VisiblyBasic

Code Documentation

Introduction

VisiblyBasic code is contained within .pseudo files. Each .pseudo file contains metadata at the beginning to indicate what subroutine is being tested, and what tests are being performed. VisiblyBasic code begins with one of the following keywords:

- GET
- SET
- RETURN
- FOR/NEXT
- WHILE/ENDWHILE
- IF/ELSEIF/ELSE/ENDIF

Each line is processed individually, as such any one line must either begin with one of these keywords, be blank, or contain a comment, indicated by a #.

VisiblyBasic code is executed through compiling .pseudo files into .asm files, which are then executed directly by the included interpreter. Within the code, the supported data types include arrays and unsigned integers. Arrays are 0 indexed. Any integer that passes below zero will remain equal to zero. Any number that is divided to form a decimal, is rounded down to the nearest integer.

Metadata

Metadata must be in following format:

```
#-- type: (sorted-search|unsorted-search|sorting)
#-- main: <subroutine name>
```

“type:” states the form of test data which should be fed to the program. Main determines where the program should start. Since metadata emulates comments it is ignored by the

compiler and simply fed through to the assembler files for later use by the interpreter module.

Get/Set/Return Statement

Each of these statements are in the following format (substitutions, optional elements, etc. are indicated with EBNF).

```
GET <variable name>["()"]
```

"()" indicates a literal, not part of EBNF

A GET statement retrieves input parameters. When "()" is included at the end of a GET statement, the parameter retrieved is known to be an array.

```
SET <variable name>["(",<expression>,")"] = <expression>
```

, in EBNF indicates a concatenation

A SET statement updates a variable or array value. New variables being created, and existing variables being modified both need the SET keyword to be updated.

```
RETURN <expression>
```

A RETURN statement exits the subroutine and outputs the expression value.

For Loop

```
FOR <variable name> = <expression start> TO <expression end>
   {<sub-process>}
NEXT <variable name>
```

{ } indicates repeating items

<variable name> is initially set to the value of <expression start>, and increments at the completion of each iteration of the loop, until the value of <expression end> is reached. <expression start> is only evaluated as the loop starts. <expression end> is reevaluated at the start of each loop. If <variable name> is modified within the loop, it won't create any unexpected behavior, and will continue to iterate and then be validated at the end of each loop.

While Loop

```
WHILE <conditional expression>
   {<sub-process>}
ENDWHILE
```

The loop is iterated through each time the <conditional expression> is met. If the <conditional expression> is not met initially, the loop is skipped.

If Statement

```
IF <conditional expression 0>
   {<sub-processes 1>}
[{ELSEIF <conditional expression 1>
   {<sub-processes 2>}}]
[ELSE
   {<sub-processes 3>}]
ENDIF
```

[ ] indicates optional elements

When the initial condition, <conditional expression 0> is met {<sub-processes 1>} is executed. If not each of the optional conditions <conditional expression 1> is checked. If this condition is met, {<sub-processes 2> is executed. Any number of ELSEIF statements can be included (if allowed for by ram limits). An optional ELSE statement can be included, where {<sub-processes 3>} is executed if none of the above conditions are met.

Expression

An expression is a set of mathematical symbols, which define a process to occur to integer literals or variable values. Expressions can contain the following tokens:

- Literal Variables - such as “i”, “length”, “value”, etc. - These must be defined variables, using SET or GET
- Array Expression - such as “array(5)”, “values(i + 2)”, etc. - The array must be a defined variable using GET. The variable being gotten from the array may be an expression itself.
- Numerical constant - such as “10” “1_000” “55” - any “_” in a numerical constant is ignored. Commas and decimal points may not be included.
- Operators - more details under next subheading
- Conditions - more details under next subheading

Operators & Conditional Expressions

Operators include all processes which are used to modify a numerical value. The following operators are listed by order of operations. Symbols are processed in ascending order. Symbols at the bottom are processed first. Some operators have equal precedence.

Symbol	**Title**	**EBNF**	**Precedence?**	**ASM translation**
||	OR	<expr:a> || <expr:b>		OR a b
&&	AND	<expr> && <expr>		AND a b
|	OR	<expr> | <expr>		OR a b
^	XOR	<expr> ^ <expr>		XOR a b
&	AND	<expr> & <expr>		AND a b
==	Equal?	<expr> == <expr>	EQUAL	XOR a b ATZERO a
!=	Not Equal?	<expr> != <expr>		XOR a b
>	Greater than?	<expr> “>” <expr>	EQUAL	CSUB a b
<	Less than?	<expr> “<” <expr>		CSUB b a EQUATE a b
<=	Less than or equal to?	<expr> “<=” <expr>		CSUB a b ATZERO a
>=	Greater than or equal to?	<expr> “>=” <expr>		CSUB b a ATZERO a EQUATE a b
<<	Left shift	<expr> << <expr>	EQUAL	SAL a b
>>	Right shift	<expr> >> <expr>		SAR a b
+	Add	<expr> + <expr>	EQUAL	ADD a b
-	Subtract	<expr> - <expr>		CSUB a b
*	Multiply	<expr> * <expr>	EQUAL	MULT a b
/	Divide	<expr> / <expr>		DIV a b
%	Modulo	<expr> % <expr>		MOD a b

“<”, “>” “<=”, “>=” represent literal <, >, <=, >= respectively

The remaining operators which do not consist of a pair of numerical values being modified in a certain fashion include the following.

Symbol(s)	EBNF	Operation/Asm translation	Priority?
!	!<expr>	ATZERO a	Lowest priority
( )	(<expr>)	Completes operation within brackets first	Peak priority
.	<variable>.<property>	When <property> = length: Gets length of array	N/A

Assembler

The majority of assembler operations are for processing operators such as addition and subtraction as listed under the Expression subheading. Additional instructions are included for variable manipulation, pointers, getting and setting.

The following instructions are included:

Instruction formatting	Symbolic Operation	Operation description
ADD a b	a = a + b	Addition
CSUB a b	a = MAX(a - b, 0)	Subtraction with a cap
MULT a b	a = a * b	Multiplication
DIV a b	a = FLOOR(a / b)	Division with rounding
MOD a b	a = a % b	Modulo
ATZERO a	a = (a == 0) ? 1 : 0	Inversion, not operator
OR a b	a = a | b	Bitwise OR
AND a b	a = a & b	Bitwise AND
XOR a b	a = a ^ b	Bitwise XOR
SAL a b	a = a << b	Left shift
SAR a b	a = a >> b	Right shift
SET a <constant>	a = <constant>	Set “a” to the value of a constant

EQUATE a b	a = b	Set the value of a to the value of another variable
PNTSET a b	a = mem[b]	Set “a” to the value in memory referred to by “b”
LOCSET a b	mem[a] = b	Set the value referred to by “a” to the value at “b”
JUMP a b	if b != 0: goto a	Jump to label a if b doesn’t equal 0

All variables (usually a b), are not constants, and are rather pointers to a value in memory. A constant/numeric literal is denoted by <constant>

A line of assembler may be blank, contain a comment (beginning with an apostrophe), or contain a label, which is used when jumping. Metadata is contained within comments. Labels are formatted like any other variable, with “:” being appended to the start.

A user of VisiblyBasic may execute arbitrary assembler code using the program. The user may also modify compiler generated assembler code. However, guaranteed successful execution of the program is less likely if the assembler code is written incorrectly.

PNTSET Example

Pointers:

Name	**Index**
a	0
b	2

Values:

Operation	**Index**		
	0	**1**	**2**
Initially	0	200	1
PNTSET a b	200	200	1

Explanation

Since “b” refers to index 2, where the value 1 is stored initially, the value at index 1 is retrieved. This value is 200. Since “a” refers to index 0. The value at index 0 is replaced with 200.

LOCSET Example

Pointers:

Name	**Index**
a	0
b	2

Values:

Operation	**Index**		
	0	**1**	**2**
Initially	1	0	200
LOCSET a b	1	200	200

Explanation

Since “a” refers to index 0, which contains the value 1, index 1 is going to be written to. Since “b” refers to index 2, which contains the 200, this is the value being written. Thus, the value at index 1 is replaced with 200.