Shank Language Definition, V3

Shank is different in some significant ways from languages that you may already know.

The biggest differences are:

1) Shank functions don't have return values.
2) Shank functions can alter variables that are passed into the function, when the variable is marked as "var"
3) Shank doesn't use curly braces {} for blocks – it uses indentation.
4) Shank for loops are much simpler than Java or C for loops.
5) Assignment uses :=, but comparison uses =

Overall Function Format

The overall format of a function:

function definition (define)
constants (optional)
variables (optional)
body

For example:

```
define start (args : array of string)
constants pi = 3.141
variables a, b, c : integer
      a := 1
      b := 2
      c := 3
```

Comments

Comments can span multiple lines. They start with { and end with }

Example:

```
{ This is a comment }
```

Please note: Comments currently start with (* and end with *) and the switch to curly brace comments is a planned change.

Blocks

Blocks are one or more statements that are run one after another. Blocks are indented one level more than the "owner" of the block. To end a block, you simply "un-indent" one level.

Indentation on empty lines or lines that contain only a comment is not counted.

Example:

```
for a from 1 to 10
      write a
      write a + 1
write "not in the block"
```

Built-in types:

integer (64-bit signed integer)
real (floating point)
boolean (constant values: true, false)
character (a single number/letter/symbol)
string (arbitrarily large string of characters)
array of (any of the above; no arrays of arrays)

Additional Data Types

Enumerated types can be created with the enum keyword:

```
enum color = [red, green, blue]
```

Records (similar to structures in C) can be created with the record keyword:

```
record student
      name : string
      gpa : real
```

Both data types are constructed at the file level (not per function). Records may contain arrays and other records. Record members can be accessed with "dot notation":

```
variables goodStudent : student { declaration }

goodStudent.gpa := 3.9 { usage }
```

Arrays

Arrays are declared much like other variables:

```
variables names : array from 0 to 5 of string
```

All arrays are 1 dimensional and must have a range declared at definition time. Referencing a variable is done using square brackets:

```
      names[0] := "mphipps"
      write names[0]
```

Variables

Variable declarations are defined by the keyword variables, a list of names, then a ":" and then the data type. A name must start with a letter (lower or upper case) and then can have any number of letters and/or numbers. There can be more than one variables line in a function. Variable declarations can also appear at the file level (i.e. outside a function and "global").

Example:

```
variables variable1, a, foo9 : integer
variables name, address, country : string
variables myColors : color { is an enum }
variables myFavoriteStudent : student { is a record }
```

Constants

Constants are variables that are set at definition and cannot be changed after definition. They do not require a data type since the data type is inferred from the value. There can be more than one per function. The data type of the function need not be consistent in the single line.

```
constants myName = "mphipps"
constants pi = 3.141
constants class = "ICSI311", goodGrade = 95
```

Type limits

Types can be limited at declaration time using “from” and “to”. Does not apply to booleans.

Example:

```
variables numberOfCards : integer from 0 to 52
variables waterTemperature : real from 0.0 to 100.0
variables shortString : string from 0 to 20 { string has a length limit }
```

Functions (also known as: Procedures/Methods/Subroutines)

A function is an (optional) constant section, an (optional) variable section and a block. Functions have a name and a set of parameters; this combination must be unique.

Function parameters are read-only (treated as constant) by default. To allow them to be changed, we proceed them by the keyword “var” both in the function declaration and in the call to the function.

Example:

```
define addTwo(x, y : integer; var sum : integer)
      sum := x + y
```

To call this function:

```
addTwo 5, 4, var total { total was declared somewhere else }
```

The var keyword must be used before each variable declaration that is alterable.

```
define someFunction(readOnly : integer; var changeable : integer;
alsoReadOnly : integer)
someFunction someVariable, var answer, 6
```

In contrast to other languages, **functions never return anything** except through the "var" variables. While this is unfamiliar to people who have used other languages, it is actually powerful, since you can return as many values as you choose.

```
define average(values : array of integer; var mean, median, mode : real)
```

When the program starts, the function "start" will be called.

Parameters to a function can have a default value:

```
define someFunction(readOnly : integer = 5)
```

If a parameter has a default value, all parameters after it must also have a default value.

```
define someFunction(readOnly : integer = 5, notAllowed : boolean)
define someFunction(readOnly : integer = 5, isOK : boolean = true)
```

Control structures and Loops

The only conditional control structure that we support is "if-elsif-else". Its format is:

if booleanExpression then block {elsif booleanExpression then block}[else block]

Examples:

```
if a < 5 then
      a := 5


if i mod 15 = 0 then
      write "FizzBuzz "
elsif i mod 3 = 0 then
      write "Fizz "
elsif i mod 5 = 0 then
      write "Buzz "
else
      write i, " "
```

There are three types of loops that we support:

```
for integerVariable from value to value
      block


while booleanexpression
      block


repeat until booleanExpression
      block
```

Note – the control variable in the for loop is **not** automatically declared – it must be declared before the for statement is encountered. From only counts forward by one.

Examples:

```
for i from 1 to 10
      write i { prints values 1 … 10 }

for j from 10 to 2 { this loop will never execute }
      write j

while j < 5
      j := j + 1

repeat until j = 0
      j := j - 1
```

Since these are statements, they can be embedded within each other:

```
if a < 5 then
      repeat until a = 6
            for j from 0 to 5
                  while k < 2
                        k := k + 1
            a := a + 1
```

Operators and comparison

Integers and reals have the following operators: +,-,*,/, mod. The order of operations is parenthesis, *,/,mod (left to right), then +,- (also left to right).

Booleans have: not, and, or. The order of operations is not, and, or.

Characters have no operators.

Strings have only + (concatenation) of characters or strings.

Arrays have only the index operator [], but all relevant operators apply to an indexed element.

Comparison can only take place between the same data types.

= (equals), <> (not equal), <, <=, >, >= (all done from left to right).

Importing/Exporting

All functions and data types are scoped to the file that they are in, by default. If a file wants functions or data types to “become visible” to other files, it must declare a module name:

```
module FileIO { must be the first line in the file }
```

It can then export:

```
export open, close, read, write { can occur anywhere }
```

Files can import from any module:

```
import FileIO { imports all of FileIO that is exported }
import FileIO [ open, close, read ] { doesn't import write or seek }
```

Variables can also be declared at the file scope; they are then accessible by any function in the file. They cannot be exported.

```
variables thisIsGlobal : integer { left indented, not in a function }
```

Unit Testing

To facilitate unit testing, unit tests are marked as such and included in the same file as the code:

```
define someFunction(x : integer; var y : integer)
      { some implementation }


test someFunctionWorks for someFunction(x : integer; var y : integer)
variables t : integer
      someFunction 5, var t
      assertIsEqual 7, t


test someFunctionWorksNegative for someFunction(x : integer; var y : integer)
variables t : integer
      someFunction -5, var t
      assertIsEqual -23, t
```

References

To support dynamic memory management, Shank supports references to records. These are very separate from the fixed memory management explained previously. Dynamically allocated items are always references. It is not possible to get a reference to a local or global variable, only dynamic memory.

References are created using the refersTo key word:

```
record myLinkedList
      data : integer
      next : refersTo myLinkedList

define start()
variables head, next : refersTo myLinkedList
      allocateMemory var head
      allocateMemory var next
      next.data := 5
```

```
	head.next := next
```

Note that accessing a reference before setting it is a compile time error. The only operations available on references are: . (access a record member), := (set the reference), IsSet myReference (returns true if the reference has a value).

Generics

Records and functions can both accept incomplete types – generics. The types must be completed at the point of use. With functions, this can be done by the parameter. Note, in the example below, allocateMemory requires a T; in the usage of allocateMemory, the data type passed in supplies the type. For records, we need to add the data type to satisfy the generic after the name of the data type.

Example:

```
record genericLinkedList generic T
	data : T
	next : refersTo genericLinkedList T

define allocateMemory (var output : refersTo T) generic T
	{ some implementation here }

define start()
variables head, next : refersTo genericLinkedList integer
	allocateMemory var head
	allocateMemory var next
	next.data := 5 { safety checked at compile time }
	head.next := next
```

Example:

```
record genericHashMapNode generic K, V
	key : K
	value : V
	next : refersTo genericHashMapNode K, V

define addToHashMap (newVal : refersTo genericHashMapNode K, V) generic K, V
	{ some implementation here }

define start()
variables node : refersTo genericHashMapNode string, string
	allocateMemory var node
	node.key := "hello"
	node.value := "world"
	addToHashMap node
```

Built-in functions

I/O Functions

Read var a, var b, var c { for example – these are variadic }
    Reads (space delimited) values from the user
Write a,b,c { for example – these are variadic }
    Writes the values of a,b and c separated by spaces

String Functions

Left someString, length, var resultString
    ResultString = first length characters of someString
Right someString, length, var resultString
    ResultString = last length characters of someString
Substring someString, index, length, var resultString
    ResultString = length characters from someString, starting at index

Number Functions

SquareRoot someFloat, var result
    Result = square root of someFloat
GetRandom var resultInteger
    resultInteger = some random integer
IntegerToReal someInteger, var someReal
    someReal = someInteger (so if someInteger = 5, someReal = 5.0)
RealToInteger someReal, var someInt
    someInt = truncate someReal (so if someReal = 5.5, someInt = 5)

Array Functions

Start var start
    start = the first index of this array
End var end
    end = the last index of this array

Memory Functions

AllocateMemory var newOne : T generic T
    Allocates memory for a variable of the type that newOne references. Sets newOne.

FreeMemory var oldOne : T generic T
    Frees the memory that oldOne points to. “Unsets” oldOne.

IsSet one : T, var isSet boolean generic T
    Sets isSet if one is pointing to memory.

Size someVariable: T , var size: integer generic T
    Sets size to the number of bytes that this variable takes up