



ivyscript reference

April 11, 2017

Allart Ian Vogelesang ian.vogelesang@hds.com

 **Hitachi Data Systems**

- Ivyscript is a programming language wrapper around the ivy engine control API.
 - See "programming the ivy engine"
- Ivyscript – for scripting test workflow
 - The only way to operate ivy, for now.
 - Similar to a subset of C/C++, with some minor differences.
 - Extensible - parser auto-generated from language grammar. (flex+bison)
 - Longer term
 - If we build either a CLI or a REST API on top of the ivy engine C++ API, the ivyscript programming language may cease being further enhanced.

- Statements in the programming language end with a semi-colon, like C / C++ / Java.
- C style comments are supported
 - The part from `/*` to `*/` is ignored
- C++ style comments are supported
 - From `//` to the end of the line is ignored.
- `#` style comments are supported
 - From `#` to the end of the line is ignored.

Make .ivyscript programs executable

- So-called "sha-bang" lines work.
- A sha-bang line is when you start a script with a line that specifies a path to the program used to interpret the script.
- For example, as the first line of an .ivyscript program:

```
#!/path_to_ivy_executables/ivy
```

(followed by remainder of ivyscript program)

- Then you can invoke the .ivyscript file as a program itself.

- Anywhere you can put a statement, you can put a nested block, which starts with "{" and ends with "}".
- Any variable or function declarations made inside a nested block are not "visible" to code outside the nested block.
- Nested blocks are typically used in `if` statements, looping constructs, etc.

- There are 3 types: `int`, `double`, and `string`.

- Examples of constants, also called literals:

<code>int:</code>	<code>0</code>	<code>-5</code>	<code>12345</code>			
<code>double:</code>	<code>5.</code>	<code>.5</code>	<code>5.5</code>	<code>5E-2</code>	<code>5%</code>	<code>5.5%</code>
<code>string:</code>	<code>"house"</code>		<code>" "</code>			

- There is also a hex form of `int` literal (constant)
 - `0x0` to `0x7FFFFFFF`
 - The hex form of `int` literal only supports non-negative values
- `5%` means the same thing as `0.05`.

- To include a double quote character in a string constant, escape it with a backslash:
 - `"the word \"house\" is double-quoted"`
- Other escaped characters: `\r`, `\n`, `\t`
- An escaped octal character value has 3 digits, e.g. `\001`
- An escaped hex character value has one or two digits, e.g. `\xf` or `\x0f`

- "identifiers" are eligible to serve as the name of a variable or function.
- An identifier begins with an alphabetic character (a letter) or a Japanese hiragana, katakana, or Kanji character, and continues with letters, Japanese hiragana, katakana, or Kanji characters, digits, and underscore _ characters.

- `<type> <list of identifiers <with optional = initializer expression>;`
- Examples:
- ```
int i, j;
int k = -1;
double c;
double d = 1.5;
string s;
string name = "bert";
```

- A constant (a literal of one of the types) is an expression
  - e.g. `"constant"`
- A variable reference is an expression
  - e.g. `x`
- Expressions may be combined together with operators, which operate the same as in C/C++:
  - `+, -, *, /, %, >, <, >=, <=, ==, !=, =, |, &, ^, &&, ||`
- Expressions have a type, `int`, `double`, or `string`

# Converting an expression to a different type

- `int( <expression> )`  
`double( <expression> )`  
`string( <expression> )`
- Some times called a "cast".
- The expression is evaluated and the result is converted to the target type.
- Can result in a run time error
  - E.g. evaluating `int( "cow" )` would cause a run-time error.

- + plus – for numbers, adds, for strings, concatenates
- minus
- \* multiply
- / divide
- % remainder from integer division

- `>` greater than  
`<` less than  
`>=` greater than or equal to  
`<=` less than or equal to  
`==` equal to  
`!=` not equal to
- There is no true/false type. Logical operators evaluate to an integer value just like the old C language before there was a `bool` type.
  - An `int` or a `double` value used as a logical expression means "false" if the numeric value is zero, and means "true" for any non-zero value.
  - Use of `int` logical values is transparent – it works the way you expect it to.

- The bitwise operators operate on the individual bits in an `int` value, exactly like in C/C++.
- |      bitwise or  
  &      bitwise and  
  ^      bitwise exclusive or

- These operate on logical expressions, which evaluate to an `int` interpreted to mean "false" if the `int` value is zero, "true" otherwise.
  - Like in C/C++ the second expression after the operator is not evaluated if the result is known from evaluating the first expression before the operator.
- `||`      logical or
  - Evaluates the first expression, and if true, returns true. Otherwise, it evaluates the second expression and returns its true/false value.
- `&&`      logical and
  - Evaluates the first expression, and if false, returns false. Otherwise it evaluates the second expression and returns its true/false value.
- `!`      not
  - Evaluates a logical expression and returns the opposite.

- `<identifier> = <expression>`
- The identifier is looked up in the symbol table at compile time, and if it's valid, at execution the expression is evaluated and the variable is set to that value.
- If the expression is not of the same type as the variable, the value may be coerced / converted, or in some cases a compile time error occurs.



- `<identifier> ( <comma separated list of zero or more expressions> )`
  - E.g. `sin(.5)`
- Identifier and parameter list signature are looked up at compile time to and if valid, a function call is built.
- At run time, the expressions are evaluated and the resulting parameter values are passed to the function, the function is executed, and the result is returned.

- Same as C/C++

- `if ( 3*4+5*6 == fred || ! Person == Nancy = 4)`

- Means

- `if ( ( ( (3*4) + (5*6) ) == fred) || ( ! (Person==(Nancy=4) ) ) )`

- If you are not sure, group with parentheses ().

- E.g.

```
- int add_three(int i)
 {
 return i+3;
 };
```

Semicolon needed for function definitions, unlike in C / C++

- Functions have a type, which is the type of the object they return to the caller.
- Functions can be "declared" without being defined yet:  

```
int add_three(int i);
```

- As in most programming languages you can "include" or "import" a copy of some ivyscript code from a library.
- In ivyscript, you say:

```
include port_scalability_test.ivyscript
```

- Just be advised that the initial implementation of this feature allows infinite loops.
  - Need to consider use cases – in some cases it could be valid to be importing multiple copies, but from different and separate places.
  - Maybe default to #pragma once behaviour, requiring specific override.

- It's OK to have different functions with the same name as long as the sequence of types of the parameters is different so the compiler can tell them apart.
- ```
int      addtwo(int i)      { return i+2; }  
string  addtwo(string s) { return s + "two"; }
```

Ivy-specific builtin functions

- `string outputFolderRoot();` from [OutputFolderRoot] **statement** – default "."
- `string testName();` root part of ivyscript file without .ivyscript suffix
- `string masterlogfile();` you can `log(masterlogfile(), "message\n");`
- `string testFolder();` root folder for output from this run
- `string stepNNNN();` from most recent [Go!], e.g. step0002
- `string stepName();` from most recent [Go]
- `string stepFolder();` subfolder for most recent [go] within testFolder()
- `string last_result();` for most recent [Go], returns "success" or "failure"
- `string show_rollup_structure();` shows type / instance / workload thread hierarchy.

Math builtin functions – same as C/C++

- `double sin(double), double cos(double), double tan(double)`
- `double sinh(double), double cosh(double), double tanh(double)`
- `double asin(double), double acos(double),
double atan(double), double atan2(double, double)`
- `double log(double), log10(double),
double exp(double), double pow(double, double)`
- `double sqrt(double)`
- `int abs(int)` - **absolute value**
- `double pi(), double e()`

- `string substring(string s, int begin_index_from_zero, int number_of_chars);`
- `string left(string s, int n);` like in BASIC, gives you leftmost / rightmost characters
`string right(string s, int n);`
- `string trim(string s);` removes leading / trailing whitespace
- `string to_lower(string s);`
`string to_upper(string s);`
- `int stringCaseInsensitiveEquality(string s1, string s2);`
- `string int_to_ldev(int n);` `int_to_ldev(0xFF)` **returns** "00:FF"
- `string to_string_with_decimal_places(double x, int n);`
`to_string_decimal_places(3.1415,2)` **returns** "3.14"

- ivy uses the default flavour of C++ `std::regex`, which I think uses the ECMAScript dialect
- `int regex_match(std::string s, string regex);`
E.g. `if (regex_match("horse","(horse)|(cow)")) then print("animal\n");`
- `int regex_sub_match_count(string s, string regex);`
- `string regex_sub_match(string s, string regex, int n);`
n must be less than `regex_sub_match_count(s, regex)`
- `int matches_digits(string s);`
`int matches_float_number(string s);`
`int matches_float_number_optional_trailing_percent(string s);`
some ivy parameters can be set to these
`int matches_identifier(string s);`
alphabetic, continued with alphanumeric and underscores
`int matches_IPv4_dotted_quad(string s);`

Accessing csv files – row and column

	A	B	C	Q	R	S	AW	AX			
Header row is row -1											
	Test Name	Step Number	Step Name	iogenerator type	blocksize	maxTags	Overall IOPS	Overall Decimal MB/s	Overall Average Blocksize (KiB)	Overall Little's Law Avg Q	Overall Average Service Time (ms)
1											
2	demo9	step0000	iops_max	random_independent	4 KiB	32	2597.16	10.638	4	64.0024	24.6432
3	demo9	step0001	baseline_service_time	random_independent	4 KiB	32	25.81	0.105718	4	0.164588	6.37691
4	demo9	step0002	1.125_x_baseline	random_independent	4 KiB	32	555.849	2.27676	4	4.08014	7.34037
5	demo9	step0003	1.25_x_baseline	random_independent	4 KiB	32	1097.86	4.49682	4	8.74646	7.96686
6	demo9	step0004	1.5_x_baseline	random_independent	4 KiB	32	1486.01	6.08671	4	14.2187	9.56836
7	demo9	step0005	1.75_x_baseline	random_independent	4 KiB	32	1722.35	7.05476	4	19.2298	11.1649
8	demo9	step0006	2_x_baseline	random_independent	4 KiB	32	1898.07	7.77448	4	24.2199	12.7603
9	demo9	step0007	3_x_baseline	random_independent	4 KiB	32	2353.68	9.64067	4	45.0346	19.1337
10	demo9	step0008	4_x_baseline	random_independent	4 KiB	32	2602.77	10.6609	4	63.9994	24.589
11	demo9	step0009	5_x_baseline	random_independent	4 KiB	32	2601.7	10.6566	4	63.9962	24.5978

Test step csv files (not shown) have one line per subinterval (both host & subsystem data)

Summary csv files like this one have one line per test step.

Use column number from 0, or say "Overall IOPS"

Row 0 is test step 0 or subinterval 0

Csv file builtin functions 1/3 – overall size

- These csv functions are the same as the standalone ivy companion csv file command line utilities
- `int csv_rows(string filename);`
 - Number of rows following the header row.
 - Returns -1 if invalid file or file empty. Returns 0 if there was only a header row.
- `int csv_columns_in_row(string filename, int row);`
- `int csv_header_columns(string filename);`
 - Same as `csv_columns_in_row(filename, -1)`

Csv file builtin functions 2/3 – individual cells

- `string csv_cell_value(string filename, int row, int column);`
`string csv_cell_value(string filename, int row, string column_header_text);`
 - You can refer to a column using an int, the column index from zero.
 - You can refer to a column using a string, the column header text.

- `string csv_raw_cell_value(string filename, int row, int column);`
`string csv_raw_cell_value(string filename, int row, string column_header_text);`
 - ivy "wraps" text fields as a formula with a string constant, e.g. `"horse"`
 - This stops Excel from interpreting 1-1 as January 1st, and 00:00 from interpreting as a time.
 - The csv file functions normally "unwrap" csv column values, removing this kind of wrapper or removing simple double quotes surrounding a value, to treat `"horse"`, `"horse"` and `horse` the same
 - Retrieving the raw value give you exactly what was between the commas in the csv file.

Csv file builtin functions 3/3 – headers & slices

- `int csv_lookup_column(string filename, string column_header);`
 - Gives you the column number for a column title string.
- `string csv_column_header(string filename, int col);`
 - Give you the text of the column header for a column number from zero.
- `string csv_column(string filename, int col);`
`string csv_column(string filename, string column_header);`
 - Gives you a "column slice" of the spreadsheet showing "raw" values.
 - E.g. "IOPS,55,66,55,44"
 - Demo number 8 shows iterating through the column slices to write out the transpose of a csv file.
- `string csvfile_row(int row);`
 - Gives you a "row slice" of the spreadsheet showing the "raw" values.
 - E.g. "random_independent",="4 KiB",32,2601.7

- `string print(string), double print(double), int print(int)`
 - Prints the specified value to stdout and then returns that value.
- `int fileappend(string filename, string s)`
 - One way to write output. Does not append a newline to `s`.
- `int log(string filename, string s)`
 - Writes a timestamp prefix before the string, and adds terminating newline if the last line in `s` doesn't already have one.
 - E.g. `log(masterlogfile(), "message");`
- `trace_evaluate(int)`
 - Turns execution tracing on/off. Zero means off, otherwise on.

- `string shell_command(string)`
 - Executes the shell command and returns its output.
 - **Runs as `root`. You have been warned.**
 - Ivy runs as `root` in our lab because ivy uses ssh to fire up ivyslave and `ivy_cmddev` on test hosts, and "`root`" has been set up to not require a password to ssh. Ivy may also need to run as `root` to do I/O to raw LUNs – not sure.
 - The only ivy component that definitely requires to run as `root` is the SCSI Inquiry tool, which has the executable that issues "SCSI Inquiry" marked `setuid` as `root`, and thus works for any user.
 - Use `shell_command()` to do almost anything
 - `grep` in an ivy output folder to find a csv file name
 - Get a time or date stamp

- As in

```
– if ( last_result() != "success" )  
  {  
    print "timed out without making a valid measurement.\n";  
    exit();  
  }
```


Statements: expression statement

- `<expression> ;`
- Executes the expression and discards the result.

- `if (<logical expression>) <statement>`
- `if (<logical expression>) <statement> else <statement>`
- `<statement>` can be a single statement, or it can be a nested block starting with `{` and ending with `}`.

- `int x = 1;`

```
if ( x >= 0 ) print( "x is greater than or equal to zero.\n");  
else          print( "x is less than zero.\n");
```

```
if ( x >= 0 ) { print( "x is greater than or equal to zero.\n"); x = x + 1; }  
else          { print( "x is less than zero.\n");                x = x - 1; }
```

Statements – traditional C style for loop

- `for (<initializer expression> ; <logical expression>; <epilogue expression>)`
 <loop body statement>
- The initializer expression is run.
- Then the logical expression is evaluated, if false, execution of the statement is complete.
- Otherwise, the loop body statement is run, then the epilogue expression is run, then we loop back to where we will evaluate the logical expression.

Example of traditional for loop

- ```
int i;
for (i=0; i<10; i=i+1)
{
 print("i = " + string(i) + "\n");
}
```
- Note that it's not `for (int i=0; i<10; i++)`
  1. The initializer is an expression, not a statement, so can't declare `i` to be an `int`.
  2. There is no increment operator `++` as in C++.

# Statement – list-style for loop

- For <identifier> = { <list of expressions> } statement
- E.g.

```
int i;
for i = { 0, 1, 2 }
 print("i = " + string(i) + "\n");
```

```
string s;
for s = { "cat", "dog", "mouse" }
{
 print ("A " + s + " has four legs.\n");
}
```

# Statement – while loop

- `while ( <logical expression> ) <loop body statement>`
- The logical expression is evaluated, and if false, execution of the statement is complete.
- Otherwise, the loop body statement is executed and then we loop back to evaluating the logical expression again.

# Statement – do - while loop

- `do <loop body statement> while ( <logical expression> ) ;`
- The loop body statement is executed, and then the logical expression is evaluated, and if the result was "false", execution of the statement is complete.
- Otherwise, and then we loop back to running the loop body statement again.



**HITACHI**  
Inspire the Next

# Thank You

 **Hitachi Data Systems**