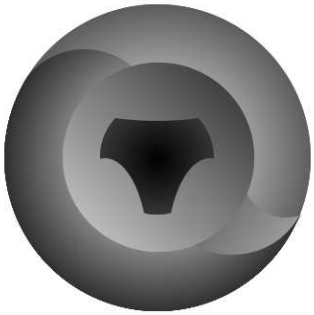

Tao Language
for
Scripting and Computing

An Introduction to Tao Language

(draft)

Li-Min Fu

Italy, 2005

Preface

What is Tao Language?

Tao is an object-oriented scripting language with dynamic-typing variables supporting complex data structures. It has powerful text processing ability such as string regular expression matching. It provides built-in numerical data types such as complex number and multi-dimensional numeric array, and their corresponding operations and computations are very convenient in Tao. It can be easily extended with C++, through a simple and transparent interface.

How it came out?

I started to designed and implemented Tao Language from the beginning of May 2004. Before that time, I spent a few weeks studying Gene Ontology (GO), and wrote a small graphical software to display Directed Acyclic Graph (DAG) of GO. Then I found it would not be convenient to do costumized computation based on GO DAG with my graphical software, because the software must be rebuilt each time there would be any changes in the computation method, which can happen frequently when the method is not stabilized. What's more, it's not trivial to make a convenient interface for a complicated computational task. So I realized the goodness of scripting languages, and spent about two weeks to write some Perl scripts to retrieve information from GO and construct the DAG. But that experience with Perl was not very nice, because of its complicated syntax.

Then I started to think about the possiblity to design a new language with simple syntax, and formed some basic ideas. I searched through Google trying to find something about how to write an interpreter, but there were few things which were practically useful for me. But I started write some lines of codes anyway to give it a try just for curiosity to see if those ideas would work. Within two weeks, after two times rewriting (almost completely) the codes, it started to work well for simple arithmetic calculation, and it was even much fast than Perl for such calculation (unfortunately, when the interpreter becomes more complicated, its efficiency drops down, but it is still comparable with Perl; but if Tao numeric types are properly used, it can be much faster than Perl). I was encouraged by it very much, and decided to go ahead, and spent many evenings and weekends to improve it. Now it becomes what you will see in this document.

About the name and log?

Tao comes from a Chinese word, it is the same "Tao" in Chinese philosophy and religion Taoism. Its basic meaning is "path, way", but it also means the ultimate principle of the universe in Chinese culture and philosophy. It's believed that such principle must show extreme simplicity. And simplicity is what this language and its interpreter intended to have.

The log was also designed by myself using GIMP software in Linux. It includes three layers, the upper layer is in the center, resembling 'T'. The middle layer is a circle with a tail, resembling 'a'. And the bottom layer is a "Taiji" pattern which connects this Tao to that "Tao" in Chinese culture. With its central part covered, it forms an 'o'.

Contents

Chapter 1

Introduction

Tao is a language with very simple syntax compared with other scripting languages such as Perl, Python, while still very expressive. It provides some commonly used data structure such as list/array, hash/dictionary, which can be manipulated in a very straightforward way.

Text processing, as an indispensable feature of a scripting language, is well supported in Tao. Like Perl, Tao has direct syntax support for string regular expression patterns, which are similar to that in Perl with a few changes.

Unlike those popularly used scripting languages, Tao has built-in numerical data types such as complex number and multi-dimensional numeric array. And their corresponding operations and computations are very convenient in Tao.

It can be easily extended with C++, through a simple and transparent interface.

To run Tao script, in command console do,

[linux]$ ./tao source.tao

[windows]$ tao.exe source.tao

Here it is supposed that the executable is named as *tao* (or *tao.exe*) and is put in the save directory of *source.tao*.

Chapter 2

Basic Tao Data Types

All Tao variables are internally represented as references to objects. So Tao variables don't have fixed types, which can be changed during running time. In any time point, an Tao variable can be a number, a string, an object, a hash, an array of any type, a numeric array, a routine or a namespace - almost anything in Tao.

By default, most operations such as assignment, appending, inserting and erasing array elements are done based references with apropriate dereferencing.

Variable names can be specified with keyword **const**, **local**, **extern** or **share**. A **const**-specified variable is a constant, normally its value can not be changed. But, in special case, its value is allowed the change. The simplest case is, using keyword **const** again to re-initialize its value.

```
const it="old";
it="new"; # Wrong. A warning will arise.
          # No assignment will be done. "it" is still "old".
const it="new"; # Right. Now "it" is "new".
```

local can be used to "localize" a variable, which has no effect to the outside of its scope which is valid only until the next unpaired "}".

```
a=100;
{
```

```
   local a="local";
}
print(a); # It prints "100", instead of "local".
```

extern is used to specify an external variable, which should be available at compiling time from the input namespace. The keyword **share** in Tao is very much like **common** in Fortran77 (If I remember correctly). Two or more variables, declared in different places (connected by namespace during compiling) with the same name and specified with **share**, refer to the same data. See chapter 11.

In the above examples, # means starting a comment until the end of a line. To comment multiple lines, use <<< and >>> pair. Note that <<< and >>> are interperated as the starting and ending of comment **only if** they are put in the beginning of a line.

In Tao, variables are declared directly by assignment,

```
a = 100;
b = "A string";
c = { a, b }; # A list
```

2.1 String

In Tao, the operations of string is very convenient and almost all of them are done by operators with subindexing. In addition to the normal subindex, string can have subindex in form of $string[from : to]$, depending its context, it can either return a substring of characters with indices from index $from$ to index to, or change the characters between these two indices.

```
a = "0123456789";

# get substring by subindexing:
b = a[1:5];
# b="12345";

# conversions by arithmetic operations:
c = 0 + a[8:9];
```

```
# c=89;
d = "" + 12 + 34;
# d="1234";

# comparison:
e = d < a;

# insert, replace and erase by assignment:

# replace:
a[2:5] = "abcdefgh";
# a="01abcdefgh6789"

# insert, if the second index is smaller than the first.
# insert before the first index:
a[4:3] = "INSERT";
# a="01abINSERTcdefgh6789"

# erase:
a[5:10]="";
# a="01abIdefgh6789";

# length:
print( a.#, "\n" );
# print: 14
```

There are three built-in functions related to string operations including **number()**, **pack()** and **unpack()**:

number(*string/array*, *base* **):** convert string(s) to number(s). The first parameter must be a string or an array of strings, and the second is the base by which the strings are interpreted. The default base is 10.

```
a = { ".1", "a" };
print( number(a,16),"\n" );
# print: { 0.0625, 10 }
```

pack(*number/array* **):** convert number(s) to character(s) which are packed as a string. The first parameter must be a number or an array of numbers.

```
a = pack(55); # a="2".
b = pack({77,78,88,66,67,55}); # b="MNXBC7".
```

unpack(*string* **):** unpack a string to characters and convert them to numbers.

```
a = unpack("adfAfga");
# a={ 97, 100, 102, 65, 102, 103, 97 }
```

2.2 Array

An array is a list of objects. It should be noted that the index of Tao arrays starts from 0 to the array size minus 1, like in C/C++.

Creation of Array

In Tao, there is several different way to create an array. The simplest is by enumeration,

```
array = { 1, 2, 3, 4 };
```

It can also be created by specifying a range,

```
array = { 1 : 1: 4 };
```

The syntax for this is,

```
array = { init_value : incremental_value : element_number };
```

This is also true for array of strings or complex numbers. For example,

```
array = { "A" : "BC" : 4 };
```

will give a string "ABCBCBC".

An array can also hold objects of different types,

```
array = { 1, "second", 3 };
```

In Tao, arrays can be created in another convenient way, like

```
array = {10} : "apple";
```

this will create an array with 10 "apple"s. The syntax for this is,

```
array = { size_specifier } : initialization_element;
```

In this way, one can put multiple (arbitrary number of) size specifiers to create multi-dimensional arrays,

```
array = {5}{10} : "apple";
```

this gives an array of 5 elements, each of these elements is an array of 10 "apple"s.

Array Size

The size of array can be got by using right operator **.#**

```
apples = {10} : "apple";
print( apples.# ); # print 10.
```

Array Methods

Two methods **insert()** and **erase()** are provided to modify an array. Other array operations can be done by flexible subindexing.

insert() takes two parameters, the first is the object to be inserted, the second is the index where it will be insert. The default value for the second parameter is the size of the array. So *array.insert(e)* will simply append *e* to *array*.

erase() also takes two parameters, the first is the starting index and the second is the ending index of the elements to be erased. The default value

for the first is 0, for the second is the array size minus 1. *array.erase()* will simply erase all elements in *array*.

Another method **flat(** *array* **)** is provided to get a flated array of *array*. The elements and their order in the flated array are the result of the following process, consider the *array* as a tree where arrays are nodes and any other type of objects are leaves, then a deep-first traversal is performed on the tree and the leaves are inserted into the flated array when they are visited. This method is useful in the conversion between normal arrays and numeric arrays.

```
array = {"one",{"two",10,{1,2},"three"},[1,2]};
flated = array.flat();
# flated={"one","two",10,1,2,"three",[1,2]}
```

Array Subindexing

Array operations other than **insert** and **erase** can be done by flexible subindexing:

```
array10 = {10} : "apple";

index = [ 1, 3, 5, 7, 9 ]; # numeric vector

# sub arrays of "array10":
array5A = array10[ 0 : 5 ]; # get the first 5 apples.
array5B = array10[ index ]; # get apples with odd index.

# assignment:
array10[ 9 ] = "kiwi";
# replace the last apple with kiwi.

array10[ 0 : 5 ] = "orange";
# replace the first 5 with oranges.

array10[ index ] = "peach";
# replace the odd index apples with peach.
```

in the above example, subindex [$id1$: $id2$] indicates all indices from index $id1$ to $id2$. If $id1$ is not specified, 0 is used; and if $id2$ is not specified, the last index in the array is used.

Sort Array

The sorting of array is very simple, just do like this,

```
a = { 4, 3, 5, 2, 6, 1 };
sort( a, @1<@2 );
# @1,@2 indicate two successive elements in the array.
# @1<@2 means to sort the array ascendantly.
# a={1,2,3,4,5,6}

a = { 4, 3, 5, 2, 6, 1 };
sort( a, @1<@2, 3 );
# The last parameter 3 indicate to sort until the
# largest 3 elements are properly sorted.
# a={2,3,1,4,5,6}

a = { 4, 3, 5, 2, 6, 1 };
b = sort( a[2:], @1<@2 );
# Sort from the third element and return a new array.
# b={1,2,5,6}

array = {{3,"A"},{4,"M"},{2,"E"},{6,"K"},{2,"C"},{5,"H"}};
# "array" is an array of array which contain a number
# and a string. We can sort it by the numbers or by
# by the strings.

# Sort by the numbers:
sort( array, @1[0]>@2[0] );
# array={{6,"K"},{5,"H"},{4,"M"},{3,"A"},{2,"E"},{2,"C"}}

# Sort by the strings:
sort( array, @1[1]<@2[1] );
# array={{3,"A"},{2,"C"},{2,"E"},{5,"H"},{6,"K"},{4,"M"}}
```

sort(*array*, *exprs*, *m* **):** sort *array* by *exprs*. The heapsort algorithm is used. The sorting is in place, during sorting, the sorted elements are stored in the last part of the array. If *m* is present, sort *array* until the last *m* elements are replaced with sorted values. *exprs* must contain two transient variables @1 and @2 (See Chapter 8 for transient variables), during sorting, two elements are passed to @1, @2, and the comparison is done by *exprs*.

2.3 Hash

Hash is a set of key/value pairs. Usually its elements are accessed by the keys.

Hash can also be created by enumeration,

```
hash = { key1 : value1, key2 : value2 ... };
```

here the keys must be strings, while values can be anything. If $key1$ happened to be an array of strings and in the same time $value1$ happened to be an array, each element from $key1$ and the element from $value1$ with the same index will form a $key/value$ pair which is inserted into the $hash$. If these two arrays are of different size, the extra elements are ignored.

The assignment $hash[key] = value$ will insert a pair of $key/value$ into the hash, if this key is not presented in the $hash$; otherwise, the value corresponding to key is replaced with $value$. So one can also create a hash in this way,

```
hash = { : }; # empty hash.
hash[ key1 ] = value1; # insert key1/value1 pair
hash[ key2 ] = value2; # insert key2/value2 pair
```

The keys of a hash can be extracted by right operator .%, and the values can be extracted by right operator .@. A hash can also have paired subindices specifying a range of keys. For example, $hash[key1 : key2]$ specifies a subhash with key starting from $key1$ and ending at $key2$, if $key1$ is not presented in the hash, the starting key is the smallest key larger than $key1$; if $key2$ is not presented in the hash, the ending key is the largest key smaller than $key2$.

Examples:

```
nv = { 1, 2, 3 };
hash1 = { "A":1, "C":"name", "B":nv };

hash2={:};
hash2["a"]=nv;
```

```
hash2["b"]=hash1;

# Array of hashes:
hash3 = {10} : {:};
hash3[0]["a"]=nv;

# Get keys:
keys=hash1.%;
# Get values:
values=hash2.@;

keys = {"AA","BB","cc","DD"};
values = {11,22,33,44,55};
hash = { keys : values };

print( hash, "\n" );
# print: { "AA"=>11, "BB"=>22, "DD"=>44, "cc"=>33 }
print( hash["AA":"D"], "\n" );
# print: { "AA"=>11, "BB"=>22 }
print( hash["B":], "\n" );
# print: { "BB"=>22, "DD"=>44, "cc"=>33 }
```

2.4 Numeric Types

To support powerful numeric computation, Tao provides built-in numeric types such as complex number and multi-dimensional numeric array. See Chapter 7.

Chapter 3

Basic Tao Operaters

Tao Language supports a set of abundant operators to facilitate the writing of more expressive scripts. Many of these operators are versatile and can be used for different data types.

Arithmetic:

+	addition
−	substraction
∗	multiplication
/	division
%	mod
∧	power
++	prefix and postfix increment (are the same in Tao)
−−	prefix and postfix decrement (are the same in Tao)

Numeric and String Comparison:

==	equal
! =	not equal
<	less than
>	greater than
<=	less than or equal
>=	greater than of equal

Boolean Logic:

&&	and
||	or
!	not

Assignment and Composite Assignment:

Assignment: $=,\ :=,\ +=,\ -=,\ *=,\ /=,\ *=,\ \%=,\ \wedge=,\ \&=,\ |=$

$=$, $:=$ are for all data type, and $=$ can only be used in a stand alone assignment statement; to do assignment within another expression one must use $:=$ instead of $=$. This is to avoid the confusion between $=$ and $==$, each of which can happen to be a typo of another inducing a bug. The others are mainly for numeric data types (numbers, numeric arrays), but $+=$ can also be used for string.

Regular Expression Matching and Splitting:

$=\sim$	pattern matching, usage: $str =\sim /expr/$ it returns boolean result in boolean or arithmetic expressions, such as in $if(str =\sim /expr/)$; it returns the matched substring (if there is) in assignment, such as in $var = (str =\sim /expr/)$.
$!\sim$	pattern NOT matching, usage $str!\sim /expr/$ it returns boolean result in boolean or arithmetic expressions, such as in $if(str!\sim /expr/)$; it returns the unmatched substring (if there is) in assignment, such as in $var = (str!\sim /expr/)$.
$\sim\sim$	only used in $var = str \sim\sim /expr/$, extracts sub-strings matching or NOT matching *expr*;

See Chapter 6 for more details.

Right Operators:

.#	get string length or array size
.@	get hash values
.%	get hash keys
.?	get variable information

Numerical Operators:

See Chapter 7.

Miscellaneous:

+	string concatenation
.	function call by an object
=?	binary type comparison operator, check if two operands have the same type
!?	binary type comparison operator, check if two operands have different type
$e_0?e_1 : e_2$	if evaluation of e_0 gives TRUE, e_1 is evaluated and returned, otherwise, e_2 is evaluated and returned.

Chapter 4

Logic and Loop Controls

Currently on **if**, **else if**, **else**, **while**, **for**, **foreach**, **break** and **skip** are implemented.

If Else Controls

```
syntax:
if(expr1){
   block1;
}else if(expr2){
   block2;
}else{
   block3;
}
```

If *expr*1 is true, *block*1 is executed; otherwise, if *expr*2 is true, *block*2 is executed; otherwise, *block*3 is executed;

While Control

```
syntax:
while(expr){
   block;
}
```

If *expr* is true, *block* is executed and repeated until *expr* becomes false.

For Control

```
syntax:
for( init; condition; step ){
   block;
}
```

The execution sequence of **for** statement is *init* → *condition* → *block* → *step* → *condition* → *block* → *step* → ..., if *condition* is always true. Once *condition* become false, the loop is exited.

Foreach Control

```
syntax:
foreach(array:element){
   block;
}
```

For each *element* in *array*, *block* is executed.

Other Controls

break can be used to exit a loop, and **skip** can be used to skip the rest part of script and start the next cycle of a loop. **skip** is equivalent to **continue**.

Chapter 5

File IO

So far Tao only supports basic handling of files and IO. It can only perform a few simple standard IO operations such as read from and write to STD and files. In Tao, a file IO stream is a constant object created as a return value of function "open()". The first parameter of "open()" is the file path and name, the second is the attribute of the stream - so far only two is supported, "w" for write only stream and "r" for read only stream.

Two functions for IO are provided, "print()" and "read()". "read()" reads a whole line with the new line symbol choped. See comments in the following sample.

```
# Open a file for writing:
fout=open("test1.txt","w");

# Write to the file:
fout.print("log(10)=",log(10));

# Open a file for reading:
fin=open("test2.txt","r");

# Read from the file:
line=fin.read();
fin.read(line);

# while reading succeeds:
while( fin.read(line) ){
   # Write to std out:
```

```
   print(line,"\n");
}

# Read from std in:
d=read();
print(d,"\n");
```

Chapter 6

String Regular Expressions in Tao

In Tao language the only thing similar to Perl is the string regular expressions (*Regex*). Like Perl, Tao also has direct syntax support for regular expressions, and provides convenient *Regex* matching functionalities.

A regular expression is a string representing a pattern (rules), from which a set of strings can be constructed. The pattern represented by a regular expression is used to search in a string for sub-strings, which can be constructed from that pattern, namely sub-strings that match that pattern. A number of operations can be performed on the resulting sub-strings, including extraction, replacing and splitting etc.

There are three basic operator for regular expression matching: $=\sim$, $!\sim$ and $\sim\sim$. In this chapter, for convenience, when **regular expression** or *Regex* is used, it can mean the representation string itself or this string together with the matching operators. It should be easy to infer which one it actully means.

	Regex operoters
$=\sim$	pattern matching, usage: $str =\sim /expr/$ it returns boolean result in boolean or arithmetic expressions, such as in $if(str =\sim /expr/)$; it returns the matched substring (if there is) in assignment, such as in $var = (str =\sim /expr/)$.
$!\sim$	pattern NOT matching, usage $str!\sim /expr/$ it returns boolean result in boolean or arithmetic expressions, such as in $if(str!\sim /expr/)$; it returns the unmatched substring (if there is) in assignment, such as in $var = (str!\sim /expr/)$.
$\sim\sim$	only used in $var = str \sim\sim /expr/$, extracts sub-strings matching or NOT matching *expr*;

In all these case, the Tao interpreter first tries to find a matched substring. If found, in boolean and arithmetic expression, $=\sim$ will return true, while $!\sim$ will return false. But in assignment, $=\sim$ will return the matched substring, whereas $!\sim$ will return the substring located between the previously matched substring (or the begin of the string) and the current mached substring (or the end of the string); while $\sim\sim$ will return the matched and unmatch substring alternately. See section 6.6

```
a = "This is a string for the string regex example.";

if( a=~/string/ ){
   print("matched\n");
}

while( a=~/string/ ){
   print("matched\n"); # print twice
}

b = a !~ /string/;

print(b,"\n");
# print: This is a
```

6.1 Simple Word Matching

The simplest regular expression is a string not containing metacharacters such as

$$\{\}[\ ]()\ \wedge @.*+?\backslash$$

which are reserved for special usage. However they can be used together with '\' such as "\\" to match "\", "\{" to match "{" and "\@" to match "@" etc.

Examples:

/word/: to match any string containing "word"
/word\ /: to match any string containing "word "

6.2 Matching Character Classes

A character class is a set of possible characters that match a single character in a particular position. Usually, a character class is represented by brakets [...] containing the possible characters. If there is $\wedge$ after [, it means matching characters which are not in this class.

Example:

/[tT]ao/: to match strings containing "tao" or "Tao"
/[$\wedge$ABC]/: to match any strings not containing "A", "B" or "C"

One may also use a range of characters described by a starting character, a range operator $-$ and a ending character.

Examples:

/x[0-9]/: to match "x0", or "x1", ..., or "x9"
/[a-z]0/: to match "a0", or "b0", ..., or "z0"

One may also use abbrievations such as,

$\backslash d$	digits: [0-9]
$\backslash s$	whitespaces: [\ $\backslash t \backslash n$]
$\backslash c$	lower case letters: [a-z]
$\backslash w$	word characters: [0-9a-zA-Z_]
$\backslash D$	negative to $\backslash d$: [$\wedge 0-9$]
$\backslash S$	negative to $\backslash s$: [$\wedge \backslash s$]
$\backslash C$	upper case letters: [A-Z]
$\backslash W$	negative to $\backslash w$: [$\wedge \backslash w$]

Examples:

```
a = "ABC123abcDEF456GHI";

b = a=~/[0-3\cA-C]+/;
# b="ABC123abc";

c = a=~/[^0-3\cA-C]+/;
# c="DEF456GHI";

d = a=~/\D+/;
# d="ABC";
```

6.3 Repeat Matching

Quantifier metacharacters can be used to specify how many times a portion of a regular expression must be matched.

$e?$	match e 0 or times
$e*$	match e any number of times
$e+$	match e at least once
$e\{m\}$	match e exactly m times
$e\{m,\}$	match e at least m times
$e\{m,n\}$	match e at least m times and at most n times

```
a = "abc12defg3345ga4ga";
b = repeat( a=~/\c+\d\d?\c*/) ); # see section 6.6
# b={ "abc12defg", "ga4ga", "da3" };
```

```
a = "12ab1234ab123456ab";
b = a=~/\d{2}/;
c = a=~/\d{3,5}/;
d = a=~/\d{5,}/;
# b="12", c="1234", d="123456"
```

6.4 Embedding Expressions in Regular Expression

Sometimes it would be necessary match a string to the results of another expression, this can be done by embedding that expressions in a regular expression by inserting @{*exprs*}. For example,

```
a = "This is a string for the example.";
c = "string";
d = a =~ /@{c+" for"}/;

print( d, "\n");
# print: string for

e = a =~ /@{ "is a " + (d := ( a =~ /@{c+" for"}/) ) }/;

print( e, "\n");
# print: is a string for
```

6.5 Matching with Alternate and Groups

In the previous sections, only one possiblity is allowed to match in one position, but this is not adequate in some situation. Then how do we provide alternates in one position, the solution is to create a *Regex* grouping, and put the alternates in the grouping and separate them by |.

```
a = "black cat 0001";
b = "white dog 0002";
```

```
c = a=~/(cat|dog)/; # c="cat"
d = b=~/(cat|dog)/; # d="dog"

e = "black bat 0003";

f = a=~/((b|c)at|dog)\ \d+/; # f="cat 0001"
g = b=~/((b|c)at|dog)\ \d+/; # g="dog 0002"
h = e=~/((b|c)at|dog)\ \d+/; # h="bat 0003"
```

Like in Perl, *Regex* grouping can also be used to extract parts of the matched substring. For each grouping, the matched substring can be extracted and stored in a special variable − transient variable (See Chapter 8) as called in Tao. Unlike in Perl, where the extraction is done automatically for all groupings, and one must explicitly prevent them if they are to be avoided, in Tao, the matched substring for grouping is extract only if the transient variable is explicitly specified by format $/(@var : regex)/$.

```
a = "====this is a string for the example======";
b = a=~/(@1:\w|\s)*(@2:string)(@3:\w|\s)*/;

# b="this is a string for the example";
# @1="this is a ";
# @2="string";
# @3=" for the example";
```

However, the Tao transient varibales are valid only in one statement, as a result, one can't use $@1, @2, @3$ directly in its following statestment. So one must assign $@1, @2, @3$ to other variables in the same statement of the *Regex*. This can be done by appending an arithmetical expression in the end of *Regex* as $/regex/arith\_exprs/$, where $arith\_exprs$ will be evaluated after the *Regex* matching is performed.

```
b = a=~/(@1:\w|\s)*(@2:string)(@3:\w|\s)*/d:={@1,@2,@3}/;

print(b,"\n");
# print: this is a string for the example

print(d,"\n");
# print: { "this is a ", "string", " for the example" }
```

By embedding arithmetical expression of such transient variables, strings with repeated patterns can be easily matched. For example, /(@1 : \c+)@{@1}/ can be used to match "mama", "papa", or "coco" etc.

6.6 String Splitting and Replacing

In Tao there is no special functions for string splitting and replacing by *Regex*, instead, a combination of a generic function and the *Regex* operators can do the work well. That generic function is **repeat(** *exprs* **)**, which repeats evaluating *exprs* as long as certain condition is satisfied and return all the results of *exprs* as an array. The condition for repeating varies for difference type of expression (See Chapter8). For a regular expression the condition is the success of finding a substring which satisfies that regular expression.

```
string = "ABC123DEF456GHI";

# Extract substrings of digits:
sv1 = repeat( string=~/\d+/ );

# Split the string with digits as the separator:
sv2 = repeat( string!~/\d+/ );

# Extract all substrings of digits and non-digits:
sv3 = repeat( string~~/\d+/ );

print( sv1, "\n", sv2, "\n", sv3, "\n" );
# print:
{ "123", "456" }
{ "ABC", "DEF", "GHI" }
{ "ABC", "123", "DEF", "456", "GHI" }
```

If a *Regex* operator modifier "**s**: is put after the operator, and an arithmetical expression is appended in the end of *Regex* as */regex/arith_exprs/*, the matched (for $=\sim$) or the un-matched (for $!\sim$) substring is replaced with the result of *arith_exprs*, which must return a string as its result.

```
a = "There is a cat";
b = a=~s/cat/"dog"/;
```

```
print( b, "\n", a, "\n" );
# print:
cat
There is a dog
```

repeat() can also used to replace all satisfied substrings,

```
a = "ABC123DEF456GHI";
b = repeat( a=~s/(@1:\d+)/""+(1000+@1)/);
# b={ "123", "456" };
# a="ABC1123DEF1456GHI";
```

For convenience, another modifier "**gs**" is provided to do the same work without using **repeat()**,

```
a = "ABC123DEF456GHI";
a =~gs/(@1:\d+)/""+(1000+@1)/;
# a="ABC1123DEF1456GHI";
```

Another example to finish this chapter,

```
a="12123421631934563216641994521534";
b = repeat( a=~/ (@1: \d+) @{ @1=~gs/6/9/ } / );
# b={ "1212", "163193", "16641994" }
```

In this example $/(@1 : \backslash d+)@@1 = \ gs/6/9/$ will match digits in which, the last half is the same as the first half if there is no '6' in the first half; if there is '6'(s) in the first half, the last half should have '9'(s) in the corresponding place of '6'(s). eg: 1212, 163193, 16641994.

Chapter 7

Numeric Computation in Tao

To support powerful numeric computation, Tao provides built-in numeric types such as complex number and multi-dimensional numeric array.

7.1 Complex Number

The imaginary part of a complex number is represented by symbol $.

```
a = 1+$; # a=1+i;
```

asic complex arithmetic calculations such as addition (using operator "+"), substraction ("-"), division ("/") and multiplication ("*") are also supported.

7.2 Numeric Array

Numeric arrays can be created in the same way as normal array, using [ ] where **{ }** are used.

```
# Numeric vectors, single row matrices, by enumeration:
a = [ 1, 2, 3, 4 ];

# By specifying a range:
```

```
b = [ 1 : 2 : 10 ];

# Complex vector:
c = [ $, 2, 3+4$ ];

# Enumerate 2 x 3 matrix:
d = [ 1, 2, 3; 4, 5, 6];
```

In numeric vector enumerations, if an element in the enumeration is a numeric array, the elements of this array will be inserted into the new numeric vector; if an element in the enumeration is a normal array, its elements are inserted to the numeric vector with appropriate conversion, in fact, this array is expanded in the enumeration.

```
a = [2][2] : 1;
b = [ 10, a ]; # b=[ 10, 1, 1, 1, 1 ]

c = { 1, [ 2 , 3 ], a };
d = [ 100, c ];
# equivalent to expand "c" in the enumeration, i.e.
# d=[100,1,[2,3],a];
# so d=[100,1,2,3,10,1,1,1,1].
```

Numeric arrays can also be created with syntax:

```
numarray = [ N1 ][ N2 ] ... [ Nk ] : init_element;
```

here *init_element* can be a scalar or another numeric array. And *numarray* will be a $N1 \times N2 \times ... \times Nk$ dimensional array with all element to be *init_element*.

```
array1 = [ 4 ] : 0; # array1=[0,0,0,0].

array1 = [ 1, 2, 3, 4];
array2 = [3] : array1; # 3x4 array.
array3 = [2] : array2; # 2x3x4 array

print(array2);
# it will print:
```

```
row(0,:):       1       2       3       4
row(1,:):       1       2       3       4
row(2,:):       1       2       3       4

print(array3);
# it will print:
row(0,0,:):     1       2       3       4
row(0,1,:):     1       2       3       4
row(0,2,:):     1       2       3       4
row(1,0,:):     1       2       3       4
row(1,1,:):     1       2       3       4
row(1,2,:):     1       2       3       4
```

The structure of Tao numeric array is such: a matrix of $N1 \times N2$ is composed of $N1$ row vectors of size $N2$; an array of $N1 \times N2 \times N3$ is composed of $N1$ matrices of size $N2 \times N3$, and so on.

Some people may not used to this kind of multi-dimensional numeric array creation, for them, there is another option, that is the using of Tao internal function **numarray()** to create numeric array on fly. See Section 7.6.

7.3 Operators for Numeric Array

left operand	right operand	suported operator
scalar	array	$+, -, *$
array	scalar	$+, -, *, /$
		$=, + =, - =, * =, / =$
only for:		
real array	real number	$\%, \wedge, \&\&, \|\|$
real array	real number	$\% =, \wedge =, \& =, \| =$
array	array	$+, -, *, + =, - =, < * >, < / >$
		$+ =, - =, * =, / =$ pairwise
only for:		
real array	real array	$\% =, \wedge =, \& =, \| =$ pairwise
unary operations		
array		$++, --$
	array	$++, --, '$ $(transpose)$

Note: $< * >, < / >$ are pairwise element operators.

Examples:

```
a = [ 1, 2, 3 ] + 10;
print( a, "\n" );
# print:
[ 11, 12, 13 ]

b = [ 3, 2, 1 ] * [ 1, 2, 3; 4, 5, 6; 7, 8, 9 ];
print( b, "\n" );
# print:
[ 18, 24, 30 ]

b *= 2;
b += [ 10, 20, 30 ];
print( b, "\n" );
# print:
[ 46, 68, 90 ]

a = [2][5] : 1;
b = [2][3] : 2;
a[:,2:4] += b;
print( a, "\n" );
# print:
row(0,:):       1       1       3       3       3
row(1,:):       1       1       3       3       3

b++;
print( b, "\n" );
# print:
[ 47, 69, 91 ]

c = a <*> b;
print( c, "\n" );
# print:
[ 517, 828, 1183 ]

print( b', "\n" );
# print:
[ 47, 69, 91 (T)]
# (T) in the end means this is a column vector.
```

```
c = [ 1, 2, 3; 4, 5, 6 ];
print(c,"\n");
# print:
row(0,:):       1       2       3
row(1,:):       4       5       6

print(c',"\n");
# print:
row(0,:):       1       4
row(1,:):       2       5
row(2,:):       3       6
```

7.4 Subindexing of Numeric Array

The subindexing of a numeric array is different from that of a normal array. The way of subindexing Tao numeric arrays is very much similar to that in Matlab. All the indices for each dimension should be put in a single pair of [ ] separated by comma. For example, for a $N \times M$ matrix mat, $mat[1, 2]$ indicates the element at the cross between the second row and the third column (index starts from 0 in Tao). Symbol **:** can also be used to specify a range of indices, e.g., $mat[1 : 3, 2 : 4]$ indicates a submatrix of mat composed of elements at the cross between rows $1, 2, 3$ and columns $2, 3, 4$. The default value in the position before **:** is 0, and the default value in the after is last index in that dimension, i.e., the size in the dimension minus 1. So $mat[: 3, 2 : 4]$ is equivalent to $mat[0 : 3, 2 : 4]$, and $mat[1 :, 2 : 4]$ is equivalent to $mat[1 : N - 1, 2 : 4]$. And if all are omitted in one dimension indicates taking all the indices in that dimension, i.e. $mat[, 2 : 4]$ is equivalent to $mat[0 : N - 1, 2 : 4]$.

Examples:

```
a = [ 1, 2, 3; 4, 5, 6 ];
a[:,1] += 10; # add 10 to the second column.
print( a, "\n" );
# print:
row(0,:):       1       12      3
row(1,:):       4       15      6
```

```
a[0,1:] *= 10;
# multiply the last two columns in the first row by 10.
print( a, "\n" );
# print:
row(0,:):       1       120     30
row(1,:):       4       15      6
```

7.5 Setting Precision for Numeric Array

To improve the computational efficiency and reduce memory usage, Tao provides users the possibility to set desirable precision for real numeric array. The default precision for a numeric array is **double**. For real arrays, their precision can be set to **byte**, **short**, **int**, **float** and **double** in the following way,

```
# For enumeration and range, put the keyword after [.
a = [ byte 1, 2, 3 ];
b = [ short 1 : 1 : 10 ];

# Or, put the keyword before the size specifiers:
c = int [10] : 1;
d = float [5][5] : 10;

e = int [5] : 10.5;
print(e);
# print:
[ 10, 10, 10, 10, 10 ]
```

7.6 Basic Functions for Numeric Array

In addition to the flexible subindexing operation, there are some basic functions and methods to manipulate a numeric array. They are functions **count()**, **max()**, **min()**, **sum()**, **prod()**, **mean()**, **stdev()**, **varn()**, **permute()**, **invpermute()**, **convolute()**, **which()**, **apply()**, **noapply()**, **numarray()**, and methods **size()**, **reshape()**, **resize()**.

count(array): count the number of elements in *array*.

max(array): get the maximum value in *array*.

min(array): get the minimum value in *array*.

sum(array): get the sum of all elements in *array*.

prod(array): get the product of all elements in *array*.

mean(array): get the mean value of all elements in *array*.

stdev(array): get the standard deviation of all elements in *array*.

$$stdev = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

where $\{x_i\}_{i=1..N}$ are the elements in *array*.

varn(array): get the sample variance of all elements in *array*.

$$varn = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$$

where $\{x_i\}_{i=1..N}$ are the elements in *array*.

Note: for all the above functions and the following functions which(), apply(), noapply(), if *array* is followed with subindices, the computation are performed on the specified elements without creating a sub-array!

For example,

```
a = [ 1, 2, 3; 4, 5, 6 ];
b = mean( a[1,:] );
# get the mean value for the second row.
```

numarray(vector, *exprs*, type): create a numeric array of *type* with dimensions specified by *vector* and being initialized by *exprs*. *type* must be a string of these: "*byte*", "*short*", "*int*", "*float*", "*double*" or *complex*. And *vector* must be a vector specifying the size in each dimension. *exprs* can contain transient variables. If *exprs* and *type* are omitted, the array will be created as double with 0s.

```
# Create an array with 2 rows and 4 columns:
a = numarray([2,4]);
print(a,"\n");
# print:
row(0,:):       0       0       0       0
row(1,:):       0       0       0       0

# ++i is evaluated multiply times.
# One can see the evaluation order from this example.
i=0;
b = numarray([2,4], ++i );
print(b,"\n");
# print:
row(0,:):       1       2       3       4
row(1,:):       5       6       7       8

# Here @1 is a transient variable for the first index,
# namely row index, and @2 is for the second (i.e. column
# index). For each element, its indices are passed to
# @1 and @2, and @1+@2+0.5 is evaluated to give a value
# to that element.
c = numarray([2,4],@1+@2+0.5,"float");
print(c,"\n");
# print:
row(0,:):       0.5     1.5     2.5     3.5
row(1,:):       1.5     2.5     3.5     4.5
```

which(): to find elements satisfying certain condition. It can be used in two modes:

1 **which(array** *oper* **value):** where **oper** must be one of $==, !=, <, >, <=, >=$, it returns the indices as an array of elements which satisfy **element** *oper* **value**. For example,

```
a = [ 1, 2, 3; 4, 5, 6 ];
b = which( a<5 );
# find elements with value less than 5.

print( b );
# print:
row(0,:):       0       0
```

```
row(1,:):       0       1
row(2,:):       0       2
row(3,:):       1       0
```

2 **which(array, exprs):** returns the indices as an array of elements which make *exprs* to be true. The passing of the elements and their indices as parameters to *exprs* is done with **transient variables**(see next chapter for details). As an example,

```
a = [ 1, 2, 3; 4, 5, 6 ];
b = which( a, @0 % 2 ==0 );
# find elements with even values.

print( b );
# print:
row(0,:):       0       1
row(1,:):       1       0
row(2,:):       1       2
```

Here @0 is a transient variable represent the element values.

apply(array, exprs): for each element of *array*, evaluate *exprs* and update the element with the value of *exprs*. The passing of the elements and their indices as parameters to *exprs* is also done with transient variables.

noapply(array, exprs): identical to **apply()** except that the elements are not updated by the values of *exprs*. The passing of the elements and their indices as parameters to *exprs* is also done with transient variables. For example,

```
a = [ 1, 2, 3; 4, 5, 6; 7, 8, 9 ];
sum = 0;
noapply( a, sum += @0 ); # another way to calculate sum.

apply( a, @1==@2 );
# here @1 the transient variable representing the first
# index of an element, and @2 is for the second.
# assign 1 to an element if its first and second indices
# are equal, otherwise assign 0.
# this will change "a" to an unit matrix.
```

```
print(a);
# print:
row(0,:):       1       0       0
row(1,:):       0       1       0
row(2,:):       0       0       1

apply( a[1,1:2], 10*(@1+@2) );
# a[i,j]=10*(i+j)
# for the last two elements in the second row

print(a);
# print:
row(0,:):       1       0       0
row(1,:):       0       20      30
row(2,:):       0       0       1
```

permute(array, vec): permute *array*, and return a new array. If *array* is a N dimensional array, *vec* must be a vector of permutation of $[1, 2, ..., N]$. For example, if $N = 3$, *vec* must be $[1, 2, 3]$, $[1, 3, 2]$, $[2, 1, 3]$, $[2, 3, 1]$, $[3, 1, 2]$ or $[3, 2, 1]$. If $vec = [2, 3, 1]$, and $b = permute(array, vec)$, $array[i_1, i_2, i_3] = b[i_2, i_3, i_1]$.

invpermute(array, vec): inverse permute *array*. It is the reverse of permute(), and takes similar parameters. If $vec = [2, 3, 1]$, and $b = invpermute(array, vec)$, $array[i_2, i_3, i_1] = b[i_1, i_2, i_3]$.

convolute(array, kernel): compute the convolution of *array* and *kernel*, and return a new array. *kernel* must has the same dimensions as *array* and with **odd** size in each dimension.

array.size(): probably **size()** should be named as *shape*, since it return the dimensions of *array*, i.e., if *array* is $n_1 \times n_2 \times ...$ array, $array.size()$ will return $[n_1, n_2, ...]$.

array.resize(n_1,n_2,...): resize *array* to $n_1 \times n_2 \times ...$ dimensional array. Memory is reallocated if neccessary, new allocated elements are initialized to 0. It may also take single vector as parameter.

array.reshape(n_1,n_2,...): reshape *array* to $n_1 \times n_2 \times ...$ dimensional array. No memory reallocation, so $n_1 \times n_2 \times ...$ must be the number of elements in *array*. It may also take single vector as parameter.

7.7 Basic Mathematical Functions

A number of basic mathematical functions are supported in Tao. If a function takes (if it can) a numeric array as parameter, it will apply the function to each of the elements and return the results as a new array of the same type and shape, except for **arg()** and **norm()** which, when take a complex array as parameter, will return a double array of the same shape.

For Real Types Only

The functions for real number or array only includes **acos()**, **asin()** and **atan()**.

For Complex Types Only

The functions for complex number or array only includes **arg()**, the angular coordinate of a complex in polar coordinates, and **norm()**, the normal of a complex.

For Both Real and Complex Types

The functions for both real and complex number or array, includes **abs()**, **cos()**, **sin()**, **tan()**, **cosh()**, **sinh()**, **tanh()**, **exp()**, **log()** and **sqrt()**. For real number **abs()** gives the absolute value, and for complex number it gives its radius coordinate in polar coordinates.

For Random Number

srand() set the seed for random number generator, and **rand()** generate a random number.

Chapter 8

Transient Varibles and "Magic" Functions

Tao language supports another interesting feature called **Transient Variables**, which is used for automatical passing of parameters in certain circumstance to provide maximum convenience and flexibility. For example, in string regular expression matching, taking the same example from Section 6.5,

```
a = "====this is a string for the example======";

b = a=~/(@1:\w|\s)*(@2:string)(@3:\w|\s)*/d:={@1,@2,@3}/;

print(d,"\n");
# print: { "this is a ", "string", " for the example" }
```

here transient variables are used to pass match substrings to another expression. In this case, @1, @2, @3 resemble the special variables $1, $2, $3 in Perl to extract matched substrings. However, unlike in Perl where $1, $2, $3 are accessable in more than one statement, the Tao transient variables are always only valid in a single statement.

Except the transient variables used in *regex grouping*, all other type of transient variables are used together with certain functions which often take expressions themselves instead of the results as their parameters. Because of this, I call them as "magic" functions. For example, the "magic" functions **which()**, **apply()** and **noapply()** for numeric arrays all can take expressions

of transient variables. The rules for using transient variables in such functions is that, @0 always represents the element value of a numeric array, and @1 represents the first subindex of the element, @2 represents the second and so on, up to 9 subindices. See Section 7.6.

Another two "magic" functions which also use transient variables is **iterate()** and **iterget()**. They can take arbitrary number of parameters, but the first one usually should be either an array or a range of index specified as $id1 : id2$. For each element of the array in the first parameter or each index between $id1$ and $id2$, the rest expressions in the parameter list are evaluated. The only difference between **iterate()** and **iterget()** is that, **iterate()** only evaluates those expression, while **iterget()** evaluates them and return an array composed of their results (iterget: iterate and get). In those expression in the parameter list, the element of the array or the index in $id1 : id2$ can be accessed by transient variables. These two functions can be nested within each other. For the outmost one, it uses @1 as its transient variable, and the first nested **iterate()** or **iterget()** uses @2, the second nested uses @3 and so on. For example,

```
array1 = { "one", "two" };
iterate( array1, print(@1," : ") );
# print: one : two :

array2 = { 1, 2, { "AA", "BB" } };
iterate( array1, iterate( array2, print( @1, "\t",@2, "\n" ) ) );
# print:
one     1
one     2
one     { "AA", "BB" }
two     1
two     2
two     { "AA", "BB" }

iterate( array2, iterate( @1, print( @2, "\t" ) ), print("\n") );
# print:
1
2
AA      BB


array2.erase( array2.#-1, array2.#-1 );
# array2={ 1, 2 }
```

```
array3 = iterget( array1, iterate( array2, @1, @2 ) );
# array3={{{"one",1},{"one",2}},{{"two",1},{"two",2}}}
```

repeat() is a magic function with no transient variables associated to itself. It takes an expression (usually a regular pattern matching expression or a generator expression, see Chapter 9) as its parameter, it repeats evaluating that expression as long as that expression can generate new result, and return all the results as an array. As you probably have seen in Chapter 6, **repeat()** can be used to repeat to match(by $=\sim$, or unmatch by $!\sim$) all substrings or replace all matched or unmatched substrings. It also can be used to get all rest possible returning values of a generator.

```
routine generator(){
   for(i:=0;i<5;i++){
      yield i;
   }
}
a = generator();
b = repeat( generator() );
# b={1,2,3,4}
```

Chapter 9

Tao Routines

Tao routines are declared with keyword "routine",

```
routine function(a,b){
   a=10;
   b="test";
   return a,b;
}

[r1,r2]=function("AAA",111); // r1=10; r2="test".
r3=function(r1,r2); // r3=[10,"test"].
```

All parameters are passed in by references.

Since the parameters are type-less, it is not necessary to overload functions as what is done in C++. If the types of parameters have to be checked so as to run different scripts, type comparison operators =? and !? can be used, =? checks if the operands in two sides have the same type, and !? checks if the operands in two sides have different types.

```
routine ovldfun( a ){
   if( a =? "" ){
      print("\"a\" is a string.\n");
   }else if( a =? 0 ){
      print("\"a\" is a number.\n");
   }else{
```

```
         print("En, let me guess...\n");
      }
   }
```

In Tao, a function is an object, and its name can be used as a variable (constant variable), which can be assigned to other variables. A variable pointing to a function can invoke that function by using method "run(...)".

```
routine one(){
   print("Printed by ",one,"\n");
}
routine two(){
   print("Printed by ",two,"\n");
}
routine three(){
   print("Printed by ",three,"\n");
}

alias=ovldfun;
alias.run("1");
alias.run(1);
alias.run();

routs = { one, two, three };
foreach( routs : rout ){
   rout.run();
}
```

As you have seen in the first example, **return** is used to return data from a routine. When a routine exits by executing **return** statement or by reaching to the end of the routine, the state to the routine is reseted its initial state. In Tao, there is another keyword can be used in the place of **return**, that is **yield**, which yields data from the rouine and exits the routine. However, a routine exited in this way "remembers" its state at the point where it exited, when this routine is called again, it will start from the statement next to exit point. The state of a routine is never reset as long as it doesn't reach a **return** statement or the end of the routine.

```
   routine generator(){
      for(i:=0;i<5;i++){
```

```
      yield i;
   }
}
a = generator();
b = {};
while( i:=generator() ){
   b.insert(i);
}
# or: b = repeat( generator() );
# b={1,2,3,4}
```

Chapter 10

Object-Oriented Programming in Tao

In Tao language, the syntax for object-oriented programming is slightly similar to C++. When defining a class in Tao, keywords "private", "protected" and "public" can be used to specify the permission to access its member data and methods(functions) from its sub-classes and outside. Member data and methods specified with "private" are absolutely private in the sense that they cannot be access directly by anything else but itself. But they can be accessed by its sub-classes through protected or public methods, or by outside through public methods only.

Like in C++, a member method with the same name as the class name is considered as a constructor. It is always public, even if it is specified with "private". This constructor method is alway executed when an Tao object is created. No destructor is required in Tao, since Tao can take care of destroying an object when it is not used any more.

Note: so far in Tao, class can only inherit from one single parent class. So far Tao only support single line inheritance, namely one class can has at most one parent class. Unlike C++, no additional inheritance flags are require in sub-classing. A sub-class can access the protected and public data and methods in its parent classes.

```
class Base{

   private
```

```
        name,id;
        routine priv(){}

    protected  d,e;

    public  f;

        routine test(p){
            f=p;
            print("Printed by Base::test()\t",p,"\n");
            return f;
        }
        routine func();

        # constructor:
        routine Base(a,b){
            name=a;
            id=b;
        }
}
routine Base::func(){
    print("This method is defined outside of class body\n");
}

class Sub : Base{

    public  aa;

        routine test(p){
            print("Printed by Sub::test()\t",p,"\n");
        }

        # constructor, similar to C++:
        routine Sub(a,b,c):Base(c,a){
            aa=a;
        }
}
```

There is no feature such as polymorphism and virtual functions in Tao. Since an Tao object always knows the exact class type from which it is created. So when you use an object to access a data member or method member, it

always finds the correct one.

```
obj=Sub("s","dog",3);
a="Parameter";
obj.test(a);
# gives:
# Printed by Sub::test()   Parameter
```

Sometimes one may need to known if an object is an instance of a particular class, this can also be done with type comparison operators =?, !?.

```
if( obj =? Base ){
   print("obj is of Base type\n");
}else if( obj =? Sub ){
   print("obj is of Sub type\n");
}
```

Chapter 11

Namespacing & Importing Modules

11.1 Namespacing in Tao

Another important feature of Tao is its flexible namespacing. It is mainly used in dynamic importing of Tao modules, dynamic creating subroutines and classes and other places which are not directly visible from users. What's more, the command line mode of interpretation can more easily be implemented with this namespacing feature in the future.

In fact, the compilation of Tao scripts is associated with two namespaces, one is the input namespace, the other output namespace. The input namespace is used to resolve symbols (such as shared or external variables, subroutines classes) which are not defined in the compiling scripts. And the output namespace is used to store the variables, subroutines and classes defined the compiling scripts. In each individually compiled piece of script, the current namespace is represented by "this" variable (Everything in Tao is a variable or object, so is a namespace. A namespace is a constant variable.), and its members can be accessed with operator "::" or "." (They haven't differentiated yet, but they will in the future).

11.2 Importing of Tao Modules

To import classes and routines defined in another file, use import statement,

$$import\ modname\ :\ "path/file";$$

and access them by $modname :: rout\_class()$ or $modname.rout\_class()$.

```
import cluss:"./sample/class.tao";
# Or: cluss=import("./sample/class.tao");
# A namespace "cluss" is created.

import func:"./sample/function.tao";
# A namespace "func" is created.

# Accessing classes and routines in a namespace.

obj=cluss::klass();
obj.fun();

func::fun();
```

With the import statement, the modules are imported in compiling time. If one want to import in running time, one can use function **import()**. **import()** can also take a second and third parameters which are the same parameters as in **compile()** and **eval()** (See the next section).

11.3 Dynamic Creation Of Tao Routines and Classes

To create routines or classes in running time, the internal function compile() is required.

compile(scripts, nsIn, nsOut): compile a block of Tao scripts. It takes 3 parameters, and returns a namespace variable. The first parameter should be a string of Tao scripts. The second should be the input namespace, it should be used if the a dependency between the scripts to be compiled and

the variables, routines and classes defined in the input namespace. The third is the output namespace, which should be used if one wants to use an existing namespace to store newly compiled variables, routines and classes.

eval(scripts, nsIn, nsOut): evaluate a block of Tao scripts. It takes the same parameters as **compile()**, but after compiling, it also execute the compiled codes.

```
a = "a=1; b=\"2\"; print(a+b,\" \",b+a);";
eval(a);
```

Namespace together with **compile()** provide a possibility to create classes and subroutines in running time.

```
class Base{
   public:
      routine test(){ print("base class\n"); }
}

b = "a dynamic subroutine";

a = "routine dynsub(b)
{ print(\"A dynamic subroutine.\n\",b,\"\n\"); }
print(\"This is compiled in running time.\n\n\");
dynsub(); ";

c = "class Sub : Base { public
routine test2(){ print(\"a dynamic class\n\"); } }";

# "this" is the current default namespace.
# Compile codes "a", and use "this" as input namespace
# to resolve dependency.

ns1 = compile( a, this );
# ns1" is returned as a new namespace storing
# routines and classes defined in "a".

# Run a routine defined in "a":
ns1::dynsub( b );
```

```
# "main()" is the main routine of a compiled code.
ns1::main();

# Use "this" as input namespace, "ns1" as output namespace.
# And "ns1" is return, "ns2" becomes an alias of "ns1".
ns2 = compile( c, this, ns1 );

# Create an object of a class defined in "c":
d = ns2::Sub();
d.test();
```

Chapter 12

Extending Tao with C++

The power of Tao can be extended by writing C++ modules and loading, using them in Tao scripts. An C++ module for Tao is simply a C++ library with certain interface, through which tao interpreter can create an object of a C++ class that is defined in the library and invoke its class methods.

The basic way to create C++ modules for Tao, is to subclass from **TaoPlugin** and reimplement some of the virtual functions, such as **rtti()** to ensure that one can know the exact type of an object in running time, **newObject()** to create an instance of user-defined plugin, **runMethod()** to run a specific method. **TaoPlugin** is defined in header file **taoPlugin.h**.

```
class TaoPlugin : public TcBase
{
  public:
    TaoPlugin(){}
    virtual ~TaoPlugin(){}

    short type()const{ return TAO_PLUGIN; }
    virtual short rtti()const{ return TAO_PLUGIN; }

    virtual TaoPlugin* newObject( TcArray *param=0 ){
      return new TaoPlugin();
    }
    virtual void runMethod( const char *funame,
        TcArray *in=0,TcArray *out=0)
    {
      cout<<"TaoPlugin::runMethod() is not reimplemented.\n";
```

```
    }
    virtual void print( ostream *out=0, bool reserved=0 ){};
};
```

Here **TcBase** is the base class for all C++ types. To pass parameters from Tao scripts to C++ methods, Tao defined a set of C++ type corresponding to each of the basic Tao data types as well as their conversions.

TcNumber	<=>	TaoNumber
TcString	<=>	TaoString
TcComplex	<=>	TaoComplex
TcArray	<=>	TaoArray
TcHash	<=>	TaoHash
TcByteArray	<=>	TaoByteArray
TcShortArray	<=>	TaoShortArray
TcIntArray	<=>	TaoIntArray
TcFloatArray	<=>	TaoFloatArray
TcDoubleArray	<=>	TaoDoubleArray
TcComplexArray	<=>	TaoCompArray

When a method is called by a C++ object in Tao scripts, Tao interpreter invokes **runMethod()**, passing the method name as the first parameter and two **TcArray** *in* and *out* as the second and third parameters. The first **TcArray** *in* contains C++ types converted from Tao variables in the parameter list of the method in Tao scripts. The second **TcArray** *out* is passed to hold returned variables, which will be converted into Tao data types.

As a simple example,

```
#include"stdio.h"
#include"taoPlugin.h"

// Use this macro defined in "taoPlugin.h"
// to do certain initialization.
INIT_TAO_PLUGIN;

class MyPlugin : public TaoPlugin{
   // Use this macro to define a special constructor
   // for plugin registration:
   TAO_PLUGIN(MyPlugin);
```

```
char   *name;

public:
MyPlugin(){ name=0; }

// This function must be reimplemented to create
// your own plugin objects.
TaoPlugin* newObject( TcArray *param=0 ){
   return (TaoPlugin*)new MyPlugin;
}

// This funtion must be reimplemented to invoke other
// methods or do the work you want. "funame" is the
// function name you want to use in Tao script.
// "in" is an array holding parameters passed in.
// "out" is an array to hold return variables
// which can be use in Tao scripts.
void runMethod( const char *funame,
      TcArray *in=0, TcArray *out=0 )
{
   // Suppose all parameters are passed properly.
   if( strcmp(funame,"setName")==0 ){
      TcString *tstr=(TcString*)in->getElement(0);
      char *chars=tstr->getChars();
      name=(char*)realloc( name,
            (strlen(chars)+1)*sizeof(char) );
      memcpy( name, chars,
            (strlen(chars)+1)*sizeof(char) );
   }else if( strcmp(funame,"getName")==0 ){
      TcString *tstr=new TcString;
      tstr->setChars(name);
      out->insert( tstr );
   }else if( strcmp(funame,"reshapeArray")==0 ){
      TcNumArray *array=(TcNumArray*)in->getElement(0);
      TcIntArray *shape=(TcIntArray*)in->getElement(1);
      // reshape() is a method define in TcNumArray
      // to reshape a numeric array.
      array->reshape( shape->getAsVector(), shape->size() );
   }else{
      printf("Warning: method undefined!\n");
   }
```

```
   }
};

// One instance must be declared to register you plugin
// with the name which you want to be used as class name
// in Tao scripts. Here the C++ class name is used.
MyPlugin plugin("MyPlugin");
```

Now if this C++ module is compiled into "cppmod.so"(or "cppmod.dll"), you may do like this in you Tao scripts,

```
load CppMod:"./cppmod.so"; # or cppmod.dll
# Or: CppMod=load("./cppmod.so");

obj = CppMod::MyPlugin();
obj.setName("this is my plugin");
name = obj.getName();
print("name returned by getName(): ", name, "\n");
# print:
name returned by getName(): this is my plugin

array = [2][2][2]:1;
shape = [int 2, 4];

print("array before reshaping:\n");
print( array );
# print:
array before reshaping:
row(0,0,:):     1       1
row(0,1,:):     1       1
row(1,0,:):     1       1
row(1,1,:):     1       1

obj.reshapeArray( array, shape );

print("array after reshaping:\n");
print( array );
# print:
array after reshaping:
row(0,:):       1       1       1       1
row(1,:):       1       1       1       1
```

```
# Note:
# "array" is modified in the C++ module.
```

The load statement loads the C++ modules in compiling time. If one want to load in running time, one can use function **load()**.

If *TaoPlugin* :: *rtti*() is reimplemented, it is also possible to check the plugin type in Tao scripts with .? or *about*() and type comparison operators =?, !?.

Note that, only the two header files **taoCpptype.h** and **taoPlugin.h** are needed to build a C++ module for Tao, no additional library is required for linking **!**

For more detailed documentation for the C++ type for Tao, please go to:

Chapter 13

Miscellaneous Issues and Functions

13.1 Other Built-in Functions

about(*exprs* **):** return the type and address of an object returned by *exprs* as a string.

system(*command* **):** execute a system command represented as a string.

time(): get the time since the Epoch (00:00:00 UTC, January 1, 1970), measured in seconds.

astime(*seconds*, $unit_{max}$ **):** return the time of *seconds* represented in $unit_{max}$, which must be a string and be one of "*day*", "*hour*" or "*minute*", and the units smaller than $unit_{max}$. It returns a hash with time units as keys.

```
a = 10000;
b = astime( a, "hour" );
# b={ "hours"=>2, "minutes"=>46, "seconds"=>40 }
```

asctime(*seconds* **):** represent *seconds*, elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC), as calendar time. It returns a hash with year, month and day etc as keys, which are similar to the

members in the C structure **tm** define in **time.h**. However the ranges of the values are slightly different, namely, the "*month*" ranges from 1 to 12, and the "*yday*" ranges from 1 to 366.

```
a = time();
b = asctime( a );
# b={"day"=>23,"hour"=>16,"isdst"=>0,"minute"=>40,"month"=>4,
# "second"=>4,"wday"=>6,"yday"=>113,"year"=>2005}
```