The Pooh Language

... In which we talk to the computer, and it talks back to us.

Introducing the Pooh programming language for kids and grown ups

Expressing data

In which we talk about the kinds of data that the computer knows about, and how to name the data so that it can be of use to us.

Variables

Variables are places in the computers memory where values are stored; In a program, each variable has a name, a variable name must be one or more characters in length and must begin with a letter or underscore. The characters allowed in variable names are letters, numbers and the underscore sign
 	
Some examples:
Valid variable names: X , A9 , Not_equal
Invalid variable names: _a , 9b , another%bug
	Example: **var-env.p**
	a = 2
	b = a * a
	c = a + b
	
	`pooh -x var-env.p`

A variable is defined when it is assigned a value; this happens when the variable name is placed left of the = sign and the value that is assigned to the variable is placed to the right of the = sign. See example var-env.p

```
001|a = 2
002|b = (a:2 * a:2):4
003|c = (a:2 + b:4):6
```

When a variable stands to the right of the = sign, its value is looked up and used. It is an error to use a variable that is not defined. See example var-inv.p

Example: var-inv.p

```
b = a * a
```

```
pooh  var-inv.p
Error: variable a is used without
having been assigned a value
1. |b = a * a
   |....^
```

The following names are reserved words and may not be used as variable names, these names are reserved for other purposes.

```
and break byref elseif elsif end eq false False
for ge gt if include lg loadextension loop ne
next null Null not optional or return sub true
True while
```

Numbers and numeric expressions

A floating point number is expressed as a base number (mantissa) times 10 raised to some power (exponent)

Number = Mantissa x 10^{Exponent}

Number	*Expressed as*
3.245×10^{4}	3.245E4
-42×10^{22}	-42E22 or -0.42E24
5.6×10^{-4}	5.6E-4 or 0.56E-5

A floating point constant is expressed by first writing the Mantissa then E and then the exponent

A numeric expression performs a calculation on numbers, the expression returns the result of the calculation.

operators that take two arguments are

- `a + b` add the value of a to b
- `a - b` substract the value of b from a
- `a * b` multiply the value of a times value of b
- `a / b` divide the value of a a by b
- `a ^ b` returns a raised to the power of b
- `a % b` returns the remainder of dividing a by b.

Parenthesis can be used to group operations; an expression within parenthesis is computed first, before computing the result outside of the parenthesis. See example **var-par.p**

Example: **var-par.p**

```
a = 2
b = 3
c = (a + 1) * (b + 3) / ( a * b)
```

```
pooh -x var-par.p
001|a = 2
002|b = 3
003|c = (((a:2 + 1):3 * (b:3 + 3):6):18 /
(a:2 * b:3):6):3.000000e+00
```

Strings

A string is a character sequence of any length. A string constant is enclosed in ' quote characters. Space characters are part of the string. Strings are values and can be assigned to variables.

A string constant can span multiple lines.

Two or more string or integer values can be joined into one string by means of the .. operator. The .. operator has two arguments, the value of the argument to the right of the operator is appended to the value of the argument that stands left of the operator.

One can join multiple values, If one writes

```
    a = b .. c .. d .. e
```

than that is equivalent to

```
    a = ( ( b .. c ) .. d ) .. e )
```

See example var-s1.p – Note that println line takes the value of e and prints it on the screen.

A string can have embedded expressions. The text that is enclosed between [and] brackets is an expression, the value of the expression is computed and the result is inserted into the resulting string.

For example the expression

```
  a = 'string area of [ a ^ 2] square meters'
```

is logically to

```
  a  = 'a string area of '  .. (a ^ 2 ) .. ' square meters'
```

The brackets are a convenient shorthand for joining strings and expressions. See example var-ex.p – Note that println line takes the value of e and prints it on the screen.

If you want to write text that contains a single opening or closing bracket, then the bracket would be misinterpreted as the start of an embedded expression. To avoid this use two ' colon delimiters for the start and end of a string. In this case the start of an embedded expression is [[and the end of an embedded expression is]]

Because of this rule, it is difficult to express an empty string.The function emptystring() returns the value of an empty string. The empty string is 0 characters in length. See example **var-empty.p**

Example string

```
string_variable = 'hello world'

a = 'a string spanning
two lines'
```

Example: var-s1.p

```
a = 'words '
b = 'form  '
c = 'sentences '
d = 42
e = a .. b .. c .. d
println(~msg e )
```

pooh -x var-s1.p

```
002|a = 'words '
003|b = 'form '
004|c = 'sentences. '
005|d = 42
007|e = a:'words ' .. b:'form ' .. c:'sentences. ' ..
d:42
008|println( ~msg e:'words form sentences.
42' )...
words form sentences. 42
```

Example: var-ex.p

```
a = 3
b = 'Square with side [ a ] has an area of [ a ^ 2]
square meters'
println( ~msg b )
```

pooh -x var-ex.p

```
001|a = 3
002|b = 'Square with side ' .. a:3 .. ' has an area
of ' .. (a:3 ^ 2):9 .. ' square meters'
003|println( ~msg b:'Square with side 3 has an
area of 9 square meters' )...
Square with side 3 has an area of 9 square
meters
```

```
a =  ''string with [ bracket ''
a  = '' string with [ bracket [[ 42 ]] ''
```

Example var-empty.p

```
empty = emptystring()
println( ~msg 'the empty string is [ size( ~arg
empty ) ] characters long' )
```

pooh -x var-empty.p

```
001|... = emptystring(  )...
001|empty = emptystring(  ):''
002|println( ~msg 'the empty string is ' ..
```

Another function is newline() it returns the delimiter string between lines of text.

```
size( ~arg empty:'' )...
002|println( ~msg 'the empty string is ' ..
size( ~arg empty:'' ):0 .. ' characters long' )...
the empty string is 0 characters long
```

Relational expressions - Comparing numbers

Relational expressions check if a specific statement is true or not true; For example the relation expression a < b checks if variable a holds a number that is smaller then number b.

The result of a relational expression is itself a number 1 if the relation is true (in the above example: the value of a is indeed smaller then b) and 0 if the relation is false (the value of a is bigger or equal to b);

Relation expressions are used in statements such as the If statement. In the example stmt-if.p the If statement checks if the value of a is smaller then b, if this condition is true the string 'The value of a is smaller then b' is shown; if the condition is false, then the string 'The value of a is bigger or equal to a'

There are the following kinds of relational expressions between numbers

a == b	The number value of a is equal to b
a != b	The number value of a is not equal to b
a < b	The number value of a is smaller than b
a <= b	The number value of a is smaller or equal to b
a > b	The number value of a is bigger than b
a >= b	The number value of a is bigger or equal to b

If a or b holds a string value, then this string is converted into a number, before the expression is checked.

Example: stmt-if.p

```
a = 42
b = 2
if a < b
  println( ~msg 'The value of a
is smaller than b' )
else
  println( ~msg 'The value of a
is bigger of equal to a' )
end
```

```
pooh -x var-s1.p
001|a = 42
002|b = 2
003|if (a:42 < b:2):false
004|else
006| println( ~msg 'The value of
a is bigger or equal to a' )...
The value of a is bigger or
equal to a
006|end # if
```

Relational expressions - Comparing strings

A number may have different annotations; for example the number 1 is equal to the number 1.0 which is equal to the number 1.0E1

However the string values '1' '1.0' and '1.0E1' are all different. This is the reason why we have a different set of relational expressions for strings than for numbers.

There are the following kinds of relational expressions between numbers

a eq b	The number value of a is equal to b
a ne b	The number value of a is not equal to b
a lt b	The number value of a is less than b
a le b	The number value of a is less or equal to b
a gt b	The number value of a is greater than b
a ge b	The number value of a is greater or equal to b

If a or b holds a integer value, then integer value is converted into a string format before the expression is checked.

Arrays

An array contains a numbered sequence of values; think of an array V as owning a sequence of entries: v_1 v_2 v_3 v_4 .... An value of the subscript identifies its value, so by having a sequence number, say 2, you can look up the value v_2 from the array.

An array variable is defined when an array value is assigned to a variable name.

ArrayVariable = []

The value [] is an empty array without any entries.

ArrayVariable = [2, 'aaa', 3]

Here ArrayVariable gets the value of an array, where the first element of the array is the number 2, the second element of the array is the string 'aaa' and the third element is the number 3.

The expression ArrayVariable[1] returns the first element of the array value ArrayVariable – the number 1

The expression ArrayVariable[4] is special, since we did not define an entry with the index 4, the computer can't return its value, instead the expression will return the special Null value; which stands for 'no this value does not exist at all'.

Note the angular brackets [] that are placed around the key index – which identifies the values that is stored in the array.

println(~msg 'The first element is ' .. ArrayVariable[1])

This expression prints out the string 'aaa' the second element

Example var-vec.p

```
array = [ 1, 3, 5, 7, 11 ]

println( ~msg 'the first prime is ' .. array[ 1 ] )
println( ~msg 'the fourth prime is ' .. array[ 4 ] )
println( ~msg 'the fifth prime is ' .. array[ 5 ] )
```

pooh -x var-vec.p

```
001|array = [ 1 , 3 , 5 , 7 , 11]
003|println( ~msg 'the first prime is ' ..
array[1]:1 )...
the first prime number is 1
004|println( ~msg 'the fourth prime is ' ..
array[4]:7 )...
the fourth prime number is 7
005|println( ~msg 'the fifth prime is ' ..
array[5]:11 )...
the fifth prime number is 11
```

of the array.

The first element is 1

Tables

A table owns a set of entries, each entry is identified by its own key value. The key value can be a string, and integer, in fact any value!

An table variable is defined when an table value is assigned to a variable name.

TableVariable = {}

The value {} is an empty table without any entries.

Please look at the example **var-hash.p** . Here the table myTable gets a table value which:
- maps the key 'aaa' to the value 1
- maps the key [1, 2, 3] to the value 222222
- maps the key 42 to a function without a name (also known as anonymous function). This function prints out the string 'hello world', when it is called.

The expression myTable['aaa'] returns the value 1, which is identified by the key 'aaa'.

Note the brackets { } that are placed around the key index – which identifies the values that is stored in the table.

The expression myTable['kuku'] is special, since we did not define an entry with the index 'kuku', the computer can't return its value, instead the expression will return the special Null value; which stands for 'no this value does not exist at all'.

Example: var-hash.p

```
myTable = { 'aaa' : 1,
            [ 1, 2, 3 ] : 222222,
            42 : sub ()
              println( ~msg 'hello world' )
                 end
           }
println( ~msg myTable{ 'aaa' } )
println( ~msg myTable{ [1, 2, 3 ] } )
myTable{ 42 } ()
```

pooh -x var-hash.p

```
001|myTable = { 'aaa' : 1 , [ 1 , 2 , 3]  :
222222 , 42 : sub () }
007|println( ~msg myTable{'aaa'}:1 )...
1
008|println( ~msg myTable{[ 1 , 2 , 3] }:222222
)...
222222
009|myTable{42}:sub (  )...
004| println( ~msg 'hello world' )...
hello world
```

Null values

If the program is looking up a value of an array or table entry, and no entry with the given key exists, then the Null value is returned;

Example: var-null.p

```
array=[]
a = 42
if array[10] != Null
```

The null value is special; it means 'no this value does not exist at all'. One may not add a Null value to a number ;

One can't add a Null value to a number, using the Null value in an arithmetic expression results in an error.

The can test if an expression yields the null value or not; example **var-null.p** first checks that array[10] is not a null value; only if this is not true then it proceeds to use the value in a computation.
This is done by the if statement; the statements within the if statement are done only if the condition (array[10] is not equal to Null) is true.

```
  b = array[10]
  c = a + b
end
```

pooh -x var-null.p

```
001|array = [ ]
002|a = 42
004|if (array[10]:Null != Null):false
006|end # if
```

References to values

Sometimes two different variable name may refer to the same data entry.

Normally the = sign (assignment operator) copies the value of the left hand side into the variable name that stands to the right of the operator sign.

The operator := creates an alias; the variable name to the left of the operator := will refer to the same value as the expression to the right of the operator.

Please see the example var-ref.p. The value a holds the string 'bbb'; the variable b now refers to the same value as owned by the variable a.
When the variable a is assigned a different value (the value 'bbb') the variable b will then refer to the new value of a.

Please note: You can refer to the value of an alias (reference) in the same way as you can refer to the value of a real value!

This is different to most other programming languages, where one has an explicit construct that will look up the value of a reference. In the Pooh language you are spared these confusing details.

Example: `var-ref.p`

```
a = 'aaa'
b := a
a = 'bbb'
println( ~msg b )
```

pooh -x var-ref.p

```
001|a = 'aaa'
002|b := a:'aaa'
003|a = 'bbb'
004|println( ~msg b:->
'bbb' )...
bbb
```

Statements

In which we talk about how to change data, and how to tell the computer to do stuff.

Assignment

The assignment statement allows to give a specific value to variable name.

VariableName = <expressions>

When the program is run, the value of the expression is computed, and because of the assignment statement, the variable VariableName will then refer to this computed value.

If a variable name has never been assigned a value, it is not defined and its value can't be used in a computation. In the pooh language the program will not be able to run at all. You will have to fix this problem and then try run the program again.

If .. elsif .. else

The if statement is used to decide on a course of action in a program; It has the following form.

```
If <expression>
  <one or more statements>
end
```

If the expression is true, that is the value of the expression is not 0, then all statements that follow up until the end keyword are evaluated. The example **stmt-if0.p** shows that off.

```
If <expression>
  <plan A: one or more statements>
else
  <plan B: one or more statements>
end
```

This previous form expresses a choice with an alternative, if the expression is true, that is the value of the expression is not zero, then the set of statements after the if up until the keyword else is evaluated; otherwise, if the expression if false, that it it has the value 0, then the alternative course of action is taken, all statements right after else up until the end keyword are evaluated.

One can also do multiple choices

```
if <expression>
  <planA : one or more statements>
elseif <expression>
```

Example stmt-if0.p

```
a = 42
b = 2
if a > b
  println( ~msg 'The
value of a is smaller
than b' )
  println( ~msg 'and
now lets have a party!'
)
end
```

pooh -x stmt-if0.p

```
001|a = 42
002|b = 2
003|if (a:42 >
b:2):true
004| println( ~msg 'The
value of a is smaller
than b' )...
The value of a is
smaller than b
005| println( ~msg 'and
now lets have a party!'
)...
and now lets have a
party!
005|end # if
```

```
  <planB: one or more statements>
elseif <expression>
   <planC: one or more statments>
else
    <allOtherCases: one or more statements>
end
```

Incidentially one can write both **elsif** and **elseif** are valid keywords that mean the same thing, that's because I often misspell them.

While loops

The while loop looks as follows

```
while <condition>
  <one or more statements>
end
```

If the <condition> is true, that is its value is not zero then the statements from after the condition up to the end keyword are executed. After that condition is checked again, if it is still true, then again the same statements are evaluated, all this continues until the condition is false, its value is false.

The example stmt-while.p computes the sum of squares for all integer numbers from 1 to 7.
First the variable I is set to one; the while statement checks that the value of I is smaller or equal than 7;

```
sum = sum + i * i
```

Here the square of value of i is added to the value of sum. Next the value of I is incremented; this is very important since otherwise the loop would have continued for ever, or at least until the program is stopped by the impatient user.

Now with this knowledge we can write the first meaningful program, it tells the story of Genesis.

Example: stmt-while.p

```
i=1
sum = 0
while i <= 7
 sum = sum + i * i
 i = i + 1
end
println( ~msg 'sum of squares for integers
from 1 to 7 is ' .. sum )
```

Example: var-first.p

```
day = 1
while day < 8
```

```
  print( ~msg 'Day ' .. day .. ' : ' )
  if day == 1
    println( ~msg 'Create heaven and earth, create light' )
  elsif day == 2
    println( ~msg 'Create firmament, divide waters, call firmament Heaven' )
  elsif day == 3
    println( ~msg 'Gather water into seas, dry land, create grass, herbs and fruit trees' )
  elsif day == 4
    println( ~msg 'Create lights on firmament, create calendar, create sun and moon, day and night' )
  elsif day == 5
    println( ~msg 'Create water creatures, birds, big crocodiles, and all other animals, bless to
multiply' )
  elsif day == 6
    println( ~msg 'Create land animals, Create Man and Woman in His Image/kind, bless and command
to multiply, allow Humans to eat, say it is very good' )
  elsif day == 7
    println( ~msg 'Shabbes' )
  end
  day = day + 1
end
```

Functions also known as proceedcake

A program may need the same sequence of statements in several places at ones; one solution is to copy the same statements all over the program text, however this solution create much headache : one may have made an error and then the error is then repeated all over, because the program text that contains the error was copied many times over. Now instead of fixing the error once you have to fix thirteen of them; as many as there are copies of the original text. Bother!

The solution to the problem is called a function: wherever a function is used, all the statements of the function are evaluated; on the other hand the function has all them statements in one place.

Example: stmt-func-hello.p

```
println( ~msg 'hello world' )
```

pooh -x stmt-func-hello.p

```
001|println( ~msg 'hello world' )...
hello world
```

We have already encountered uses of functions throughout the text. See example stmt-func-hello.p uses the function **println** shows a message on the screen, followed by a newline delimiter; each input value has its own name, for the function **println** the name of the parameter is **~msg**

Example: stmt-func.p

```
sub pythagoras( a, b )
  c = a * a + b * b
  return sqr( ~num c )
end
println( ~msg 'the third side is ' .. pythagoras(
~a 5 ~b 12 ) )
```

A more complicated example is **stmt-func.p** here the function pythagoras receives two numbers a and b as input, each one stands for the length of the side of a right handed triangle; the function returns the length of the third side, by applying the Pythagorean theorem (the length of the third side is the square root of the sum

pooh -x stmt-func.p

```
005|println( ~msg 'the third side is ' ..
pythagoras( ~a 5 ~b 12 )...
002| c = ((a:5 * a:5):25 + (b:12 *
b:12):144):169
003| return sqr( ~num c:169 )...
003| return sqr( ~num c:169 ):1.300000e+01
```

of the squares of the other two sides)

A function is defined by the following form

sub <function_name> **(** <one or more input parameter names separated by , > **)**
  <the statements that are evaluated when function is called>
end

The input values of a function are also called 'parameters' of a function. The value of a parameters is set when the function is used / called.

When the function wants to pass a value back to the caller, it has to use the **return** statement; the return statement stops the function and returns to the place in the program where the function has been called; if **return** statement is followed by an expression, then the value of this expression becomes the value of the function call.

Another important detail: if the function defines a variable (like the variable c in line 5) then this variable is not visible from where the function has been called – the variable is local to the function from where it is defined. (In most other scripting languages all variables are visible from all places by default – they are global)

If you take a step back, then In a sense a function can be understood by thinking only about
1) the values that the procedure receives as input when it is used
2) the values that the procedure returns back to the program when it has finished.

This way of thinking has an advantage: it ignores the details of each function, so one can build larger programs, because the mind is freed to remember other things.

The disadvantage of this way of thinking is that it is easy to loose touch with the details; which is bad because one does no longer know what to do, as no longer knows how the function work and why it works and how.
I think that one has to balance between the two extremes, as with most thing in live.

```
005|println( ~msg 'the third side is ' ..
pythagoras( ~a 5 ~b 12 ):1.300000e+01 )...
the third side is 1.300000e+01
```

More tricks with functions

In which we talk about more tricky functions.

Functions and variable scope

If	

.h3 Parameters passed by reference
.h3 Optional parameters
.h3 Functions as values / Call me backson
.h3 Functions without a name

Green threads also known as pooh routines

When a regular function is called, the statements that make up the body of the function are evaluated, when the function is completed, the computer magically return to the spot from where the function was called and then continues to evaluate from right after the function call.

Threads are different: A thread is a unit that call its own functions, as usual. Each such thread performs a task of its own. Now lets imaging two threads. The consumer of honey thread – also known as the Bear thread, and the producer of honey thread the Bee thread.

Imagine the consumer thread, the Bear thread; the Bear is always busy with playing, humming and visiting Christopher robin or his many friends. From time to time the Bear thread is hungry. The bear thread calls the **resume** function on the bee thread, the Bear thread goes to sleep until it gets the data back from the Bee thread.

The bee thread is resumed; Imaging the producer thread of bees; the bees are always busy in a loop with sending scout bees to find new flowers, then send out the worker bees to gather the nectar and bring it back to the he beehive; the worker bees have to stack up the honey somewhere. From time to time some new honey is ready; the problem is that you can't just stop the bee thread and return to a calling function, as if it were a regular function, that would mess up the state of the bees. So the bee thread passes the honey data to the function **yield**, which suspends the bee thread, passes the data back to the calling Bear thread,

Now the Bear thread is once again active, the **resume** function returns with the data it received from the bee thread (the honey if you have noticed).

Note that only one process is active at a given time; the other threads are suspended.

Example: poohthread.p

```
sub myrange( from, to )
   i = from
   while i <= to
     threadyield0( ~yieldval i )
      i = i + 1
   end
end

thread := makethread ( ~func myrange )

thread( ~from 1 ~to 10)

[ running, value ] =
threadyieldvalue( ~thread th )

while( running != 0 )

  println( ~msg 'thread returned [ value ]' )

  [ running, value ] =
resumethread( ~thread th )
end

stopthread( ~thread th )
```

Also with this example we all know now why green threads are known as Pooh routines. The function makethread takes a function reference and returns a new special function reference; the returned function reference, when called will create a new thread. The new thread will receive its argument from the calling thread. When the new thread has a result that it wants to pass back, then it calls **threadyield** or **threadyield0**	

.h3 For loops
.h3 Higher order functions
.h3 Closures
.h3 Tables as objects
.h3 Recursion