Peanut
Script Language

Author: farmerliao.tw@gmail.com

Index

1 Introduction

Peanut is an interpreted programming language.
The motivations to devise Peanut language are to (1) provide a C-like script language for programmers who compose C applications and (2) provide a choice to encapsulate C libraries/programs for crafting applications in scripts. Syntaxes of statements and expressions, operators, precedencies of operators, flow controls etc. almost adopt what C defines.

Besides the C-like feature, there are other features and they are:

1. Multitasking programming
2. Function, variable, and constant declarations with namespace resolutions
3. Untyped variable
4. Value passing and reference passing
5. Sub-array passing
6. Object programming
7. Extensible

Some of the features, for the object programming feature as an example, are not seen in C, and some are found in many languages.

1.1 Packages and Files

Peanut is provided as a package of files. Each package is targeted at one operating system. Files in a package are listed int the following table:

doc/*.pdf	peanut.pdf : This file; describing Peanut script language. peanut_ext_bna.pfd : a Peanut extension module providing big number (with mass digits) arithmetics in Peanut scripts peanut_ext_clib.pdf : a Peanut extension module providing frequently used C functions for use in Peanut scripts
examples/*.pnt	Peanut script examples used in this document
pxm/*.pxm	pxm_bna.pxm : an extension module that provides functions of big number arithmetics pxm_clib.pxm : an extension module that provides functions of frequently used C functions
pni.exe	The Peanut Interpreter
*.pnt	Some Peanut example scripts

The Peanut interpreter execution file (pni.exe or pni) and extension module files in a package are system dependent.
Currently, packages for Windows and Ubuntu are provided only.

1.2 Script Execution

To run a script, you should type the Peanut interpreter file name (e.g. pni.exe) followed by the script file name on Windows "Command Prompt" or Ubuntu Console. For example:

```
pni.exe    c02_qsort.pnt
```

Besides, you could input arguments after the script file names. For the following as an example, two arguments, 2 and 3000, are specified to the script file "c03_sqrt.pnt". The argument 2 is specified to calculate the square root of 2; and the argument 3000 is passed in order to calculate the square root with 3000 fraction digits.

```
pni.exe    c03_sqrt.pnt    2    3000
```

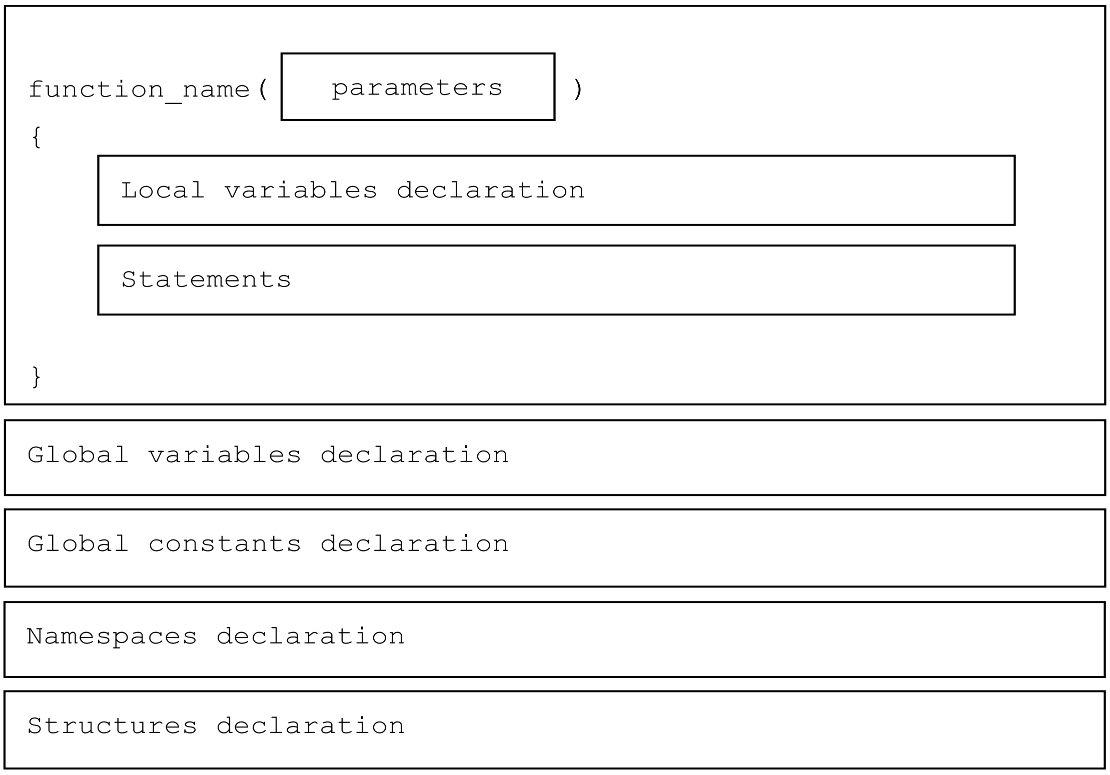
2 Overall Script Structure

Let’s write the first script in Peanut.
Example 1 shows a script with only one function implemented. In this example, the main function prints a string “Hello World!”. This script is a tiny and complete Peanut script.

```
main()
{
    print("Hello World!");
}
```

Example 1– p210.pnt

Basically, a Peanut script is composed with functions, global variables, global constants, structures, and namespaces. Function `main` is the entry point, from which the script starts.



function_name (
parameters
)
{
Local variables declaration
Statements
}
Global variables declaration
Global constants declaration
Namespaces declaration
Structures declaration

Figure 1

Figure 1 depicts the structure of a Peanut script in a file from a high-level view.
A function starts with a unique name followed by a pair of parenthesis. Parameters of the function may be defined within the parenthesis. A pair of braces follows to

encapsulate the body of the function. In the body of the function are local variables and the function's statements. More functions can be implemented in this manner, but please note that a function can't be declared within another.
Outside the functions might be global variables, global constants, and structures.
A larger script could be decomposed into multiple files and are integrated together through preprocessor directive #include. The outlook of a script with #include directives looks like Figure 2.

```
/* main file */
#include <file1>
#include <file2>
main()
{
    ...
}
...

/* file1 */
sum(a, b)
{
    return (a + b);
}
...

/* file2 */
sub(n1, n2)
{
    Return (n1 - n2);
}
...
```

Figure 2

Refer to Example 2. In line 1, preprocessor directive #include includes implementations in file "p220_routine.pnt". Functions `sum10` and `sumN`, implemented in "p220_routine.pnt" file, are called in function `main`.
From line 3 to line 5, global constant C1 and global variables a2 and v1 are declared.
Function main engages line 7 to line 29. Within a function, like the function main in Example 2, local variables, line 9 to line 10, are always declared before statements.

A line comment starts from // to the end of that line. Wordings after // have no effects in the script file. Refer to line 12 for the line comment.
A block comment starts from /* and ends with */ for that we see on line 16 and 17 as the example.
Line 13 to line 28 are statements of the function main.

```
 1  #include <p220_routine.pnt>
 2
 3  const C1 = 20;
 4  var a2[C1];
 5  var v1 = 0;
 6
 7  main()
 8  {
 9     var N[50];
10     var i = 0, sum = 0;
11
12     // initialize a2 as {1, 2, 3, ..., C1}
13     for (i = 0; i < C1; i++)
14        a2[i] = i + 1;
15
16     /* initialize N as
17       {1, 2, 3, ..., 49, 50}; */
18     for (v1 = 0; v1 < 50; v1++)
19        N[v1] = v1 + 1;
20
21     sum = sum10(a2[0..9]);
22     sum += sum10(a2[10..19]);
23     print("sum of numbers in a2 is ", sum, "\n");
24
25     sum = sumN(20, N[0..19]);
26     sum += sumN(11, N[20..30]);
27     sum += sumN(19, N[31..49]);
28     print("sum of numbers in N is ", sum);
29  }
```

Example 2 – p220_overall.pnt

3 Types of Values

A Peanut script supports 3 types of values: integers, floating-point numbers, and strings.
Integers are 0, 1, -1, 2, -2, 3, -3, …, 100, -100, ‘A’, ‘\t’, ‘\n’ etc.
Floating-point numbers are numbers with decimal point for 0.1, -13.0, 2.007 as examples.
And a string is collection of a sequence of characters leading and ending with a double quote (“). For example, “Hello word”, “12345678”, and “3.14” are strings.
See Example 3. Values 1, and 10 on line 5 are integers. Value 1.25 on line 7 is a floating-point number. On line 8, “x = “, “y = “, and “\n” are strings.

```
 1  main()
 2  {
 3      var x, y;
 4
 5      for (x = 1; x < 10; x += 1)
 6      {
 7          y = 1.25 * x + 11;
 8          print("x = ", x, ", y = ", y, "\n");
 9      }
10  }
```

Example 3 - p310

4 Variable

Variables are used to store values while the script is run. In a Peanut script, all variables should be explicitly declared before they are used.

To declare variables, variable declaration statements are used.

A variable declaration statement starts with key word var and ends with a semi-column. Between the keyword var and semi-column could be a single variable name or a list of variable names separated with comma(s). It is optional for the programmers to declare one variable with an initialization by specifying an initialization expression with an equal sign (=) after the variable name. Variables without initializations are automatically initialized with 0's.

Refer to the followings for the syntax of a variable declaration statement.

```
var v1;
var v2 = 10;
var v3 = "a string";
var v4, v5 = 5, v6 = 6.0;
```

4.1 Scope

There can have local variables and global variables in a script.

One of the major differences between a local variable and a global variable is their scope.

A scope of a variable is the variable's visibility in the script. One local variable is only visible within the function in which the variable is declared; and outside the function, the local variable is invisible and is invalid. A global variable, however, is visible in any function.

Refer to Example 4. Statements from line 1 to line 3 declare global variables gv1, gv2, gv3, gv4, and gv5. And gv5 is initialized as 5 as it is declared.

Between line 7 and line 11, local variables lv1, lv2, lv3, lv4, lv5, and lv6 are declared within function main. Variable lv4 is initialized as a string when it is declared, and lv6 is initialized as the sum of gv5 and lv3. From line 13 to line 22, each variable is outputted, but the statements on line 16 and 17 are a little different from all the other statements. The 2nd argument to function print on line 16 or 17 is not declared as a local variable within line 7 and line 11, and is not seen on line 1, 2, and 3, either.

In fact the 2nd argument of print on line 16 and 17 (i.e. gv6 and gv8) are declared as global variables on line 25, which is at the place outside any function.

```
var gv1;
var gv2, gv3, gv4;
var gv5 = 5;

main()
{
    var lv1;
    var lv2, lv3 = 3;
    var lv4 = "four";
    var lv5 = 5,
        lv6 = gv5 + lv3;

    print("gv1 = ", gv1, "\n");
    print("gv4 = ", gv4, "\n");
    print("gv5 = ", gv5, "\n");
    print("gv6 = ", gv6, "\n");
    print("gv8 = ", gv8, "\n");

    print("lv1 = ", lv1, "\n");
    print("lv2 = ", lv2, "\n");
    print("lv4 = ", lv4, "\n");
    print("lv6 = ", lv6, "\n");
}

var gv6 = 6, gv7, gv8 = "eight";
```

Example 4 – p410.pnt

4.2 Limitation of Local Variables

Comparing a local variable's declaration and usage with a global variable's, another major difference between a local variable and a global variable is that a local variable should always be defined first and then used. This kind of rule that a variable's declaration position should be prior to usage position is called backward reference. And another rule that allowing a variable being used at the place prior to where it is declared is called forward reference. A local variable declaration and usage is limited by a backward reference, but a global variable has no such a limitation.
Besides, between any two local variable declaration statements cannot exist any non-local variable declaration statement.

4.3 Limitation of Global Variables

A limitation of global variable is that the operands in a global variable's initialization expression should be constants only. Neither function calls nor references to other global variables are allowed to be the operands of a global variable's initialization expression.

4.4 Elementary Variable

A variable that is declared to store one single value is called an elementary variable. Every variable in Example 4 is an elementary variable.

4.5 Register Variable

4.6 Array Variable

An array variable is a variable that multiple data values can be stored in it.
The concept of an array variable can be visualized as Figure 3. In the figure, one array variable arr stores 7 data values. Any value in the array is indexed by 0 or a positive integer. Each value is stored or read by specifying the index. For example, value 1.02 engages index 0 of the array, and value "X", which is a string, engages index 4.

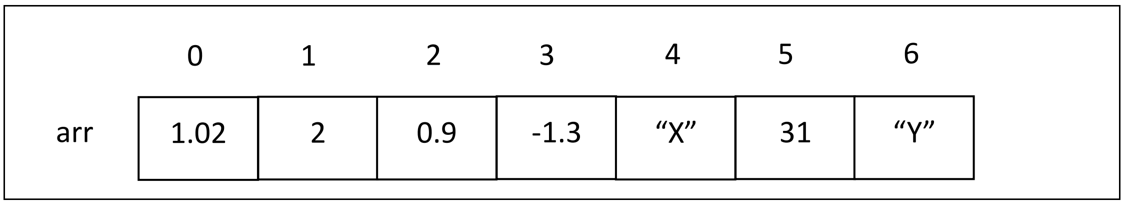


Figure 3

We say that one "storage" in the array to store one data is called an element.
Data values are read or updated by referring the array variable name and indexes.
The number of indexes used to access the data value is called the dimension number.
In a Peanut script, the maximum dimension number of an array variable is 8 (and of course the minimum is 1). **No initialization expression is supported to initialize an array variable.**
To declare an array variable, pairs of brackets are used after the name of the array

variable with constant expressions within the brackets to indicate dimension sizes. Valid statements to declare array variables are illustrated as the following.

```
var arr1[10];
var arr2[5+5];
var arr3[C1+C2];
var arr4[10][10][10];
var arr5[SZ1][SZ2][SZ3+SZ4];
```

Refer Figure 4 to for the description of an array declaration.
One thing should be noted is that all the operands of all array size expressions should be constants, either constant values or named constants. Neither a variable nor a function call is allowed.

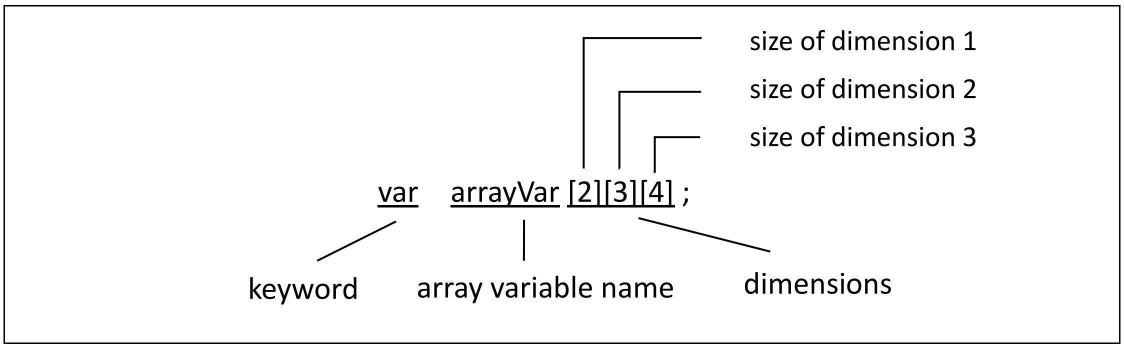


Figure 4

To access an element in the array, the variable name of the array and indexes of all dimensions should be specified. Refer to Figure 5 for read an element from the array declared in Figure 4.

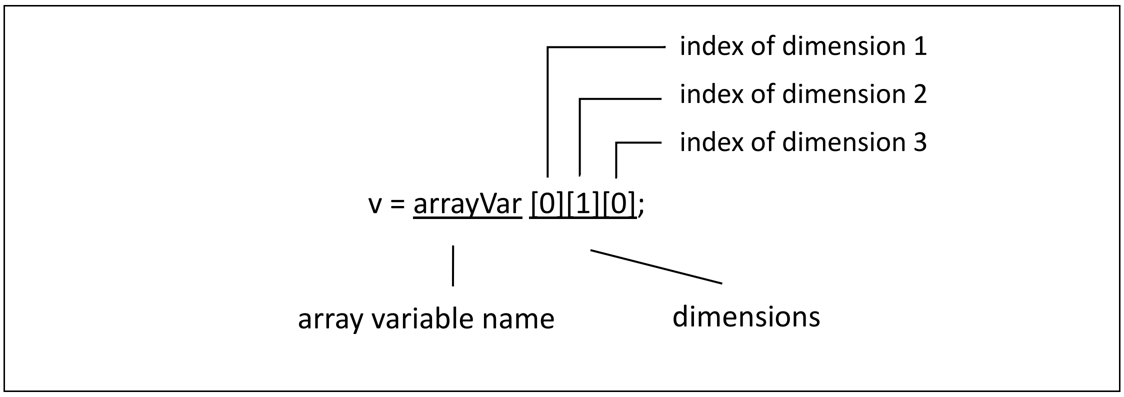


Figure 5

Refer to Example 5. We see that there is a global array variable, gv, declared on line 4. Dimensions of array gv is 2; the size expression of the 1st dimension is SZ1 and the size expression of the 2nd dimension is SZ2. And note that both SZ1 and SZ2 are constants.

```
 1  const SZ1 = 2;
 2  const SZ2 = 10;
 3
 4  var gv[SZ1][SZ2];
 5
 6  main()
 7  {
 8     var v2[SZ2], v3[SZ2];
 9     var i, index;
10
11     for (index = 0; index < SZ2; ++index)
12     {
13        v2[index] = index;
14        v3[index] = SZ2 - index;
15
16        gv[0][index] = v2[index] + v3[index];
17        gv[1][index] = v2[index] * v3[index];
18
19        print(gv[0][index], " ", gv[1][index], "\n");
20     }
21
22  }
```

Example 5 – p420.pnt

On line 13, we see that v2, which is an 1-dimension array, is accessed by specifying 1 index; On line 16, gv, which is a 2-dimension array, is accessed with 2 indexes specified.

Say S_1, S_2, …, S_m are sizes of a m-dimension array variable. Usually, we use S_1xS_2x …xS_m to describe an array's "structure".

5 Literal Constant and Named Constant

A constant is a value that cannot be altered during the script is run.
In a Peanut script, constants that are explicitly specified by values in the places where they are used are called literal constants. See Example 6. Values 12, 3.14, and “Area = “, and ‘A’ are all literal constants.

```
main()
{
    var r = 12;
    var a = r * r * 3.14;

    print("Area = ", a);
}
```

Example 6 – p510.pnt

Sometimes a constant is used dozens of times in a script. In Peanut, a programmer is allowed to declare global constants with values, like global variables declarations, and uses them in the script by names of constants. The keyword const is used to define named constants.
A named constant, for the following pi as an example, is defined with starting keyword const.

const pi = 3.14;

After the keyword const is an assignment statement. In the left-hand side of the assignment operator (i.e. =) is the name of the named constant; where the right-hand side of the operator is an arithmetic expression. Other defined named constants or literal constants are allowed in the expression.
Hence, Example 6 script can be rewritten as that of Example 7.

```
const pi = 3.14;

main()
{
    var r = 12;
    var a = r * r * pi;

    print("Area = ", a);
}
```

Example 7 - p520.pnt

But note that a named constant can be defined globally only. Local named constants within a function are not supported.

Similar to a global variable, operands in an initialization expression of a named constant have to be constants, either literal constant values or named constants; neither function calls nor variables are allowed.

Peanut also allow the programmer to use character constants. And character constants are treated as integers when they are processed by Peanut. The following program outputs 75 instead of symbol K on the screen.

```
print(‘K’);
```

6 Function

A function takes arguments as inputs through its parameters, executes statements, and eventually returns a value to the caller. Value 0 is always returned after one function finishes its execution implicitly if there is no explicit return statement executed. In one function, different types of data can be returned.
In a Peanut script, functions are classified into 3 types; they are built-in functions, user functions, and extension functions.

6.1 Built-in Function

A built-in function is a function implemented in the Peanut interpreter.

6.1.1 print

Prototype	`print(val, ...)`	
Description	Print values of arguments. The number of arguments can be any. The value type of the arguments can be any. If the value is an integer, the value will be shown in the decimal format.	
Return Value	0	
Parameter	val	The argument can be a: literal constant, named constant, elementary variable, array variable, or object variable. Any number of arguments can be specified.
See Also	printh	

```
struct Point_t
{
    var x;
    var y;
};

main()
{
    var abc[3];
    var i;
    struct Point_t pA;

    print("Hello World\n");
    print(0, 1.0, "2");

    print("\n\n");
    for (i = 0; i < 3; ++i)
        abc[i] = 100 * i;
    print(abc);

    print("\n\n");
    pA.x = 5.0;
    pA.y = -2;
    print(pA);
}
```

Example 8 - p610.pnt

6.1.2 printh

Prototype	`printh(val, ...)`	
Description	Print values of arguments. The number of arguments can be any. The value type of the arguments can be any. If the value is an integer, the value will be shown in the hexadecimal format.	
Return Value	0	
Parameter	val	The argument can be a: literal constant, named constant, elementary variable, array variable, or object variable. Any number of arguments can be specified.
See Also	print	

6.1.3 intsize

Prototype	`intsize(void)`	
Description	Get the size in bit of an integer constant.	
Return Value	32 or 64	
Parameter	N/A	This function has no parameter.
See Also		

```
main()
{
    print("Integer size in bit is: ", intsize());
}
```

Example 9 - p620.pnt

6.1.4 isstring

Prototype	`isstring(val)`	
Description	Determine the argument is a string or not.	
Return Value	0 : not a string 1: a string	
Parameters	val	The argument can be any value
See Also		

```
main()
{
    var ans1 = isstring("Hello World");
    var ans2 = isstring(1+2+3);
    var ans3, array[4];

    ans3 = isstring(array);

    print("ans1 = ", ans1, "\n");
    print("ans2 = ", ans2, "\n");
    print("ans3 = ", ans3);
}
```

Example 10 - p630.pnt

6.1.5 task_create

Prototype	`task_create(tname, arg2t)`	
Description	Create a task run in parallel.	
Return Value	0: fail Otherwise: the handle of the task	
Parameters	tname	Name of an user function with 1 parameter.
	arg2t	An argument (value) passed to the created task.
See Also		

```
tfunc(a)
{
   print("one task is created with argument a = ", a, "\n");
   sleep(10);
}

main()
{
   var t1Handle, t2Handle;

   t1Handle = task_create(tfunc, 1);
   t2Handle = task_create(tfunc, "str2");

   task_join(t1Handle);
   print("t1 terminate!\n");

   task_join(t2Handle);
   print("t2 terminate!");
}
```

Example 11 - p640.pnt

6.1.6 task_join

Prototype	`task_join(tHandle)`	
Description	Wait for the task terminates. Note that the caller to this function is suspended until the task terminates.	
Return Value	0	
Parameters	tHandle	The value returned from task_create function.
See Also	`task_create`	

6.1.7 task_isrunning

Prototype	`task_isrunning(tHandle)`	
Description	Query the task, which is specified by the task handle, is running or not.	
Return Value	1 = the task is running 0 = the task is not running (is terminated)	
Parameters	tHandle	The value returned from task_create function.
See Also	`task_create`	

6.1.8 mutex_create

Prototype	`mutex_create()`	
Description	Create a multual exclusion object (mutex in short) for controlling synchronizations between tasks.	
Return Value	0: fail Otherwise: the handle of the mutex	
Parameters	N/A	This function has no parameter.
See Also		

```
var mutex;

main()
{
    var t1, t2;

    mutex = mutex_create();

    t1 = task_create(t1, 1);
    t2 = task_create(T3, 2);

    task_join(t2);
    task_join(t1);
}

t1(myId)
{
    var i;

    sleep(10);

    mutex_lock(mutex);
    print("t1:myId=", myId, ", ");
    for (i = 0; i < 10; ++i)
        print(i);
    print("\n");
    mutex_unlock(mutex);
}

T2(myId)
{
    var i;

    mutex_lock(mutex);
    print("T2:myId=", myId, ", ");
    for (i = 0; i <= 20; ++i)
        print(i);
    print("\n");
    mutex_unlock(mutex);
}
```

Example 12 - p650.pnt

6.1.9 mutex_lock

Prototype	`mutex_lock(mHandle)`	
Description	Lock down a mutex in order to access critical data shared between tasks.	
Return Value	TBD	
Parameters	mHandle	The handle of the mutex to be locked. This value is returned from a call to mutex_create().
See Also	mutex_create	

6.1.10 mutex_unlock

Prototype	`mutex_unlock(mHandle)`	
Description	Unlock the mutex which was locked by the caller task after accessing critical data.	
Return Value	TBD	
Parameters	mHandle	The handle of the mutex to be unlocked. This value is returned from a call to mutex_create().
See Also	mutex_create	

6.1.11 cond_create

Prototype	`cond_create()`	
Description	Create a condition variable for synchronizations between tasks.	
Return Value	0: fail Otherwise: the handle of the created condition variable	
Parameters	N/A	This function has no parameter.
See Also	cond_wait cond_broadcast	

6.1.12 cond_wait

Prototype	`cond_wait(hCond, hMutex)`	
Description	Suspend the task, which calls this function, to wait for the broadcast sent through hCond. To enter the task into the state waiting for a broadcast sent through hCond, the task should first locks hMutex before cond_wait() is called. Once the task successfully enters the state waiting for the condition (i.e. the task is blocked), the mutex hMutex will be automatically unlocked; other tasks can use the same hMutex and hCond pairs to enter the state waiting for the same condition broadcasted.	
Return Value	0	
Parameters	hCond	the handle of a condition variable
	hMutex	the handle of a mutex variable
See Also	cond_broadcast	

6.1.13 cond_broadcast

Prototype	`cond_broadcast(hCond)`	
Description	To wake up (i.e. unblock) all tasks which are waiting for the broadcast sent through hCond condition variable.	
Return Value	0	
Parameters	hCond	A condition handle created by cond_create() function
See Also	cond_wait	

```
var mutex1;

var mutex;
var condVar;

var count_g = 0;
var suspendFlag_g = 1;
var roundData_g;

const TASK_NUM = 20;
const TEST_ROUNDS = 5;

Ti(mytid)
{
   var i, l;

   for (i = 0; i < TEST_ROUNDS; ++i)
   {
      mutex_lock(mutex);
      ++count_g;
      cond_wait(condVar, mutex);
      mutex_unlock(mutex);

      sleep(30 + mytid % 7);
      mutex_lock(mutex1);
      print(i, " ", roundData_g, " ", mytid, "\n");
      mutex_unlock(mutex1);

      mutex_lock(mutex);
      --count_g;
      mutex_unlock(mutex);

      while (suspendFlag_g == 1)
         sleep(1);
   }
}

```

```
main()
{
   var i, c;
   var tids[TASK_NUM];

   mutex1 = mutex_create();
   mutex = mutex_create();
   condVar = cond_create();

   for (i = 0; i < TASK_NUM; ++i)
      tids[i] = task_create(Ti, i);

   for (i = 0; i < TEST_ROUNDS; ++i)
   {
      //prepare data
      roundData_g = i + 1000;
      print("\n\nround-", i, ": data = ", roundData_g, "\n");

      //wait for all tasks entering 'cond_wait'
      while (1)
      {
         mutex_lock(mutex);
         c = count_g;
         mutex_unlock(mutex);

         if (c != TASK_NUM)
            sleep(1);
         else
            break;
      }

      mutex_lock(mutex);

      //control Ti tasks to suspend after their doing jobs
      suspendFlag_g = 1;

      //broadcast to wake up tasks
      cond_broadcast(condVar);
```

```

        mutex_unlock(mutex);

        //wait for all tasks finishing their jobs
        while (1)
        {
            mutex_lock(mutex);
            c = count_g;
            mutex_unlock(mutex);

            if (c != 0)
                sleep(5);
            else
                break;
        }

        suspendFlag_g = 0;
    }

    for (i = 0; i < TASK_NUM; ++i)
        task_join(tids[i]);

    print("\nEnd\n");
}
```

6.1.14 backtrace

Prototype	`backtrace()`	
Description	To dump the list of function calls that are currently active in the task.	
Return Value	1	
Parameters	N/A	the handle of a condition variable
See Also		

6.1.15 sleep

Prototype	`sleep(ms)`	
Description	Suspend the caller task for a while.	
Return Value	0	
Parameters	ms	Time in mini second.
See Also	mutex_create	

6.2 User Function

User functions are functions implemented in Peanut by the programmers.
A user function starts with a unique function name. After the function name is the parameters definition which is a list of names separated by commas and encapsulated within a pair of parentheses if this function requires arguments. If one function requires no argument passed, i.e. the function has no parameter, the programmer could simply keep blank for the parameter definition or put the keyword void.
Following the parameter definition is the function body enclosed within a pair of braces. Within the function body, the local variables should be defined first before they are used in the function statements.
Refer to Example 13. Functions implemented by the programmer are sum, mul, showGlobals, test1, and main. Among them, both sum and mul functions have two parameters. All the other functions (showGlobals, test1, and main) have no parameter.

```
var g1 = 2, g2 = 3;

sum(a, b)
{
    return (a + b);
}

mul(a, b)
{
    return (a * b);
}

showGlobals()
{
    print("g1 = ", g1, "\n");
    print("g2 = ", g2, "\n");
}

test1(void)
{
    var v1, v2;

    v1 = sum(g1, -g2) == sum(-g1, g2);
    v2 = mul(g1, -g2) == mul(-g1, g2);

    print("v1 = ", v1, "\n");
    print("v2 = ", v2, "\n");
}

main()
{
    test1();
    showGlobals();
}
```

Example 13 - p660.pnt

Like a local variable, the visibility of a parameter is visible within the function only. Different functions can have same name local variables or parameters, but please note that same name variable or parameter in one function will not affect that in another.

It is allowed to name a local variable or a parameter with the same name as a global variabl’s or a global constant’s. See the following Example 14. Within function func1, the global constant c1 is redefined as a parameter on line 4, and the global variable v1 is redefined as a local variable on line 6. Within func1, c1 and v1 no longer represent the global constant and global variable.
Within function func2, v1 is used as a parameter name. Refering to v1 within func2 represents the argument passed-in by the caller instead of the global variable v1. But referring to c1 within func2 still refers to the global constant c1.
Within function main, c1 and v1 represent the global constant and global variable.

```
1  const c1 = 100;
2  var v1 = "Hello World";
```

```

func1(c1)
{
   var v1;

   v1 = c1 * c1 + 1;

   print("c1 = ", c1, "\n");
   print("v1 = ", v1, "\n");
}

func2(v1)
{
   print("c1 = ", c1, "\n");
   print("v1 = ", v1, "\n");
}

main()
{
   func1(5);
   func2(6);

   print("c1 = ", c1, "\n");
   print("v1 = ", v1, "\n");
}
```

Example 14 – p670.pnt

Output of the Example 14 is:

```
c1 = 5
v1 = 26
c1 = 100
v1 = 6
c1 = 100
v1 = Hello World
```

6.2.1 Array Parameter

In the Example 14, value 5 is passed to a parameter c1 of function func1 on line 22. If we want to pass thousands of values to a function in this way, thousands of parameters would be a problem.
Similar to an array variable declaration, a parameter can be declared as an array parameter!
To declare an array parameter, put pairs of brackets after the parameter name and put a dimension size within each pair of brackets like the following Example 15.

```
maxSum(a[1000], b[1000])
{
    var i, s, max;

    max = a[0] + b[0];
    for (i = 1; i < 1000; ++i)
    {
        s = a[i] + b[i];
        if (s > max)
            max = s;
    }
    print("Max. sum.  = ", max, "\n");
}

minProd(a[2][1000])
{
    var i, p, min;

    min = a[0][0] * a[1][0];
    for (i = 1; i < 1000; ++i)
    {
        p = a[0][i] * a[1][i];
        if (p < min)
            min = p;
    }
    print("Min. prod. = ", min, "\n");
}

main()
{
```

```
31     var A[2][1000];
32     var i, j;
33
34     for (i = 0; i < 2; ++i)
35     for (j = 0; j < 1000; ++j)
36        A[i][j] = 1 + ((37 + i*10 + j) * 48 + 39) % 1381;
37
38     maxSum(A[0], A[1]);
39     minProd(A);
40  }
```

Example 15 – p680.pnt

In the above example, function maxSum takes 2 parameter arrays. Parameter a is a 1-dimension array with 1000 elements; and so is parameter b; and function minProd takes 1 parameter array which is a 2-dimension array.

On line 31, A is declared as a 2-dimension array with the dimension sizes 2 and 1000.

On line 38, the function main calls function maxSum with 1-dimension arrays A[0] and A[1].

On line 39, the function main calls function minProd with 2-dimension array A.

6.2.2 Varied-size Array Parameter

In many cases, functions have to be implemented to process non-constant number of datum. In a Peanut script, a varied-size array parameter is allowed.

A varied-size array parameter is declared like a normal array parameter (fixed-size array parameter) but dimension sizes are not specified. Without specifying the dimension sizes of a parameter array, real sizes of the array parameter's dimensions would be determined by the arguments.

See the following Example 16. On line 25, function func1 is called with passing a 10x10 array as the 3rd argument; On line 26, a 3x4 array is passed.

```
1  func1(dsize1, dsize2, a[][])
2  {
3     var i, j;
4
5     for (i = 0; i < dsize1; i++)
6     for (j = 0; j < dsize2; j++)
7        print("a[", i, "][", j, "] = ", a[i][j], "\n");
8     print("\n");
}

main()
{
    var i, j;
    var A[10][10];
    var B[3][4];

    for (i = 0; i < 10; ++i)
    for (j = 0; j < 10; ++j)
       A[i][j] = i * 10 + j;

    for (i = 0; i < 3; ++i)
    for (j = 0; j < 4; ++j)
       B[i][j] = i * 100 + j;

    func1(10, 10, A);
    func1(3, 4, B);
}
```

Example 16 – p690.pnt

6.2.3 Call by Value

In Example 15, the function call maxSum(A[0], A[1]) on line 38 causes to copy data values in A[0] array and A[1] array copied to parameters a and b of function maxSum before maxSum function is executed. Copying data value(s) from a variable to the parameter in order to pass argument(s) is called call-by-value, and we called this kind of parameter a call-by-value parameter.

A call-by-value parameter has its own storage space for data. In Example 15, parameters a and b on line 1 are call-by-value parameters. Updating a call-by-value parameter in one function has no effect to variable from which data value is copied.

Take a look at Example 17. On line 24, function main calls function max with 3 arguments x, y, and z whose datum are copied to parameters a, b, c of the max function respectively. Within function max, parameter a might be updated.

```
 1  max(a, b, c)
 2  {
 3     print("a = ", a, ", b = ", b, ", c = ", c, "\n");
 4
 5     if (b > a)
 6        a = b;
 7     if (c > a)
 8        a = c;
 9
10     print("a = ", a, ", b = ", b, ", c = ", c, "\n");
11     return (a);
12  }
13
14  main()
15  {
16     var x, y, z;
17     var maxv;
18
19     x = 32 % 10;
20     y = (x + 33) % 10;
21     z = (x * (y +3)) % 10;
22
23     print("x = ", x, ", y = ", y, ", z = ", z, "\n");
24     maxv = max(x, y, z);
25     print("x = ", x, ", y = ", y, ", z = ", z, "\n");
26     print("maxv = ", maxv);
27  }
```

Example 17 – p6a0.pnt

See the followings for the output of Example 17. We see that a is changed after the operations from line 5 to line 8. However, variable x is not changed.

```
x = 2, y = 5, z = 6
a = 2, b = 5, c = 6
a = 6, b = 5, c = 6
x = 2, y = 5, z = 6
maxv = 6
```

6.2.4 Call by Reference

In some cases, we hope a variable referred as an argument in a fuction call be updated by the callee function besides passing value. This functionality is achieved by a call-by-reference parameter.

A call-by-reference parameter is a parameter that updating a value to the parameter updates to the argument. The argument passed to a call-by-reference parameter should be a variable. Neither a function call nor a constant value nor an expression is allowed to be an argument to a call-by-reference parameter.

To declare a parameter as a call-by-reference parameter is to put an ampersand (&) before the parameter name. Refer to Example 18. There is an ampersand before x, which is a parameter of function max_min, on line 1; x is a call-by-reference parameter. So is y on line 1. Parameters a, b, and c are all call-by-reference parameters, too.

The swap function is implemented to exchange values on parameters x and y. Since both x and y are call-by-reference parameters, changing value to x or y causes the corresponding arguments be changed, too. Therefore, the function call on line 20 will exchange a and b’s values; function call on line 25 exchanges b and c’s values.

The max_min function is implemented to sort 3 values. The largest one will be in a, the 2nd largest one will be in b, and the smallest one will be in c. Since all a, b, and c are call-by-reference parameters, values on A, B, and C will be changed if parameter a, b, or c is changed within max_min function. Hence the function call on line 36 will make A own the largest value among A, B, and C; make B own the 2nd largest value; make C own the smallest value among them.

```
 1  swap(&x, &y)
 2  {
 3     var t;
 4
 5     t = x;
 6     x = y;
 7     y = t;
 8  }
 9
10  max_min(&a, &b, &c)
11  {
12     var u;
13
14     do
15     {
16        u = 0;
17
```

```
        if (a < b)
        {
            swap(a, b);
            u = 1;
        }
        if (b < c)
        {
            swap(b, c);
            u = 1;
        }
    } while (u != 0);
}

main()
{
    var A = 10, B = 30, C = 25;

    print("A = ", A, ", B = ", B, ", C = ", C, "\n");
    max_min(A, B, C);
    print("After processing:\n");
    print("A = ", A, ", B = ", B, ", C = ", C);
}
```

Example 18 – p6b0.pnt

Output of Example 18 is as the following.

```
A = 10, B = 30, C = 25
After processing:
A = 30, B = 25, C = 10
```

6.2.5 Passing Partial Array

Peanut allows partial array being passed to a function. In a Peanut script, an array sub-range expression is used to specify the portion of a specific array dimension being passed. An array sub-range expression is as the following syntax

sidx_expression .. eidx_expression

where sidx_expression is an expression standing the starting index of the specific dimension and eidx_expression is an expression meaning the ending index of the dimension; between the two expressions is a 2-dot symbol, where the 2 dots cannot be separated.

Say array variable ARY is a 3x4x5 array. The below are illustrations partial arrays of ARY. They can be arguments in function calls in order to pass partial arrays to functions.

ARY[1..2][0..3][3..3]

ARY[x..x+1][y..y+1][z+1..z+2]

ARY[min(a,b) .. max(a,b)]

The following Example 19 code utilizes passing partial array iteratively to quick_sort function to sort numbers in an array. Refer to line 54 and 59.

```
 1  const NUM = 9973;
 2  const A   = 1234;
 3  const B   = 5678;
 4
 5  partition(n, &num[])
 6  {
 7     var pv;
 8     var i, j;
 9     var temp;
10
11     pv = num[n - 1];
12     i = -1;
13     j = n - 1;
14     while (1)
15     {
16        do
17        {
18           ++i;
19        } while (i < n-1 && num[i] < pv);
20
21        do
22        {
23           --j;
24        } while (j > 0 && num[j] > pv);
25
26        if (i >= j)
27           break;
28
```

```
        temp = num[i];
        num[i] = num[j];
        num[j] = temp;
    }

    temp = num[i];
    num[i] = num[n-1];
    num[n-1] = temp;

    return (i); //the one that has been sorted
}

quick_sort(n, &num[])
{
    var temp;
    var p;

    if (n <= 4)
    {
        insert_sort(n, num);
        return;
    }

    p = partition(n, num);
    if (p > 1)
        quick_sort(p, num[0 .. p-1]);

    //num[p] is sorted

    if (n - p - 1 > 1)
        quick_sort(n - p - 1, num[p + 1 .. n-1]);
}

insert_sort(n, &num[])
{
    var min;
    var i;

    if (n <= 1)
        return;

    min = 0;
    for (i = 1; i < n; ++i)
        if (num[i] < num[min])
            min = i;

    i = num[min];
    num[min] = num[0];
    num[0] = i;

    insert_sort(n - 1, num[1 .. n-1]);
}

main()
{
    var numbers[NUM];
    var x, y;

    //scramble numbers (0 to NUM-1) into array
    for (x = 0; x < NUM; ++x)
    {
        y = (A * x + B) % NUM;
        numbers[y] = x;
    }

    quick_sort(NUM, numbers);

    for (x = 0; x < NUM; ++x)
    {
        print("numbers[", x, "] = ", numbers[x]);

        if (numbers[x] != x)
            print(" != ", x, " (NG)\n");
        else
            print("\n");
    }
```

```
}
```

Example 19 – p6c0.pnt

6.2.6 Return Value

A return statement, which can be one of the followings, is used to feedback an execution result to where the function is called.

```
return   expression   ;
return   ;
```

With an expression follows the keyword return, the execution result of the expression is feedbacked.

Without expression follows the keyword return, 0 is returned.

And explicitly, any function returns the latest operation result to the caller if the function is not terminated by a return statement.

For the following as an example, the `return (5 + 6);` statement in function A() explicitly returns 11, the sum of 5 and 6. Hence, we get `A() = 11` output.

The statement `return;` in function B() in the result produces `B() = 0` as the output.

Function C() has no return statement. The latest operation is the concatenation of string "`str`" and string "`ing`". Therefore, we see `C() = string` is outputted.

The execution result of the condition expression at line 18 of function D() is 0; therefore line 19 statement will not be executed. Hence the condition expression at line 18 is the latest operation of function D(); eventually we see `D() = 0` is outputted.

Line 27 is the final statement that function E() does. So, we get `E() = 11`.

```
1  A()
2  {
3      return (5 + 6);
4  }
5
```

```
B()
{
    return;
}

C()
{
    "str" + "ing";
}

D()
{
    if (A() < B())
        return 100;
}

E()
{
    if (A() < B())
        return 100;
    else
        A();
}

main()
{
    print("A() = ", A(), "\n");
    print("B() = ", B(), "\n");
    print("C() = ", C(), "\n");
    print("D() = ", D(), "\n");
    print("E() = ", E(), "\n");
}
```

Example 20 – p6d0.pnt

Output:

```
A() = 11
B() = 0
C() = string
```

```
D() = 0

E() = 11
```

6.2.7 `main()` Function

A Peanut script starts from the `main()` user function. The main() function should be provided in a script; otherwise, the interpreter stops after parsing file(s).
Two types of main() function APIs are supported: main() and main(argc, argv[]).
While the API is main(argc, argv[]) is implemented, the Peanut interpreter would pass the number of arguments and a flexible-size array of strings to parameters argc and argv respectively. Arguments passed to the main() function includes all values appears after pni.exe invoked in the command line.

6.2.8 Limitation on Calling User Functions

In a user function call, no user function calls can exist in the arguments of the function.

This is a known issue.
For example, it is not allowed to have funcB(1, 2, 3) while calling to funcA.

```
    funcA(123, funcB(1, 2,3), 0);
```

The workaround is to save the result funcB(1, 2, 3) in a variable, and put that variable in the place where funcB(1, 2, 3) is; like the following:

```
    V1 = funcB(1, 2, 3);
    funcA(123, V1, 0);
```

7 Expression, Operator, and Statement

Expressions are calculation details in a Peanut script.
Most expressions consist of one or many operators and operands. Operands in an expression could be literal constants, named constants, variables, function references, and expressions. A single operand is treated as an expression, too. And data types of the operands should be compliant to what the operators require.
Refer to the following table for the operator list, precedence, associativity, and usage description. Operators in the row with smaller number (No.) have a higher precedence; operators in the same row have same precedence and they are evaluated by the associativity rule if they appear in same expression.

No.	Operators	Associativity	Description
1	() [] ++ --	Left to right	Postfix of a function or variable name. () is used after a function name to call a function. [] is used after a variable name to refer to array element(s). ++ -- are used after a variable name to increase or decrease by 1 after the variable value is retrieved (evaluated).
2	+ - ! ~ ++ --	Right to left	Unary operators operating on their right-hand-side operands. + is used as a sign on its number operand. - is used to negate on its number operand. ! is used to do a logical NOT on its operand. ~ is used to do a bitwise complement on its integer operand. ++ is used to increase a variable, whose current value is a number, by 1. -- is used to decrease a variable, whose current value is a number, by 1.
3	* / %	Lelf to right	Do multiplication, division, modulation arithmetics. * is used to multiply two numbers. / is used to do division on two numbers. % is used to do modulation on two integers.

4	+ -	Left to right	Do addition and subtraction arithmetics. + is used to add two numbers. - is used to do subtraction on 2 numbers.
5	<< >>	Left to right	Do left shift and right shift operations on an interger with some bits. << is used to do left shift operation on its left operand, which should be an integer, for some bits specified by the right operand, which is an integer, too. >> is used to do right shift operation on its left operand, which should be an integer, for some bits specified by the right operand, which is an integer, too.
6	> >= < <=	Left to right	Do relational greater-than operation, greater-than-or-equal-to operation, less-than operation, and less-than-or-equal-to operation on two numbers. A logic result 0 or 1 is got.
7	== !=	Left to right	Do equality and inequality comparisons on two operands. A logic result 0 or 1 is got.
8	&	Left to right	Do bitwise AND operation on two integers.
9	^	Left to right	Do bitwise EXCLUSIVE-OR operation on two integers.
10	|	Left to right	Do bitwise OR operation on two integers.
11	&&	Left to right	Do logical AND operation on two operands. Note that if the left-hand-side operand is evaluated as value 0 (or 0.0), the right-hand-side operand will not be evaluated.
12	||	Left to right	Do logical OR operation on two operands. Note that if the left-hand-side operand is evaluated as a non-zero value, the right-hand-side operand will not be evaluated.
13	?:	Right to left	Do a conditional operation on 3 operands. The syntax is like opd1 ? opd2 : opd3. If opd1 is evaluated as a non-zero value, opd2 is evaluated as the result of this operation;

			otherwise, opd3 is evaluated as the result of this operation.
14	= += -= *= /= %= &= ^= |= <<= >>=	Right to left	= is used to assign the right-hand-side evaluation result to the left operand, which should be a variable. Other operators are used to do corresponding arithmetic operations on the two operands and assign the operation results to the left-hand-side operand, which should be a variable.
15	, (comma)	Left to right	To form an expression list with multiple expressions. The evaluation result of an expression list is that of the right most operand. All operands are evaluated.

7.1 Statement

Most statements are formed by key words.
Besides key words var and const, Peanut uses the following key words to form richer statements: switch, case, default, exit, return, continue, break, if, else, do, while, for, namespace, struct.

7.1.1 Expression statement

An expression statement is formed by adding a semi-column at the end of an expression as the following syntax.

```
expression;
```

See the following Example 21 as an example. On line 16 to 20 are expression statements.

```
1  function1()
2  {
3      return (1);
4  }
5
6  main()
7  {
    var a = 1;
    var b;
    var ARY[3];

    ARY[0] = 1;
    ARY[1] = 3;
    ARY[2] = 7;

    function1();
    ARY[1];
    a++;
    ARY[2]--;
    b = a++ + - ++ARY[1];

    print("a = ", a, "\n");
    print("b = ", b, "\n");
    print("ARY[0] = ", ARY[0], "\n");
    print("ARY[1] = ", ARY[1], "\n");
    print("ARY[2] = ", ARY[2], "\n");
}
```

Example 21 - p710.pnt

7.1.2 Switch Statement

Syntax:

```
switch (expression1)
{
case constant_value1:
   statements1
case constant_value2:
case constant_value3:
   statements2
default:
   statements3
}
```

The expression1 expression should be enclosed by a pair of parenthesis.

The evaluation result of expression1 can’t be a floating-point number; otherwise, a run-time error will be reported.

7.1.3 Exit Statement

Syntax of an exit statement is as follows:

```
exit()    ;
```

An exit statement is used to terminate evaluation of the script.

7.1.4 Return Statement

A return statement is used to terminate current function execution and resumes the execution of the caller function at the place immediately after where this function is called.
In a Peanut script, any return statement returns a value even if there is no value specified for returning. Without specifying a returned value, Peanut returns 0 in default.
The syntax rules of a return statement is one of the followings:

```
return    ;
```

```
return    expression    ;
```

7.1.5 Continue Statement

Syntax of a continue statement is:

```
continue    ;
```

A continue statement is used within a loop.
The statement forces to jump to the end of the loop and restarts next iteration of the loop.

7.1.6 Break Statement

Syntax:

```
break    ;
```

A break statement is used within a loop statement or a switch statement.
The statement terminates evaluations of the closest loop statement or the closest switch statement.

7.1.7 Empty Statement

Syntax:

```
 ;
```

A empty statement is a statement doing nothing.

7.1.8 Compound Statement

A compound statement is a statement using a pair of braces enclosing any number of statements.

```
{
}
```

```
{
        statement1
        statement2
        …
}
```

7.1.9 If Statement

Syntax of a if statement is one of the followings:

```
if   (   expression1   )
        statement1
```

```
if   (   expression1   )
        statement1
else
        statement2
```

Let’s call the above statement1 as if-body and statement2 as else-body.
Either if-body or else-body should be a form of a single statement. If multiple

statements should be run in if-body or else-body, the programmer should write these statements in a compound statement.

7.1.10 For Statement

Syntax:

```
for   (   iexpression   ;   cexpression   ;   pexpression   )
      statement1
```

statement1 is called for-body. Similar to if-body or else-body, a for-body should be the form of a single statement. If multiple statements would be run in the for-body, the programmer should write the statements in a compound statement.
A for statement is a kind of loop statement. Both break and continue statements are allowed in a for-body.
The execution sequence of a for statement is: iexpression → cexpression → statement1 → pexpression → cexpression → statement1 → pexpression → cexpression → statement1 → pexpression → cexpression …
A for statement is terminated once cexpression is evaluated as 0 or a break statement within the for-body is executed.
All iexpression, cexpression, pexpression are optional. If iexpression is blank, execution of iexpression will be skipped from the above execution sequence of a for statement. So are cexpression and pexpression.
cexpression is short for condition expression. If cexpression is evaluated as non-0, statement1 is executed.

7.1.11 While Statement

```
while   (   cexpression   )
      statement1
```

statement1 is called while-body and should be in the form of a single statement. To have multiple statements executed in a while-body, use a compound statement.
Execution sequence of a while statement is: cexpression → statement1 → cexpression → statement1 → cexpression … A while statement is terminated once the cexpression is evaluated as 0 or a break statement in the while-body is executed.
Like a for statement, the cexpression in a while statement is also a condition expression, which determines to execute statement1 or terminate the while statement.

7.1.12 Do Statement

```
do    statement1    while    (    cexpression    )    ;
```

A do statement is similar to a while statement except the execution sequence of the body and the conditional expression. The execution sequence of a do statement is: statement1 → cexpression → statement1 → cexpression → … The do statement stops while cexpression is evaluated as 0 or a break statement in do-body is executed.

8 Structure

In short, a structure is a user-defined data type with compound datum.
Here, the term compound datum means that data of a structure may be composed of any number variables can. Besides datum, functions can be defined within a structure, too.
The syntax of a structure is:

```
struct   sname
{
      m_datum
      m_functions
}   ;
```

A structure begins with keyword struct and a name followed by a pair of braces and a semicolon. Within the braces, member datum and member functions are declared, where all member datums (m_datum) should be declared before member functions (m_functions). Both m_datum and m_functions are optional! Though it is valid to declare an empty structure without any member data or function, an empty structure is meaningless. Each variable within a structure is called a field and the variable name of a structure is called a field name.
Let's take a quick look at an example of structure definition and usages. The following Example 22 is an example defining a structure with 2 member datum and 3 member functions. On line 1, the programmer defines a structure named Point. On line 2, x and y are the member datum of the structure and from 4 to line 32 are the 3 member functions.

```
 1  struct  Point   {
 2      var x, y;
 3
 4      init(xi, yi)
 5      {
 6          x = xi;
 7          y = yi;
 8      }
 9
10      quadrant()
11      {
      if (x >= 0)
      {
         if (y >= 0)
            return (1);
         else
            return (4);
      }
      else
      {
         if (y >= 0)
            return (2);
         else
            return (3);
      }
   }

   vect2Point(struct Point P, struct Point &V)
   {
      V.x = P.x - x;
      V.y = P.y - y;
   }
} ;

main()
{
   struct Point Q, R;
   struct Point V;

   Q.x = -2;
   Q.y = -3;
   R.init(5, 6);

   print("Point Q is at quadrant ", Q.quadrant(), "\n");
   print("Point R is at quadrant ", R.quadrant(), "\n");

   Q.vect2Point(R, V);
   print("Vector from Q to R is (", V.x, ", ", V.y, ")\n");

```

```
 50     R.vect2Point(Q, V);
 51     print("Vector from R to Q is (", V.x, ", ", V.y, ")");
 52  }
```

Example 22 – p810.pnt

On line 37 and 38 within the main function, three local variables (Q, R, and V) are declared with datatype as struct Point. Variables defined as a structure type are called structure variables or objects.

Accessing a member data of an object is achieved by referring to the field name with the object name as a prefix and a dot symbol (.) standing between them. For example, object Q's field x is assigned with -2 on line 40; object Q's field y is assigned with -3 on line 41. Other examples accessing member datum can be seen on line 48 and 51.

Calling a member function of an object is achieved by similar approach as accessing a member data of the object. The object name and the dot symbol are prefixed before a function-call to the member function. For the example on line 42, member function init is called with arguments 5 and 6 and the prefix "R.". And other member function-calls can be seen on line 44, 45, 47, and 50.

8.1 Accessing to Member Datum in A Member Function

A member data of a structure can be directly accessed by referring to its name in the structure's member functions.

In Example 22, both variables x and y in member functions init, quadrant, and and vect2Point are neither local variables nor global variables. Where are x and y indeed? We know that x and y are member datum of structure Point. Hence x and y in member functions init, quadrant, and vect2Point are member datum of Point objects.

On line 44 of Example 22, member function quadrant is called with prefix "Q.", which implicitly "tells" quadrant function to access Q object's fields x and y while quadrant function accesses to variable x and y in its function body. So R.quadrant() expression on line 45 guides quadrant function to access R object's fields x and y within quadrant's function body.

8.2 Array Field

Peanut allows programmers to declare array variables as structure fields.

8.3 Nested Structure

Peanut allows programmers to declare structure variables as structure fields within structures. We call this kind of declaration that a structure member data exists within another structure is a nested structure. For Example 23 as an example, Point is declared as a structure with member data x and y, and Line is declared as a structure with data P and Q, where type of P and Q is Point. Structure Line is a nested structure.

```
struct  Point   {
    var x, y;

    init(xi, yi)
    {
        x = xi;
        y = yi;
    }

    quadrant()
    {
        if (x >= 0)
        {
            if (y >= 0)
                return (1);
            else
                return (4);
        }
        else
        {
            if (y >= 0)
                return (2);
            else
                return (3);
        }
    }

    vect2Point(struct Point P, struct Point &V)
    {
        V.x = P.x - x;
        V.y = P.y - y;
    }
} ;

struct Line {
    struct Point P;
    struct Point Q;

    init(struct Point p1, struct Point p2)
    {
        P.init(p1.x, p1.y);
        Q.init(p2.x, p2.y);
    }

    len()
    {
        struct Point V;
        var LL, Lh, Ll, Lm, i;

        P.vect2Point(Q, V);
        LL = V.x * V.x + V.y * V.y;

        Lh = LL;
        Ll = 0;
        for (i = 0; i < 30; ++i)
        {
            Lm = (Lh + Ll) / 2.0;
            if (Lm * Lm > LL)
                Lh = Lm;
            else
                Ll = Lm;
        }

        return (Lm);
    }
} ;

main()
{
    struct Point Q, R;
    struct Point V;
    struct Line L;

    Q.x = -2;
    Q.y = -3;
    R.init(5, 6);

    print("Point Q is at quadrant ", Q.quadrant(), "\n");
    print("Point R is at quadrant ", R.quadrant(), "\n");

    Q.vect2Point(R, V);
    print("Vector from Q to R is (", V.x, ", ", V.y, ")\n");

    R.vect2Point(Q, V);
    print("Vector from R to Q is (", V.x, ", ", V.y, ")\n");

    L.init(Q, R);
    print("length of L, from Q to R, is about ", L.len());
}
```

Example 23 - p820.pnt

Output of the Example 23

```
Point Q is at quadrant 3
Point R is at quadrant 1
Vector from Q to R is (7, 9)
Vector from R to Q is (-7, -9)
length of L, from Q to R, is about 11.401754
```

8.4 Member Data Resolution in A Nested Structure Variable

As we know that accessing a member data of an object is achieved by referring to the field name with the object name as a prefix and a dot symbol (.) standing between them. To access a field in a nested structure is achieved by referring the object name first and a series of dot symbol and member data name. Hence accessing the member data x of Q Point in Line L be accessed in main function of Example 23 is expressed as L.Q.x. The expression to access a member data of a nested structure variable is called member data resolution.

Note to specify array indexes after the member data name in the member data resolution if the member data is declared as an array.

9 Namespace

Namespace is a mechanism for programmers to devise scripts with higher-level modualization and avoiding name conflicts.

Syntax of a namespace statement is:

```
namespace    name
{
      declarations
}
```

A namespace is defined by keyword namespace followed by a unique name and after that a pair of braces is used to enclose the “contents” of namespace. Functions, constants, global variables, and structures can be declared in a namespace.
Example 24 is an example script on which two namespaces are declared. In namespace NS1, a constant Id (aString) and a function (output) are declared. In namespace NS2, a variable (aString) and a function (output) are declared.

```
namespace NS1
{
    const aString = "Hello";

    output()
    {
        print(aString, "\n");
    }
}

namespace NS2
{
    var aString = "World";

    output()
    {
        print(aString, "\n");
    }
}
```

```
20
21  main()
22  {
23      NS1::output();
24      NS2::output();
25      NS2::aString = NS1::aString;
26      NS2::output();
27  }
```

Example 24 - p910.pnt

9.1 Namespace Access

“Contents” declared within a namespace are in fact in the global scope. That is to say that contents declared within a namespace can be accessed anywhere.
To access one content (constant, global variable, function, structure) declared in a namespace, the namespace name should be specified besides the name of the content; between namespace name and the name of the content is a namespace delimiter which is composed with two contiguous semicolons. Please refer to Example 24. On line 23 the function output in namespace NS1 is called. On line 24 and 26, function output in namespace NS2 is called. And on line 25, value of the constant aString in namespace NS1 is assigned to variable aString in namespace NS2.

9.1.1 Access to A Constant

Syntax to access a constant-id declared in a namespace.

```
ns_name::cid
```

9.1.2 Access to A Variable

Syntax to access to a variable declared in a namespace.

```
ns_name::v_name
```

9.1.3 Function Call

Syntax to call a function declared in a namespace.

```
ns_name::f_name(arg_list)
```

9.1.4 Structure Type

Syntax to refer to a structure name declared in a namespace.

```
struct ns_name::s_name
```

9.2 Splitting Namespace

It is not necessary to put all the namespace contents within a single namespace declaration statement. One can also split contents of a namespace into groups and put each group of contents into different namespace statements with the same namespace name.

Refer to the Example 25. Namespace foo is split into two groups and are declared on line 1 and 10 respectively.

```
1  namespace foo
 2 {
 3     const C1 = 100;
 4     const S1 = "string1";
 5
 6     var f1;
 7     var f2;
 8 }
 9
10 namespace foo
11 {
12     dumpConsts()
13     {
14         print("C1 = ", C1, "\n");
15         print("S1 = ", S1, "\n");
16     }
17
18     dumpVars()
19     {
20         print("f1 = ", f1, "\n");
21         print("f1 = ", f1, "\n");
22     }
23 }
```

Example 25 - p920.pnt

9.3 Global Namespace

Take a look at Example 26. There are two prnt functions declared; one is declared on line 8 in namespace N1, and the other is defined on line 1 outside namespace N1. We see that function foo calls to prnt function on line 16. Which prnt function is called by the statement on line 16 indeed?
There is another function foo on line 20 calls to prnt function. Again, we wonder which prnt function is called, line-1 prnt function or line-8 prnt function, indeed?

```
 1  prnt(p1, p2)
 2  {
 3     print(p1, " ", p2, "\n");
 4  }
 5
 6  namespace N1
 7  {
 8     prnt(p1, p2)
 9     {
10        print("p1 = ", p1, "\n");
11        print("p2 = ", p2, "\n");
12     }
13
14     foo(x, y)
15     {
16        prnt(x, y);
17     }
18  }
19
20  foo(x, y)
21  {
22     prnt(x, y);
23  }
```

Example 26 - p930.pnt

The answers to the above two questions are: the line-16 statement calls to line-8 prnt function and the line-22 statement calls to line-1 prnt function.
Let's ignore searching rules for a function call here tentatively, but let's focus on

global scope and namespaces in Peanut.
In Peanut, anything in the global scope in fact belongs to a namespace. And anything defined outside any namespace belongs to the default namespace, global namespace. Function prnt on line 1 and function foo on line 20 in fact are in the global namespace.
To access contents in the global namespace is achieved by putting the namespace delimiter before the name of the global variable, or function, or constant-id, or structure.
Therefore, Example 26 can be rewritten as Example 27 with putting a namespace delimiter before prnt.

```
 1  prnt(p1, p2)
 2  {
 3     print(p1, " ", p2, "\n");
 4  }
 5
 6  namespace N1
 7  {
 8     prnt(p1, p2)
 9     {
10        print("p1 = ", p1, "\n");
11        print("p2 = ", p2, "\n");
12     }
13
14     foo(x, y)
15     {
16        prnt(x, y);
17     }
18  }
19
20  foo(x, y)
21  {
22     ::prnt(x, y);
23  }
```

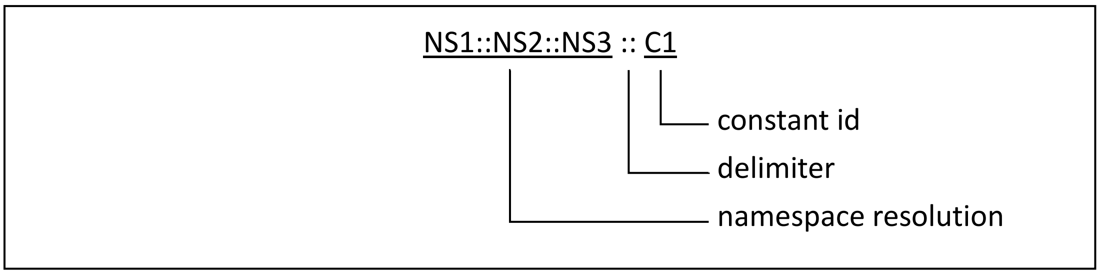
Example 27 - p940.pnt

9.3.1 Namespace of Built-In Functions

All built-in functions introduced in chapter 6 belong to global namespace.
Function calls to built-in functions can be explicitly expressed with the namespace delimiter before the built-in function name.

9.4 Nested Namespace and Namespace Resolution

Either in Example 26 or Example 27, namespace N1 is defined in "the same level" as that of prnt function on line 1. Indeed, namespace N1 belongs to the global namespace, too.
At the beginning of this chapter, it says that a namespace contents can be functions, constants, global variables, and structures. Now, we say that a namespace contents can be functions, constants, global variables, structures, and namespaces. Within a namespace, one can define namespaces.
To access, for example, a constant id in a nested namespace, one should specify the namespace resolution, the constant id, and a namespace delimiter between the namespace resolution and the constant id. The namespace resolution is composed of all the namespace names from top to bottom separated by namespace delimiters. The format of the reference to a constant id in a nested namespace looks like the following



Refer to Example 28. In main function, it outputs constants C1's defined in (nested) namespaces.

```
namespace NS1
{
    const C1 = 3;

    namespace NS2
    {
        const C1 = 6;

        namespace NS3
        {
            const C1 = 9;
        }
    }
}

main()
{
    print(NS1::C1, "\n");
    print(NS1::NS2::C1, "\n");
    print(NS1::NS2::NS3::C1);
}
```

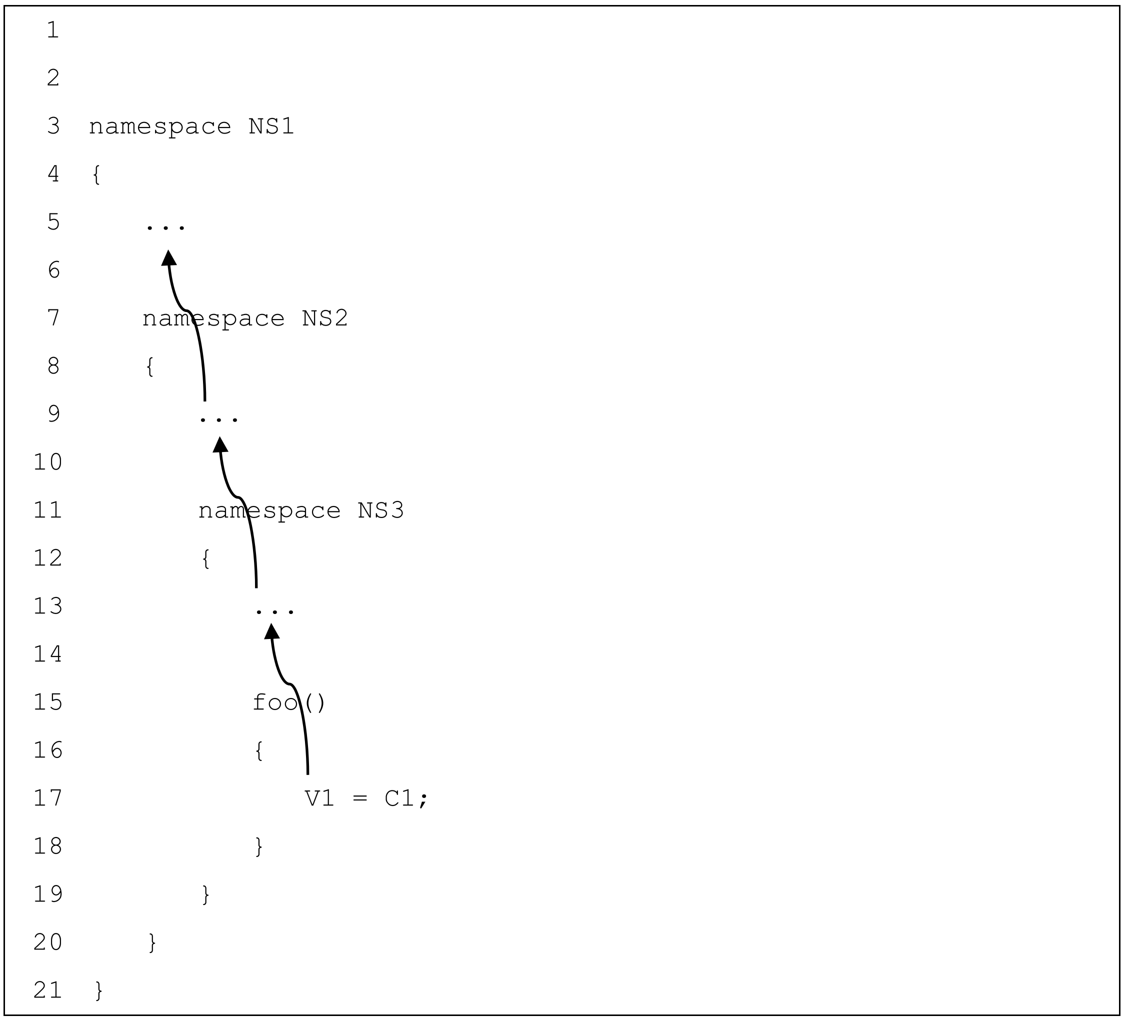
Example 28 - p950.pnt

9.5 Rules to Solve Symbols in Namespaces

In Example 27, there are two prnt functions implemented on line 1 and line 8. We see that both function calls on line 16 and line 22 have no namespace resolution specified. Peanut matches the function call on line 16 to the line-8 prnt function, and matches that on line 22 to line-1 prnt function. This result, line-16 statement calling to line-8 prnt function and line-22 calling to line-1 prnt function, seems intuitive, but what are the rules to solve symbols (function names, constant Ids, global variables, structure names) in namespaces?

Basically, there are two rules to solve symbols references in Peanut scripts. Rule1: with namespace resolution specified, Peanut looks up the symbol into the namespace described by the namespace resolution. Rule2: without namespace resolution specified, Peanut first looks up the symbol from the namespace in which the symbol reference is; if the symbol is found, the symbol is solved; if the symbol isn’t found in that namespace, Peanut looks up the symbol from the upper-level namespaces iteratively until the symbol is found or is not found in the global namespace.

Rule2 can be visualized as the below diagram. Both V1 and C1 are referred without specifying namespace resolution. Peanut looks up V1 in namespace NS3 first. If V1 is declared in NS3, then V1 on line 17 is solved. If V1 is not found in NS3, Peanut continues to search for V1 in namespace NS2 and searches for V1 in namespace NS1 and the global namespace iteratively.



```


namespace NS1
{
    ...

    namespace NS2
    {
        ...

        namespace NS3
        {
            ...

            foo()
            {
                V1 = C1;
            }
        }
    }
}
```

Actual rules to match a function call, a constant Id, or a variable without namespace resolution specified might be more complex than Rule2. And refer to the following sub sections for details.

9.5.1 Solve Function Calls without Namespace Resolutions

If the function call (without namespace resolution) A exists in a member function B which belongs to structure S, Peanut looks up function A in structure S first. If function A is not implemented in structure S, Peanut continues to search for function A from namespaces by Rule2.
If the function call exists in a non-member function, Peanut search for function A from namespaces by Rule2 directly.

9.5.2 Solve Variables and Constant Ids without Namespace Resolutions

Say a non-constant operand A in an expression exists in function F. Peanut first looks up A from local variables of function F. If A is not found, Peanut looks it up from the parameters of function F. If A is still not found, Peanut looks A up from the member data of the structure to which function F belongs. If A is still not found from the structure's member data or F is not a member function, Peanut looks up global variables and constant Ids from namespaces by Rule2.

10 Preprocessor

10.1 #include

To physically include the file contents into the native file.
Syntax

```
#include    <file_name_with_path>
```

10.2 #load

To load an extension module, which is a dynamic link library or a share object.
Syntax

```
#load    <dll_file_name_with_path>
```

10.3 __FILE__

The name of the file in which __FILE__ exists. It is a string.

10.4 __PATH__

The name of the path (directory) in which the script file exists. It is a string.

10.5 __FUNCTION__

The name of the function in which __FUNCTION__ exists. It is a string.

10.6 __LINE__

The line number on which __LINE__ exists in the file. It is an integer.

11 Extension

Peanut is devised for programmers to compose scripts cooperating with C code easily. The C code shall be compiled into dynamic link libraries with a special initialization function. We call such a dynamic link library is an extension module of Peanut.

The extension module is imported into Peanut by #load directive. An extension module can create namespaces, global variables, constant Ids, and functions for scripts.

The following 3 files should be included in your C program of the extension module:

    peanut.h
    peanutType.h
    peanutExtend.h