whenever an unexpected error occurs, the `throw` statement can be used to "throw" an exception, and abort the current function (all subsequent commands in the function will be skipped). The program flow is reduced to the user interactive level, unless the thrown exception is caught by a '`catch` command. The `throw` statement takes a single parameter, which is an identifier (not quoted) used as the name of the exception. All the exceptions are stored as global names and form their own name space. There is no need to pre-define an exception. At the first time an exception is thrown or caught, the exception will be defined and the name added to the database of exceptions, and can be referred to later.

10.3 Handling Exceptions

To catch an exception that may occur during the execution of a sequence of code, one can enclose the code in a `try -- catch` block as follows

```
try
      ...
      throw exception_name;
      ...

catch  exception_name
      /* exception handling commands here */
end
```

where `exception_name` is an identifier used to represent the exception. All the names of exceptions form a global name space and are independent from local or global variable names. For example

```
function T = rctrap(f, a, b, n)
   x = [a, b];
   try
      y = f(x);
   catch no_vec
      // f is not vectorized; proceed accordingly
```

```
   end
      // f is vectorized; ...
   ...
end
hump = map x -> y
    if x.length != 1
         throw no_vec;
    end
    // other stuff
end
```

In this example, the function `hump` is not vectorized and throws an exception `no_vec` if called with a non-scalar. The function `rctrap` takes a function as the first argument and expects it to be vectorized. It catches the `no_vec` exception and does not perform the computations if the argument passed to it throws the error.

At present, errors caught by the interpreter are not (yet) exceptions and cannot be caught. Example of such errors are index out of bound error, undefined data attribute, etc.. When such errors occur, the interpreter returns to the user-interactive level, and the variables at the interactive level and global variables are left with uncertain values. This might be an inconvenience but it's possible to write throw/catch blocks to manually detect, throw, and handle these errors and bypass the interpreter's automatic error handling mechanism.

Chapter 11

File, Input, and Output

In this chapter we present some of the facilities that enable the programs to intereact with their environment, i.e., the operating system and the user. At present our implementation is just the frontend of a minimalistic selection of the C standard input and output library routines. More operating system specific functionalities will be added later.

11.1 File

The file access model is closely based on the standard C library `stdio.h`. An opened *file* is a value and can be assigned to a variable. The `file` value has a number of attributes that make it possible for the program to check the file status and read from/write to the file.

A `file` object is not the same as the file stored on the computer. It exists only after the physical file is opened as an I/O stream associated with the interpreter program. Once it is *closed*, the file value ceases to exist in the perspective of the Shang interpreter (until the file is reopened).

A `file` is created using the built-in function `fopen`

```
f = fopen("file_name", "mode");
```

where `file_name` is the path of the file that is absolute or relative to the current working directory, and `mode` is the string that contains the options. The valid modes are

`r`: open text file for reading

`w`: create text file for writing; discard previous contents if any

`a`: append; open or create text file for writing at end of file

`r+`: open text file for update (i.e., reading and writing)

`w+`: create text file for update; discard previous contents if any

`a+`: append; open or create text file for update, writing at end

The mode may contain an additional character `b`, such as `w+b`, which indicates a binary file.

If `fopen` is called successfully, the returning value is a `file`, which has the following attributes (assuming that `f = fopen(fname, mode)` has been executed):

`f.name`: a string that contains the absolute path of the file name

`f.mode`: the mode of the file

`f.status`: 1 if the file is open, or 0 if it's closed.

`f.readline()`: reads a line from the file. If end of file is reached, `null` is returned.

`f.writeline(str)`: writes a line of text string `str` to the file.

`f.close()`: closes the file.

`f.open()`: open the file (if it's closed).

`f.eof`: 1 if the current position of the file is at end, 0 otherwise.

`f.read(n)`: read `n` bytes from the file. Return value is a vector of bytes.

`f.write(x)`: write data `x` (a character string, a scalar, or a vector) to the file.

`f.rewind()`: set the current position to the beginning of the file.

`f.flush()`: causes any buffered but unwritten data to be written. the output buffer of the file.

`f.getpos()`: get the current position (an integer) of file.

`f.setpos(n)`: set the current position of a file to a given value.

`f.tell()`: get the current position (an integer) of file.

`f.seek(whence, offset)`: set the file position of the stream to a given offset. `whence` can be `"beginning"`, `"current"`, or `"end"`.

Example: the following makes a copy of a file.

```
f1 = fopen("pinks.txt", "r");
f2 = fopen("dianthus.txt", "w");
if f1 && f2
     line = f1.readline();
     while line != null
          f2.writeline(line);
          line = f1.readline();
```

```
    end
end
f1.close();
f2.close();
```

Note that `file` values behave like pointers or references. If you pass a file to a function, the function can write to the same physical file on the disk, instead of a copy of the file (the caller and the called function have the same file instead of two independent copies). If `f` is a file, after `g = f`, `g` will refer to the same physical file. This is unlike any other type of value. But this is not a contradiction to the *everything is a value not a reference* principle, since the actual files are not part of the system.

11.2 Output

If a statement is an expression that has a value, and the statement ends with a newline without a trailing semicolon, then the value of the statement will be displayed. If the value is a numerical scalar, matrix, character string, or a list of these values, it will be displayed the usual way. If the value is a more "abstract" one, such as a function, a hash table, or, a class, the system will try to print some information about it. For example:

```
>> x = rand(1)
   0.08847010159
>> y = rand(2)
   0.346
   0.53
>> sin
   Builtin function
>> f = x -> sqrt(1+x^2)
   user defined function
```

To suppress unwanted display of answers, a semicolon should be appended at the end of the statement.

The built-in function `print` will display the value of its operand, even if the line that contains `print` ends with a semicolon. The `print` function takes one argument, and display its value, but does not return anything. For example

```
>> print(cos);
   Builtin function
```

How values are displayed by `print` function is defined by the system, and cannot be modified. To display data in customized ways, the formatted output function `printf` and `sprintf` can be used.

11.3 Input

The built-in function `readline` can be used to read a line from the command input. For example, after the following,

```
mesg = readline();
```

a character string is read from keyboard until enter is hit. Besides `readline`, `sscanf` and `fscanf` can read formatted input from a string or a file.

11.4 Formatted IO

11.4.1 printf

The function `printf` converts a sequence of values to strings in specified format and display the string. The first argument is a format string that contains all text that is to be printed literally and the formats in which the following arguments are to be converted. Each format specification starts with a % sign. The rest of the arguments are values to be converted.

For example

```
>> printf("e = %g, pi = %g\n", e, pi);
   e = 2.71828, pi = 3.14159
```

If `f` is a file, then it has a `printf` attribute function. For example,

```
>> f.printf("e = %g, pi = %g\n", e, pi);
```

will write the string `e = 2.71828, pi = 3.14159` to the file. Alternatively, we can also use the built-in function `fprintf` to write a formated string to a file

```
>> fprintf(f, "e = %g, pi = %g\n", e, pi);
```

11.4.2 sprintf

The `sprintf` function is the same as `printf` except that it does not print the result of the conversion but returns it as a string. Of course, if the trailing semicolon at end of the `sprintf` command is not present, then the returned string is still printed.

```
>> s = sprintf("e = %g, pi = %g\n", e, pi);
>> s
   e = 2.71828, pi = 3.14159
```

Normal characters contained in the format string are printed literally, such as the strings `e =` and `pi =` and `\n`. Each substring in the format that starts with the character `%` and ends with a conversion character is called a conversion specification. Valid conversion characters include `d, f, g`, etc.. For example, in the format `k = %5d, e = %g, pi = %g\n`, `%5d` and `%g` are conversion specifications, and the rest are normal characters.

Between the % sign and the type character there may be, in order,

- **Flags** (in no particular order), which modify the print specification

  `-`: left adjustment.

  `+`: the number will be printed with a sign.

  space: if the first character is not a sing, a space will be prefixed.

  `0`: fields for numbers are padded with leading zeros.

  `#`: number will be printed in alternate form. For type `o`, the printed number starts with zero. For `x` or `X`, the number printed starts with `0x`. Floating point numbers will always contain the decimal point. And trailing zeros in format `g` are kept.

- **Minimum field width** This is an integer `w`. It specifies that the converted value will be printed in at least `w` characters. If the value is shorter than `w` characters, the field width will padded left or right (depending on if left adjustment flag is present) with space or zeros (if zero flag is given).

- **A period** that separates the minimum field width and the precision.

- **Precision** This is an integer `p`. For

  strings: at most `p` characters will be printed

  integers: at least `p` digits will be printed (leading zeros might be added)

  `e,E,f` conversions: `p` digits after the decimal point will be printed

  `g,G` conversions: `p` significant digits will be printed

- **A length modifier** `l`, `L`, or `h`, which indicate that the corresponding numbers are to be printed as `long`, `long double`, and `short` respectively. For definitions of these types please refer to a C standard library manual.

The following is a list of the conversions that the functions of the `printf` family can perform.

Character	Type of Value
`d,i`	integer of signed decimal format
`o`	integer of unsigned octal format
`x,X`	integer of unsigned hexadecimal format
`u`	integer of unsigned decimal format
`c`	ASCII character
`s`	character string
`f`	floating point number of double precision, decimal notation
`e,E`	floating point number of double precision, scientific notation
`g,G`	floating point number of double precision; scientific notation if size of exponent is too big, decimal notation otherwise.
%	print a %

11.4.3 sscanf

`sscanf` is opposite of `sprintf`. It scans a string, interprets the characters according to the specified format, and convert them into a list of values. It takes two input arguments. The first is the string to be scanned. The second is the format string. The return value is a list of the values it has found from the input string. For example

```
>> sscanf("adfadf 32 sdf 12.1", "%s %d %s %f")

   (adfadf, 32, sdf, 12.1)
```

If `f` is a file, then it has the `scanf` attribute function. For example,

```
>> f.scanf("%s %d %s %f")
```

will try to read a string, an integer, and a floating point number from the file (separated by white spaces)). Similarly to `fprintf`, we can also use

```
>> fscanf(f, "%s %d %s %f")
```

where `f` is the file being scanned.

The following is a list of the conversions that the functions of the `scanf` family can perform.

Table 11.1: Scanf conversions

Character	Type of Value
`c`	single character
`d`	decimal integer
`o`	octal integer
`x`	hexadecimal integer
`e, f, g`	floating point number
`s`	character string
`%`	literal %, no conversion

.

Chapter 12

Automaton

An automaton is a program that can pause and restart. It has its own local variables and a sequence of instructions. The instructions are executed once the automaton starts running. Once it stops the flow control is returned to the interpreter, but the values of the local variables and the status and position of the instruction sequence are retained, and when the automaton restarts, it will restore the local variable values and the previous execution status.

An automaton doesn't output or return any values, nor does it take any input arguments when it starts. Instead, it works like a computer, and may have a number of ports, which are two-way channels used for data exchange. Data can be manually added to or removed from the ports, but automatons can also be connected through the ports to form a network, in which case one automaton's certain output channel becomes another automaton's input channel.

The purpose of automaton is to simplify complex flow controls and simplify function calls and data exchange. It may help design event-driven programs. At present automatons are an experimental feature. The future version might be implemented using threads provided by the operation system.

12.1 Define Automaton

The definition of an automaton starts with the keyword **automaton** and ends with the keyword **end**. In the middle there are two parts. The first part is optional. It follows right after the `automaton` keyword, and declares a list of attributes, which are enclosed in a pair of brackets. These are much like the parameters of functions. They are used like local variables and can be accessed and updated outside the automaton using the dot operator like attributes of class members.

The second part of the automaton is the sequence of commands and statements, like the body of a function. The keyword **this** can be used in the commands and provides a reference to the automaton itself.

12.2 Port

An automaton has a number of ports, through which data input and output are performed. There is no limit on the number of ports. If `K` is the maximum port number used in the automaton, then `K` ports will be created. Each port has two channels — input channel and output channel. Each channel is like a queue — data comes in from one end, form a chain, and goes out from the other end.

Inside the automaton, to output a piece of data x to port 1

```
this.put(x)
```

or to output a piece of data x to port n

```
this.put(x, n)
```

To remove the next available piece of data from port 1, and assign it to `x`

```
x = this.get()
```

or to remove the next available piece of data from port n

```
x = this.get(n)
```

Two attribute functions `input` and `output` can be used to send a piece of data to a port of an automaton, or receive a piece of data. They are used outside the definition of the automaton, i.e., in the surrounding scope of the automaton. For example,

```
A.input(x)   // input data at port 1
A.input(x, 2)   // input data at port 2
x = A.output()   // output data at port 1
x = A.output(2)   // output data at port 2
```

Example

```
u = automaton
        n = 0;
        while n == 0
             n = this.get();
             while n > 0
                  this.put(n);
                  --n;
                  stop;
             end
        end
    end
```

12.3 Run Automaton

To run an automaton A, write

```
A.run()
```

The sequence of commands in the body of the automaton will be executed from where it left off last time (or from the beginning of the body, if it's the first time the automaton is running), until a **stop** statement is reached, or until the end of the body. Usually the body of an automaton is inside a loop so that the end is never reached.

When the `stop` statement is reached, the execution of the automaton is suspended. The control of flow is returned to wherever the `run` command is issued. The values of all the local variables and parameters are retained and when the automaton is called a next time, it will start right after the `stop` statement that terminated the automaton last time.

It is also possible to use the keyword `yield` to transfer program control to another automaton. For example

```
global.u2=0;

global.u1 = automaton
    for k = 1 : 5
         this.put(k);
         yield global.u2;
    end
end

global.u2 = automaton
    while 1
         k = this.get();
         k * pi
         yield global.u1;
    end
end

u1 <-> u2;

u1.run()
```

Here the two automatons take turns to run, and yield to each other. They are called *coroutines*.

12.4 Connections between Automatons

When two ports of two automatons, say `A` and `B`, are connected, one automaton's input becomes the other one's output. So when one automaton outputs some

data, the other one can receive it from the connected port. There are several types of connection.

- Two-way connection

  ```
  A <-> B;
  ```

  The input channel of `A`'s port 1 becomes the output channel of `B`'s port 1. And vice versa.

- Right connection

  ```
  A >-> B;
  ```

  The output channel of `A`'s port 1 becomes the input channel of `B`'s port 1.

- Left connection

  ```
  A <-< B;
  ```

  The input channel of `A`'s port 1 becomes the output channel of `B`'s port 1.

In order to specify a port other than port 1, we will need to use a list which contains the automaton and the port index. For example, to connect port 2 of `A` and port 5 of `B`

```
(A, 2) <-> (B, 5);
```

Chapter 13

Built-in Functions

13.1 System utility functions

13.1.1 `run`

The function `run` takes a single parameter, which must be a character string that represents a file name. It will change the current input device to the file specified. The file is supposed to contain any Shang commands, programs, or functions. The contents of the file will be executed, and upon end of file is reached, the input device is changed back to the previous setting (usually user input).

The file name can be an absolute path name, such as

```
run("D:\my programs\test_program.x")
```

No special file extension is required, although .x is preferred.

Relative path name can be used, in which case, the current working directory will be added to the path name. The current working directory can be checked and reset using the system commands `pwd` and `cd`.

13.1.2 `with`

The function `with` is very similar to `run` except that it only run the file once, and subsequent calls to `with("file")` will not run file unless the file has been updated.

13.1.3 `pause`

The function `pause` will suspend the execution of current program until a key input is read. `pause(mesg)` prints a mesg (a character string) and wait until the user presses a key.

13.1.4 panic

The function `panic` takes one parameter which is an error message. It will terminate the execution the current program or command, clear all function stacks, return to the interactive level (where user can type in commands), and display the error message.

13.1.5 clock

The function `clock` takes no argument, and returns the clock ticks since the start of the program (the interpreter). The system clock usually ticks 1000 times per second, so that `clock()` returns the amount of milliseconds that have passed since the invoke of the interpreter.

13.1.6 etime

The function `etime` takes one or two arguments. The arguments are the numbers of clock ticks (returned by `clock()`). When called with two arguments like `etime(c1, c2)`, the time duration between `c1` and `c2` in seconds is returned. When called with one argument, like `etime(c)`, the time duration between `c` and present (in seconds) is returned.

13.1.7 time

The function call `time()` returns the current date and time as a string. For example

```
>> time()
   Sun Mar 02 12:49:23 2008
```

13.2 Elementary Math functions

13.2.1 Trigonometric functions - vectorized

All of the following functions take a numerical value, scalar or matrix, real or complex, as argument.

`sin(x)`	sine of x
`cos(x)`	cosine of x
`tan(x)`	tangent of x
`asin(x)`	arcsine of x
`acos(x)`	arccosine of x
`atan(x)`	arctangent of x
`sec(x)`	secant of x
`csc(x)`	cosecant of x

13.2.2 Exponential and power - vectorized

All of the following functions take a numerical value, scalar or matrix, real or complex, as argument.

`exp(x)`	exponential function e^x
`ln(x)`	natural logarithm $\ln x$. Parameter: `base` with default value e
`log10(x)`	$\log_{10} x$: logarithm with base 10
`log2(x)`	$\log_2 x$: logarithm with base 2
`log(x)`	$\log_{base} x$: logarithm with base `base`
`sqrt(x)`	square root $\sqrt{x}$
`cbrt(x)`	cubit root $\sqrt[3]{x}$
`sinh(x)`	hyperbolic sine
`cosh(x)`	hyperbolic cosine
`tanh(x)`	hyperbolic tangent

The function `log` has a parameter `base`, and `log(x)` is equal to $\log_{base} x$. The default value of `base` is `e`. To change `base` to 3

```
>> global.log.base = 3;
```

To spawn a new copy of `log` with `base` 3

```
>> f = log[3];
```

13.2.3 Polynomial

The built-in function `poly` is a polynomial, which has a public parameter `coeff`. The default value of `coeff` is 1, so that `poly(x) = 1` regardless of `x`. To make a nontrivial polynomial, one needs to reset the value of `poly.coeff`

```
>> global.poly.coeff = [1, 2, -2];
```

or spawn a new function like this

```
>> p = poly;  // making a copy of poly
>> p.coeff = [1, 2, -1];
```

or

```
>> p = poly[[1, 2, -2]];
```

Now `p` is a polynomial $p(x) = 1 + 2x - x^2$. `p` can be called like a normal function

```
>> p = poly[[1, 2, -2]];
>> p(100)
   -9799
```

13.3 Matrix

13.3.1 Creation and initialization of matrices

The built-in functions `zeros`, `ones`, `rand`, `band`, `binary`, `sparse`, `upper`, `lower`, and `symm` are used to generate vectors and matrices.

The function `zeros` returns vectors or matrices of zeros.

`zeros(n)`	$n \times 1$ column matrix of zeros (double precision)
`zeros(m, n)`	$m \times n$ matrix of zeros (double precision)
`zeros(n, domain)`	$n \times 1$ column matrix of zeros; storage type specified by `domain`
`zeros(m, n, domain)`	$m \times n$ column matrix of zeros; storage type specified by `domain`
`zeros(m1, m2, m3, ...)`	$m_1 \times m_2 \times m_3 \times \cdots$ multi-dimensional matrix of zeros (double precision)

The values of `m`, `n`, and `m1, m2, m3, ...` are positive integers. `domain` is one of the following built-in set functions

`_Z`	machine integers
`_L`	arbitrary size integers
`_B`	bytes
`_M`	arbitrary precision floating point numbers

The function `ones` and `rand` are similar to `zeros`; `ones` create matrices of 1's and `rand` creates matrices of random numbers.

The function `rand` has a public parameter `range`, which is a vector of two numbers. The `rand` function returns random numbers that are uniformly distributed between them.

For example, to create a 5×1 matrix of random machine integers

```
x = rand(5, 1, _Z);
```

or

```
x = rand(5, _Z);
```

To create a 5×3 matrix of arbitrary precision random floating point numbers

```
x = rand(5, 3, _M);
```

To change the parameter `range` of `rand`, use either

```
global.rand.range = [0, 10];  // alter the range parameter of rand
```

or

```
f = rand[[0, 10]];  // spawn a copy of rand with new range
```

The function `sparse` creates sparse matrices of double precision floating point numbers. `sparse(m, n)` return sparse $m \times n$ matrix, and `sparse(n)` return sparse $n \times 1$ matrix (a row vector).

The function `band` creates banded square matrices of double precision floating point numbers. `band(n, l, u)` returns an $n \times n$ banded matrix with with `l` sub-diagonals and `u` super-diagonals.

The functions `upper`, `lower`, and `symm` creates square upper triangular, lower triangular, and symmetric matrices respectively. The usage is `upper(n)`, `lower(n)`, `symm(n)`, where `n` is the number of rows (and the number of columns) of the matrix.

Note that `sparse`, `band`, `upper`, `lower`, and `symm` all create matrices of zeros. The elements of these matrices can be altered using the normal assignment operations.

Note also that the matrices created by `sparse` and `band` use special storage schemes and may save on memory as well as make operations more efficient. Matrices created by `upper`, `lower`, and `symm` use the regular dense storage scheme. Although it is not memory efficient, some linear algebraic routines may still take advantage of the fact that the matrix is triangular or symmetric.

Binary scalars and matrices are created using function `binary`.

`binary(n)`: $n \times 1$ column binary matrix of zeros

`binary(m, n)`: $m \times n$ byte binary matrix of zeros

`linspace(a, b)`: create a row vector of 100 elements evenly spaced between `a` and `b`.

`linspace(a, b, n)`: create a row vector of n elements evenly spaced between `a` and `b`.

13.3.2 Basic attributes of matrices

The following functions apply to all numeric scalar and matrix `x`.

`abs(x)`: absolute value of x

`arg(x)`: angular argument of x

`real(x)`: real part of x

`imag(x)`: imaginary of x

`conj(x)`: complex conjugate of x

`ceil(x)`: round x toward infinity

`floor(x)`: round x toward minus infinity

`fix(x)`: round x toward zero

`round(x)`: round x of toward the nearest integer

13.3.3 Other functions

`reshape(x, m)`: change the dimension of `x` to $m \times 1$.

`reshape(x, m, n)`: change the dimension of `x` to $m \times n$.

`reptile(x, m)`: replicate `m` copies of `x` and stack them vertically to form a new matrix.

`reptile(x, m, n)`: replicate mn copies of `x` and stack them like an $m \times n$ matrix to form a new bigger matrix.

13.4 Linear Algebra

Suppose that `A` is a matrix.

`lu(A)`: LU factorization of matrix A. Find matrices `P, L, U` such that `P A = L U`, where `L` and `U` are lower and upper triangular matrices and `P` is row permulation of identity matrix. `A` has to be square. Does not return `L`, `U`, or `P`. Instead, stores factorization result internally so that subsequent computations involving `A` will take its advantage. This procedure is usually unnecessary as using the backslash operator to solve lineary systems will automatically invoke LU factorization and memorize the result.

`LU(A)`: LU factorization of matrix A. Returns the list of matrices `(L, U, P)`, such that `P A = L U`.

`qr(A)`: QR factorization of matrix A. Find matrices `Q, R` such that A = QR, where `Q` is orthogonal matrix and `R` is upper triangular matrix. `A` doesn't need to be square. `qr(A)` doesn't return `Q` or `R`. It just stores the factorization results internally and use them in subsequent least square computations involving `A`.

`QR(A)`: QR factorization of matrix A. Returns the list of two matrices `(Q, R)`.

`svd(A)`: singular value decomposition of matrix A. Find matrices `U, S, V` such that `A = U'SV`, where `U` and `V` are orthogonal matrices and `S` is diagonal matrix. Doesn't return `U`, `S`, or `V`, just store them internally for late use.

`SVD(A)`: singular value decomposition of matrix A. Returns the list of two matrices `(U, S, V)`.

`eig(A)` eigenvalues and eigenvectors of square matrix A. Returns a list `(d, v)`, where `d` is a column vector of eigenvalues of `A`, and the columns of `v` are the corresponding eigenvectors.

`norm(A, p)`: returns the `p` norm of vector or matrix `A`. `p` can be `1, 2, inf`, or `"fro"`.

`norm(A)` returns the 2 norm of `A`.

`cond(A, p)`: returns the condition number of matrix `A` using matrix `p` norm.

`cond(A)`: returns the condition number of matrix `A` using matrix `2` norm.

`rank(A, t)`: returns the rank of matrix `A` using threshold `t`.

`rank(A)`: returns the rank of matrix `A` using threshold `eps`.

`det(A)`: returns the determinant of square matrix `A`.

`inverse(A)`: returns the inverse of square matrix `A`.

`cholesky(A)`: returns the Cholesky decomposition of positive definite matrix `A`.

`identity(n)`: returns $n \times n$ identity matrix.

`sign(A)`: returns the sign of `A` if `A` is a vector; or the matrix of the signs of the elements of matrix `A`.

`arg(A)`: returns the angles on the complex plane corresponding to each element of matrix `A`.

`dot(u, v)`: dot product of vectors `u` and `v`

`cross(u, v)`: cross product of two 3D vectors `u` and `v`

`trace(A)`: the trace of square matrix `A`.

`trans(A)`: the transpose of matrix `A`.

`sparsity(A)`: returns the ratio between memory savings and logical size for matrix `A`. For normal dense matrix, always returns zero. No zero value is returned only for sparse matrix, banded matrix, and range (those created by `zeros`, `ones`, and `linspace`).

`isnan(x)`: returns `1` if `x` is `NaN`, and `0` otherwise.

`hasnan(x)`: returns `1` if at least one entry of `x` is `NaN`, and `0` otherwise.

`isfinite(x)`: returns `1` if each element of `x` is finite, 0 otherwise.

13.5 String and regular expression

`strcmp(str0, str1)`: compare two strings alphabetically; returns -1, 0, 1 if `str0` is `<`, `==`, or `>` `str1` respectively.

`trim(str)`: returns a copy of `str` with leading and trailing blanks removed.

`ltrim(str)`: returns a copy of `str` with leading blanks removed.

`rtrim(str)`: returns a copy of `str` with trailing blanks removed.

`split(str)`: split `str` at the spaces. For example, `split("My gold fish is evil")` return list `("My", "gold", "fish", "is", "evil"`.

`split(str, sep)`: split `str` at the positions where string `sep` occurs.

`regexp(s)`: has three parameters `patter`, `options`, and `match`. match string `s` against regular expression `pattern`. Return the position of the first occurence.

`reggexp(s)`: has three parameters `patter`, `options`, and `match`. match string `s` against regular expression `pattern`. Return the positions of all the occurences.

`regsub()`: has three parameters `patter`, `substitute`, and `options`. match string `s` against regular expression `pattern`. Replace first occurence with `substitute`.

`reggsub()`: has three parameters `patter`, `substitute`, and `options`. match string `s` against regular expression `pattern`. Replace all occurences with `substitute`.

13.6 Math functions

`factorial(n)`: the factorial of an integer `n`.

`choose(m, n)`: returns the number $\frac{n!}{m!(n-m)!}$

`gcd(x, y)`: the greatest common divisor of `x` and `y`

`lcm(x, y)`: the least common multiple of `x` and `y`

`splinefit()`: returns a cubic spline function (see the following function `cspline`).

`splinefit(x, y)`: free cubic spline

`splinefit(x, y, yp0, yp1)`: clamped boundary condition cubic spline

`splinefit(x, y, type)`: cubic spline; type may be `"cyclic"`, `"periodic"` (same as `"cyclic"`), or `"not-a-knot"`.

`cspline(x)`: has two parameters `xnodes` and `coeff`. Returns the evaluation of cubic spline evaluated at `x`. This function is the return value of `splinefit`.

`int(f, I)`: the numerical integral of function `f` over interval `I`

`dsolve(ode, I, y0)`: the numerical solution of the initial value problem over the interval `I` defined by the ordinary differential equation `ode` with initial value `y0` at the start of `I`

13.7 Creation and initialization of data of other types

`newlist(n)`:

`newlist(n)` returns an empty list with buffer size `n`.

`newstring(n)`:

`newstring(n)` returns an empty string with buffer size `n`.

`newstring(s, n)` returns a copy of string `s` with buffer size extended to `n`.

`stack()`: returns a new empty stack

`queue()`: returns a new empty queue

`newpointer(x)`: create a pointer that points to `x`

`newptrset(s)`: create a set of pointers that point to a value in the set `s`.

`newptrnset(s)`: create a set of pointers that point to a value *not* in the set `s`.

`convert(x, type)`: convert value `x` into new type `type`. Type may be specified by the built-in set function `_D`, `_Z`, `_B`, `_M`, `_L`, `_S`, `_C`, or character strings `"D"`, `"Z"`, `"B"`, `"M"`, `"L"`, `"S"`, `"C"`.

13.8 Data processing and statistics

`x` should be numerical vector or matrix.

`fft(x)`: fast Fourier transform of vector `x`. If `x` is matrix or multidimensional matrix, return 2D or multidimensional transform.

`ifft(x)`: inverse fast Fourier transform of `x`.

`sum(x)`: sum of all elements of vector or matrix

`mean(x)`: mean of all elements of vector or matrix

`max(x)`: maximum of all elements of vector or matrix

`min(x)`: minimum of all elements of vector or matrix

`range(x)`: minumum and maximum of all elements of vector or matrix, returned as a vector

`var(x)`: variance of all elements of vector or matrix

`stddev(x)`: variance of all elements of vector or matrix

`rowsum(x)`: sum of all elements in each row

`rowmean(x)`: mean of all elements in each row

`rowmax(x)`: maximum of all elements in each row

`rowmin(x)`: minimum of all elements in each row

`rowrange(x)`: minumum and maximum of all elements in each row returned as a vector

`rowvar(x)`: variance of all elements in each row

`rowstddev(x)`: variance of all elements in each row

`columnsum(x)`: sum of all elements in each column

`columnmean(x)`: mean of all elements in each column

`columnmax(x)`: maximum of all elements in each column

`columnmin(x)`: minimum of all elements in each column

`columnrange(x)`: minumum and maximum of all elements in each column returned as a vector

`columnvar(x)`: variance of all elements in each column

`columnstddev(x)`: variance of all elements in each column

13.9 Sorting and searching

`x` can be a numerical vector, matrix, or a list of numbers or strings.

`sort(x)`: sort in ascending order. Treat `x` as row vector when it is a matrix.

`rowsort(x)`: sort each row in ascending order

`columnsort(x)`: sort each column in ascending order

`indexsort(x)`: return the index vector of the sorted copy of `x`

`rowindexsort(x)`: return the index matrix of the of `x` with each row sorted

`columnindexsort(x)`: return the index matrix of the of `x` with each column sorted

`reverse(x)`: reverse

`rowreverse(x)`: reverse each row

`columneverse(x)`: reverse each column

If one wishes to sort all the rows according to the value of a column, an index sort can be done to that column, then the sorted index is applied to the matrix. For example,

```
x = rand(8,8);
I = indexsort(x[:,1]);
x = x[I, :];
```

`find(x, v)`: find `v` in `x` (a vector, matrix, or list). Returns the index of the first occurence.

`findall(x, v)`: find `v` in `x` (a vector, matrix, or list). Returns the indices of the all occurences.

`sfind(x, s)`: find an entry of `x` that belongs to set `s`. Returns the index of the first occurence.

`sfindall(x, s)`: find entries of `x` (a vector, matrix, or list) that belong to set `s`. Returns the indices of all occurences.

`findmax(x)`: find the maximum of `x`. Returns the index of the first occurence.

`findallmax(x)`: find the maximum of `x`. Returns the indices of all the occurences.

`findmin(x)`: find the minimum of `x`. Returns the index of the first occurence.

`findallmin(x)`: find the minimum of `x`. Returns the indices of all the occurences.

13.10 Probability Distributions

All of the following functions take one or two arguments, and return a column vector or a matrix of random numbers. For example, `normal(5)` returns a column of five random numbers, while `normal(3,6)` returns a 3×6 matrix of random numbers of random numbers of normal distribution.

Each distribution may have a few public parameters, which can be modified. For example, the `normal` function has two public parameters `mu` and `sigma` with default values `mu = 0` and `sigma = 1`.

To change the values of the public parameters, the `global` keyword must be used. E.g., `global.normal.sigma = 0.5` will change the `sigma` parameter of the system's `normal` function to `0.5`. Alternatively, you can create your own copy of the normal distribution with your chosen values of the parameters. For example,

```
rv = normal[3, 2];
```

or

```
rv = normal;
rv.mu = 3;
rv.sigma = 2;
```

will generate a normal distribution with `mu = 3` and `sigma = 2`.

The private parameters `mean`, `stddev`, and `variance` are attributes (mean, standard deviation, and variance) of the distribution. For example, `rv.stddev` returns the standard deviation of the random variable `rv`. The private parameters `pdf`, `prob`, and `quantile` are attribute functions.

- `rv.pdf(x)` gives the probability density of `rv` at `x`;
- `rv.prob(a, b)` gives the probability $P(a < X < b)$, where X is a random variable of distribution `rv`, and a and b may be $\pm\infty$;
- `rv.cdf(b)` gives the probability $P(x < b)$;
- `rv.quantile(p)` gives the value b such that $P(x < b) = p$.

List of probability distributions:

`rand`: uniform distribution.
Public parameter: `range`

`randz`: random integer.
Public parameter: `range`

`normal`: normal distribution.
public parameter: `mu` and `sigma`
private parameter: `mean`, `stddev`, `variance`, `pdf`, `prob`, `quantile`

`gauss`: Normal distribution with $\mu = 0$ and $\sigma = 1$.

`gammarv`: Gamma distribution.
public parameter: `location`, `scale`, and `shape`.
private parameter: `mean`, `stddev`, `variance`, `pdf`, `cdf`, `prob`, `quantile`

`betarv`: Beta distribution.
public parameter: `a`, `b`.
private parameter: `mean`, `stddev`, `variance`, `pdf`, `cdf`, `prob`, `quantile`

`binomial`: Binomial distribution.
public parameter: `n`, `p`.
private parameter: `mean`, `stddev`, `variance`, `pdf`, `cdf`, `prob`, `quantile`

`poisson`: Poisson distribution.
public parameter: `mu`.
private parameter: `mean`, `stddev`, `variance`, `pdf`, `cdf`, `prob`, `quantile`

`chi`: Chi-square distribution.
public parameter: `df`.
private parameter: `mean`, `stddev`, `variance`, `pdf`, `cdf`, `prob`, `quantile`

`frv`: F distribution.
public parameter: `dfn`, `dfd`.
private parameter: `mean`, `stddev`, `variance`, `pdf`, `cdf`, `prob`, `quantile`

`exprv`: Exponential distribution.
public parameter: `mu`.
private parameter: `mean`, `stddev`, `variance`, `pdf`, `cdf`, `prob`, `quantile`

`student`: Student t-distribution.
public parameter: `df`.
private parameter: `mean`, `stddev`, `variance`, `pdf`, `cdf`, `prob`, `quantile`

`hypergom`: Hypergeometric distribution.
public parameter: `N`, `D`, `n`.
private parameter: `mean`, `stddev`, `variance`, `pdf`, `cdf`, `prob`, `quantile`

`multinomial`: Multinomial distribution.
public parameter: `n`, `p`

13.11 Input and output

`print(x)`: display the contents of `x`

`readline()`: read a line from user input

`writeline(f, mesg)`: write `mesg` to file `f`

`sprintf(format, x1, x2, ...`: formatted text output, returns the string

`printf(format, x1, x2, ...)`: formatted text output, prints the string

`fprintf(file, format, x1, x2, ...)`: formatted text output, prints the string to file

`sscanf(str, format)`: formatted text input, reading from string, returns a list if more than one conversions are made

`fscanf(f, str, format)`: formatted text input, reading from file `f`

13.12 Sets

These sets are implemented as built-in functions and can be used like functions or sets. For example, `_string` is the set of all character strings, then both

```
_string("Obi-Wan Kenobe buys a used car")
```

and

```
"Obi-Wan Kenobe buys a used car" in _string
```

are true.

`_string`: the set of character strings

`_list`: the set of lists

`_hash`: the set of hash tables

`_obj`: the set of class members

13.12.1 Sets of real numbers

`_R`: the set of real numbers. Has one parameter `domain`, whose value can be an interval, a list of two numbers, or `null`. If `domain` is `null`, `_R` is the set of all real numbers. Otherwise, it is the interval specified by `domain`.

`_R2`: the set of 2D vectors of real numbers. Has one parameter `domain`.

`_R3`: the set of 3D vectors of real numbers. Has one parameter `domain`.

`_R4`: the set of 4D vectors of real numbers. Has one parameter `domain`.

_Rn: the set of nD vectors of real numbers; has a parameter `n` and a parameter `domain`.

_R22: the set of 2×2 matrices of real numbers. Has one parameter `domain`.

_R33: the set of 3×3 matrices of real numbers. Has one parameter `domain`.

_R44: the set of 4×4 matrices of real numbers. Has one parameter `domain`.

_Rnn: the set of $n \times n$ matrices of real numbers ; has a parameter `n` and a parameter `domain`.

_R21: the set of 2D column vectors of real numbers. Parameter: `domain`.

_R31: the set of 3D column vectors of real numbers. Parameter: `domain`.

_R41: the set of 4D column vectors of real numbers. Parameter: `domain`.

_Rn1: the set of nD column vectors of real numbers. Parameters: `n` and `domain`.

_R12: the set of 2D row vectors of real numbers. Parameter: `domain`.

_R13: the set of 3D row vectors of real numbers. Parameter: `domain`.

_R14: the set of 4D row vectors of real numbers. Parameter: `domain`.

_R1n: the set of nD row vectors of real numbers. Parameters: `n` and `domain`.

_Rmn: the set of $m \times n$ matrices of real numbers. Parameters: `m`, `n` and `domain`

13.12.2 Sets of integers

_Z: the set of machine integers. Parameter: `domain`.

_Z2: the set of 2D vectors of machine integers. Parameter: `domain`.

_Z3: the set of 3D vectors of machine integers. Parameter: `domain`.

_Z4: the set of 4D vectors of machine integers. Parameter: `domain`.

_Zn: the set of nD vectors of machine integers; has a parameter `n`

_Z22: the set of 2×2 matrices of machine integers. Parameter: `domain`.

_Z33: the set of 3×3 matrices of machine integers. Parameter: `domain`.

_Z44: the set of 4×4 matrices of machine integers. Parameter: `domain`.

_Znn: the set of $n \times n$ matrices of machine integers. Parameter: `n` and `domain`.

_Z21: the set of 2D column vectors of machine integers. Parameter: `domain`.

_Z31: the set of 3D column vectors of machine integers. Parameter: `domain`.

_Z41: the set of 4D column vectors of machine integers. Parameter: `domain`.

_Zn1: the set of nD column vectors of machine integers. Parameter: `n` and `domain`.

_Z12: the set of 2D row vectors of machine integers. Parameter: `domain`.

_Z13: the set of 3D row vectors of machine integers. Parameter: `domain`.

_Z14: the set of 4D row vectors of machine integers. Parameter: `domain`.

_Z1n: the set of nD row vectors of machine integers. Parameter: `n` and `domain`.

_Zmn: the set of $m \times n$ matrices of machine integers. Parameter: `m`, `n`, and `domain`.

13.12.3 Sets of complex numbers

_C: the set of complex numbers

_C2: the set of 2D vectors of complex numbers

_C3: the set of 3D vectors of complex numbers

_C4: the set of 4D vectors of complex numbers

_Cn: the set of nD vectors of complex numbers; has a parameter `n`

_C22: the set of 2×2 matCices of complex numbers

_C33: the set of 3×3 matCices of complex numbers

_C44: the set of 4×4 matCices of complex numbers

_Cnn: the set of $n \times n$ matCices of complex numbers ; has a parameter `n`

_C21: the set of 2D column vectors of complex numbers

_C31: the set of 3D column vectors of complex numbers

_C41: the set of 4D column vectors of complex numbers

_Cn1: the set of nD column vectors of complex numbers; has a parameter `n`

_C12: the set of 2D row vectors of complex numbers

_C13: the set of 3D row vectors of complex numbers

_C14: the set of 4D row vectors of complex numbers

_C1n: the set of nD row vectors of complex numbers; has a parameter `n`

_Cmn: the set of $m \times n$ matCices of complex numbers; has two parameters `m` and `n`

13.12.4 Sets of double precision floating point numbers

_D: the set of doubles

_D2: the set of 2D vectors of doubles

_D3: the set of 3D vectors of doubles

_D4: the set of 4D vectors of doubles

_Dn: the set of nD vectors of doubles; has a parameter n

_D22: the set of 2×2 matrices of doubles

_D33: the set of 3×3 matrices of doubles

_D44: the set of 4×4 matrices of doubles

_Dnn: the set of $n \times n$ matrices of doubles ; has a parameter n

_D21: the set of 2D column vectors of doubles

_D31: the set of 3D column vectors of doubles

_D41: the set of 4D column vectors of doubles

_Dn1: the set of nD column vectors of doubles; has a parameter n

_D12: the set of 2D row vectors of doubles

_D13: the set of 3D row vectors of doubles

_D14: the set of 4D row vectors of doubles

_D1n: the set of nD row vectors of doubles; has a parameter n

_Dmn: the set of $m \times n$ matrices of doubles; has two parameters m and n

13.12.5 Sets of machine integers

_I: the set of integers

_I2: the set of 2D vectors of integers

_I3: the set of 3D vectors of integers

_I4: the set of 4D vectors of integers

_In: the set of nD vectors of integers; has a parameter n

_I22: the set of 2×2 matrices of integers

_I33: the set of 3×3 matrices of integers

_I44: the set of 4×4 matrices of integers

_Inn: the set of $n \times n$ matrices of integers ; has a parameter n

_I21: the set of 2D column vectors of integers

_I31: the set of 3D column vectors of integers

_I41: the set of 4D column vectors of integers

_In1: the set of nD column vectors of integers; has a parameter n

_I12: the set of 2D row vectors of integers

_I13: the set of 3D row vectors of integers

_I14: the set of 4D row vectors of integers

_I1n: the set of nD row vectors of integers; has a parameter n

_Imn: the set of $m \times n$ matrices of integers; has two parameters m and n

13.12.6 Sets of byte integers

_B: the set of bytes

_B2: the set of 2D vectors of bytes

_B3: the set of 3D vectors of bytes

_B4: the set of 4D vectors of bytes

_Bn: the set of nD vectors of bytes; has a parameter n

_B22: the set of 2×2 matrices of bytes

_B33: the set of 3×3 matrices of bytes

_B44: the set of 4×4 matrices of bytes

_Bnn: the set of $n \times n$ matrices of bytes ; has a parameter n

_B21: the set of 2D column vectors of bytes

_B31: the set of 3D column vectors of bytes

_B41: the set of 4D column vectors of bytes

_Bn1: the set of nD column vectors of bytes; has a parameter n

_B12: the set of 2D row vectors of bytes

_B13: the set of 3D row vectors of bytes

_B14: the set of 4D row vectors of bytes

_B1n: the set of nD row vectors of bytes; has a parameter n

_Bmn: the set of $m \times n$ matrices of bytes; has two parameters m and n

13.12.7 Sets of complex numbers

_DC: the set of complex numbers (no nan/inf)

_DC2: the set of 2D vectors of complex numbers (no nan/inf)

_DC3: the set of 3D vectors of complex numbers (no nan/inf)

_DC4: the set of 4D vectors of complex numbers (no nan/inf)

_DCn: the set of nD vectors of complex numbers (no nan/inf); has a parameter `n`

_DC22: the set of 2×2 matrices of complex numbers (no nan/inf)

_DC33: the set of 3×3 matrices of complex numbers (no nan/inf)

_DC44: the set of 4×4 matrices of complex numbers (no nan/inf)

_DCnn: the set of $n \times n$ matrices of complex numbers (no nan/inf) ; has a parameter `n`

_DC21: the set of 2D column vectors of complex numbers (no nan/inf)

_DC31: the set of 3D column vectors of complex numbers (no nan/inf)

_DC41: the set of 4D column vectors of complex numbers (no nan/inf)

_DCn1: the set of nD column vectors of complex numbers (no nan/inf); has a parameter `n`

_DC12: the set of 2D row vectors of complex numbers (no nan/inf)

_DC13: the set of 3D row vectors of complex numbers (no nan/inf)

_DC14: the set of 4D row vectors of complex numbers (no nan/inf)

_DC1n: the set of nD row vectors of complex numbers (no nan/inf); has a parameter `n`

_DCmn: the set of $m \times n$ matrices of complex numbers (no nan/inf); has two parameters `m` and `n`

13.12.8 Sets of other numbers

_L: the set of all arbitrary size integers

_M: the set of all arbitrary precision floating point numbers

13.13 Miscellaneous

`exist(x)`: returns 0 if `x` is `DNE`, 1 otherwise.

`to_set(x)`: convert value (vector, matrix) to a set.

`hashcode(x)`: returns the hash code of value `x`

`trivial(x)`: has one parameter `p`; returns `p` regardless the value of `x`

Chapter 14

System Commands

A few system commands can be used within the interpreter. They are like functions but when they are called the arguments don't need to be included in brackets.

`sysem`: execute an operating system command

```
>> system "cmd.exe";
```

The command prompt window will pop up.

`pwd`: show current working directory. Takes no argument

```
>> pwd;
   G:\xlab07\windows\
```

`cd`: change current working directory. Takes one argument, which is the directory to be changed to (a quoted string).

```
>> cd "..";
>> cd "secrets";
```

In Windows, absolute path starts with the drive letter followed by a colon. To change current drive to `D`, do

```
>> cd "D:";
```

`ls`: list files in current working directory. Works the same way as the shell command `ls`, but it doesn't take arguments.

`clear`: clear the screen.

`cat`: print contents of a file.

```
>> cat "boot.ini";
```

The contents of file `boot.ini` will be printed out in the command window.

`exit`: close the interpreter program. Takes no argument.

`help`: displays a help message on a given topic. Takes one argument (the help subject).

```
>> help "while"
```