Marvel Design Documentation

Expressions

Data Types

All expressions fall under exactly 1 of these data types:

Marvel Data Type	Java Equivalent
`int`	`int`
`real`	`double`
`char`	`char`
`string`	`String`
`bool`	`boolean`

Marvel is strongly typed, and it will catch type-unsafe operations at compile-time.

Operators

Arithmetic Operators

Marvel supports the following arithmetic operator symbols: `+ - * /`

`-` is also a unary operator, which flips the sign of the operand. Ex: `-x` is just `-1*x`

`+` is also a unary operator, but essentially does nothing. Ex: `+x` is just `x`

`+` can also be used to concatenate strings.

`/` is always true division, meaning it returns a `real` when dividing 2 `int`s.

`int`s and `real`s are compatible for all arithmetic operators.

Comparison Operators

Marvel supports the following comparison operators: `is isnot > < <= >=`

`is` and `isnot` are equality and inequality operators respectively.

`int`s and `real`s are compatible for all comparison operators.

`> < <= >=` are only applicable to numeric types.

Assignment Operators

Marvel supports the assignment operator: `=`

The expression on the right side must match the data type of the left.

The exception to this is `int`s and `real`s. Assigning an `int` to a `real` automatically adds a zero decimal part. Assigning a `real` to an `int` truncates the decimal part.

Logical Operators

Marvel supports the following keywords as logical operators: `and or not`

Logical operators can be used on any type; the result is a boolean. The truth value of data types is detailed later in the if statement section.

Operator Precedence

This is the order in which Marvel evaluates operations:

1. `not` `+` (unary) `-` (unary)
2. `*` `/`
3. `+` `-`
4. `is isnot < > <= >=`
5. `or and`
6. `=`

Associativity

Left-to-right associativity

Statements

Variable Declaration

```
<data type> <variable name> (= <expression>)? ;
int myInt;
real myReal = -2.4;
```

If there is no expression initially assigned, the variable will be assigned a default value, depending on its type.

Data Type	Default Value
`int`	`0`
`real`	`0.0`
`char`	`'\u1000'` (empty char)
`string`	`""` (empty string)
`bool`	`false`

Note that in Marvel, all variable declarations are global, meaning a variable declared in an inner scope will also be accessible outside that scope. Also, you have to declare the variable first before using it. Lastly, You cannot declare two variables with the same name. Violating any of these rules results in a compile error.

Assignment Statement

```
<variable name> = <expression> ;
myInt = 5;
```

Assigns the value of expression to the variable with name `<variable name>`. The types of both types must be the same. The exception is `int`s and `real`s, which can be inter-converted. If the variable is a boolean, then expression's truth value is assigned to it. (Covered later in the if statement section)

Puts Statement

```
puts <expression> (, <expression>)* ;
puts “Hello world!”;
puts “myReal =”, myReal, “myInt=”, myInt;
```

Prints the expressions to the console, separated by spaces and terminated by a new line.

Print Statement

```
print <expression> (, <expression>)* ;
```

```
print “Hello world!”, “Hi universe!”;
```

```
print myReal, myInt;
```

Exactly like `puts`, except not terminated with a new line.

Alert Statement

```
alert <expression> (, <expression>)* ;
```

```
alert “Hello world!”, “Hi universe!”;
```

```
alert myReal, myInt;
```

Exactly like `print`, except the output is not the console but in a dialog box. A single alert statements opens only one dialog box. (Not one for each expression)

Input Statement

```
input <variable name>;
```

```
input myInt;
```

Assigns the result of the dialog box input to the variable. Automatically casts the input string to the variable's type. Also prints out both the prompt and the input to the console.

The prompt is predefined, and it says: “`Enter <variable type> <variable name>: `“. Example: “`Enter real myReal: `“. This is done to encourage naming variables appropriately instead of arbitrary letters like x or z. The type is included to give a hint to the user what to input.

In case the user enters an invalid value, he is prompted again and again until he enters a valid value. Hence, the input statement always ensures that the variable is left in a consistent state without any further action on the programmer's part.

If Statement

```
if <expression> then <block>
(elsif <expression> then <block>)* (else <block>)? end;
```

```
if num is 0 then
      alert “zero”;
elsif num < 0 then
      alert “negative”;
else
      alert “positive”;
end;
```

The expression is not limited to being a boolean type. If used in an if expression, the following will evaluate to false depending on the data type:

Data Type	False Value
`int`	`0`
`real`	`0.0`
`char`	`''` (empty char)
`string`	`isEmpty()` is true
`bool`	`false`

Any other values are evaluated as true.

Unless Statement

```
unless <expression> do <block> end;
unless denom is 0 do
      quotient = num / denom;
end;
```

Sometimes expressing the negative is more readable than the positive. The unless statement executes the block only if expression is evaluated as false. However, multiple unless statements in sequence (else unless) is not readable so it is limited to a single block. Should you want to put an else block, consider using the if statement instead.

While Statement

```
while <expression> do <block> end;
while i < 10 do
      i = i + 1;
end;
```

Executes the block over and over while expression is evaluated as true.

Until Statement

```
until <expression> do <block> end;
until j is 0 do
      j = j - 1;
end;
```

Executes the block over and over while expression is evaluated as false.

Do..Repeat Statement

```
do <block> repeat (while | until) <expression> end;
do
      input num;
repeat until num end;
```

First executes the block, then repeats execution while/until the expression is true.

The example above keeps asking the user to input `num` until he enters a non-false value (again, false values are `0` for numeric, `empty` for char and string, `false` for boolean).

Functions

Function Declaration

```
to <function name> do <block> end;
to printHelloWorld do
     puts “Hello world!”;
end;
```

Defines a void function. In Marvel, functions do not have arguments, but since the function scope is shared with the entire program's scope, a function can access and modify variables outside its scope. Variables and functions can share names.

Function Call

```
call <function name> ;
```

Executes the block defined in the function with name `<function name>`. Note that like variables, functions must be declared before they are first called.

Comments

Single Line

```
# comment
```

Multi-line

```
/* Multi line
 * comment      */
```

References

Dos Reis, Anthony. Compiler Construction Using Java, JavaCC, and Yacc. Wiley, 2011.

http://www.codeproject.com/Articles/50377/Create-Your-Own-Programming-Language

http://cui.unige.ch/isi/bnf/JAVA/method_declaration.html

https://javacc.java.net/doc/javaccgrm.html

cs.lmu.edu/~ray/notes/javacc/

cse.spsu.edu/clo/teaching/cs4713/lecture/node16.html

tomcopeland.blogs.com/juniordeveloper/2007/04/java_to_python_.html

http://docs.python.org/release/2.5.2/ref/grammar.txt

www.cs.sunysb.edu/~cse304/Fall07/projects/ast/ast.html

www.cs.nmsu.edu/~rth/cs/cs471/InterpretersJavaCC.html

http://www.beanshell.org/