Bob Manual

This is a preliminary specification. VM Labs reserves the right to make changes to any and all of the interfaces described in this document.

Programming in bob

Bob is a simple object-oriented scripting language. Bob's syntax is similar to C or C++ but it is dynamically typed and has automatic memory management.

A Simple Example

Functions

To get started I'll present a simple program written in Bob. This program is a method for computing the factorial of a number.

```
define factorial(n)
{
    if (n == 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```

As you can see from this example, Bob code looks a lot like C code without the type declarations. You don't have to declare the type of a variable in Bob. Any variable can assume a value of any type. Values have types not variables.

In fact, Bob isn't exactly like C. In Bob all statements and expressions return a value. This means it is unnecessary in many cases to use the **return** statement. The above method could just as well be written like this:

```
define factorial(n)
{
    if (n == 1)
        n;
    else
        n * factorial(n - 1);
}
```

Objects

Now let's look at how objects work. You can create an object by applying the **new** operator to an existing object. The value of the variable **Object** is the root of the object hierarchy. We'll create an object based on it:

```
employee = new Object;
```

This expression creates a new object and assigns it to the variable **employee**. An object is a collection of named properties. We can add a property to an object simply by assigning a value to it. For instance:

```
employee.name = "Fred";
employee.monthlySalary = 1000;
```

Our object now has the properties **name** and **salary**. We can compute the Fred's yearly salary with the expression:

employee.monthlySalary * 12;

Prototype Objects

Sometimes it is helpful to group similar objects into a class of objects so that properties that are shared among the objects are defined only once for the whole class. Bob doesn't support the class vs. instance distinction that languages like C++ and Java do but it is possible for an object to inherit from another object. This object is called the *parent* or *prototype* object. For instance:

```
Employee = new Object;
Employee.healthProvider = "Blue Cross";

fred = new Employee;
fred.name = "Fred";
fred.monthlySalary = 1000;

barney = new Employee;
barney.name = "Barney";
barney.monthlySalary = 1200;
```

We now have three objects, **Employee** contains properties common to all employees, **fred** and **barney** are specific employees. Properties added to either **fred** or **barney** are local to those objects. Properties added to **Employee** will be visible from any object created from **Employee**.

Methods

In addition to having properties, objects can have methods. If we want to be able to compute the yearly compensation for any employee, we can define the method **YearlyCompensation**.

```
define Employee.YearlyCompensation()
{
    return this.monthlySalary * 12;
}
```

Within a method the variable **this** refers to the object for which method is being invoked. Now we can compute Fred's yearly compensation with the expression:

```
salary = fred.YearlyCompensation();
```

Initialization Method

It can also be tedious to have to initialize the properties of an object each time a new one created. To make this easier, Bob automatically invokes the **initialize** method of a new object immediately after it is created and passes the **initialize** method any parameters that were given in the **new** expression that created the object. For instance, we could set the name and initial salary of a new employee by defining the following **initialize** method:

```
define Employee.initialize(name,monthlySalary)
{
    this.name = name;
    this.monthlySalary = monthlySalary;
    return this;
}
```

We can now create Fred and Barney with the following expressions:

```
fred = new Employee("Fred",1000);
barney = new Employee("Barney",1200);
```

It is important for an **initialize** method to return the value of **this** since it is this value that is returned by the **new** expression.

Now let's define a method for showing the state of an employee object:

```
define Employee.ShowState()
{
    local compensation = this.YearlyCompensation();
    stdout.Display("Name: ",this.name,"\n");
    stdout.Display("Compensation: ",compensation,"\n");
}
```

This example shows how to declare local variables in Bob. The **local** keyword introduces local variables. Each local variable can optionally have an initial value expression. The **stdout** symbol is a global variable initialized to a **File** associated with the stdout stream. Similarly, **stdin** and **stderr** are global variables associated with **File**s of the corresponding stdio streams.

The **File** method **Display** takes an arbitrary number of arguments and displays them to the stream associated with the **File**. Another method **Print** is available that behaves the same except that it prints strings within quotes.

Calling Prototype Methods

Sometimes an object needs its own method that expands upon the behavior of a method defined in its parent or prototype object. This can be accomplished in Bob using the keyword **super**:

```
Executive = new Employee;

define Executive.initialize(name,monthlySalary,bonus)
{
    super.initialize(name,monthlySalary);
    this.bonus = bonus;
    return this;
}

define Executive.YearlyCompensation()
{
    return super.YearlyCompensation() + this.bonus;
}

richard = new Executive("Richard",2000,1000);
```

Function and Method Definitions

function-definition:

define *function-name* **(** *formal-arguments* **)**
block

method-definition:

define *object* . *method-name* **(** *formal-arguments* **)**
block

formal-arguments:

[*required-arguments*] [*optional-arguments*] [*rest-argument*] *required-arguments*:

identifier [**,** *identifier*]...

optional-arguments:

optional-argument [**,** *optional-argument*]...

optional-argument:

identifier = *default-value*

rest-argument:

identifier **..**

Blocks

block:

{ [*local-declaration*]... *statement*... **}**

local-declaration:

local *local-variable* [**,** *local-variable*]... ;

local-variable:

identifier [= *initial-value*]

Statements

statement:

block
if-statement
while-statement
do-while-statement
for-statement
break-statement
continue-statement
switch-statement
case-statement
default-statement

return-statement
expression-statement
empty-statement

if-statement:

if (*test-expression* **)** *then-statement*
if (*test-expression* **)** *then-statement* **else** *else-statement*

while-statement:

while (*test-expression* **)** *statement*

do-while-statement:

do *statement* **while (** *test-expression* **) ;**

for-statement:

for ([*init-expression*] **;** [*test-expression*] **;** [*update-expression*] **)** *statement*

break-statement:

break ;
break *expression* **;**

continue-statement:

continue ;

switch-statement:

switch (*expression* **)** *statement*

case-statement:

case *literal-value* **:**

default-statement:

default :

return-statement:

return ;
return *expression* **;**

expression-statement:

expression **;**

empty-statement:

;

Expressions

expression:

sequence-expression
logical-expression
ternary-expression
assignment-expression
bitwise-logical-expression
comparison-expression
arithmetic-expression
increment-expression
object-creation-expression
method-call-expression
l-value
literal
(*expression* **)**

sequence-expression:

expression **,** *expression*

logical-expression:

expression **||** *expression*
expression **&&** *expression*
! *expression*

ternary-expression:

test-expression **?** *true-expression* **:** *false-expression*

assignment-expression:

l-value **=** *expression*
l-value **+=** *expression*
l-value **-=** *expression*

l-value ***=** *expression*
l-value **/=** *expression*
l-value **%=** *expression*
l-value **&=** *expression*
l-value **|=** *expression*
l-value **^=** *expression*
l-value **<<=** *expression*
l-value **>>=** *expression*

bitwise-logical-expressios:

expression **|** *expression*
expression **^** *expression*
expression **&** *expression*
~ *expression*

comparison-expression:

expression **==** *expression*
expression **!=** *expression*
expression **<** *expression*
expression **<=** *expression*
expression **>=** *expression*
expression **>** *expression*
expression **<<** *expression*
expression **>>** *expression*

arithmetic-expression:

expression **+** *expression*
expression **-** *expression*
expression ***** *expression*
expression **/** *expression*
expression **%** *expression*
- *expression*

increment-expression:

++ *l-value*
-- *l-value*
l-value **++**
l-value **--**

object-creation-expression:

new *expression* **(** [*argument* [, *argument*]…] **)**

method-call-expression:

expression **(** [[*argument* [, *argument*]…]] **)**

expression . *property-expression* **(** [[*argument* [, *argument*]…]] **)**

super . *property-expression* **(** [[*argument* [, *argument*]…]] **)**

l-value:

expression **.** *property-expression*

expression **[** *expression* **]**

identifier

property-expression:

identifier
(*expression* **)**

literal:

literal-symbol
literal-function
literal-vector
literal-object
literal-integer
literal-float
literal-string
true
false
nil

literal-symbol:

\ *identifier*

literal-function:

function [*name-string*] **(** *formal-arguments* **)** *block*

literal-vector:

\ **[** [*expression* [**,** *expression*]…] **]**

literal-object:

\ **{** [*object-expression*] [*property-value* [**,** *property-value*]…] **}**

property-value:

identifier **:** *expression*

literal-string:

characters enclosed in double quotes

\b	backspace
\f	form feed
\n	newline
\r	return
\t	tab
\xNN	hex character code
\"	quote
\NNN	octal character code

literal-integer:

digits optionally preceeded by *0* to indicate octal or *0x* to indicate hexadecimal

literal-float:

digits with either an embedded decimal point or *e* followed by an exponent

Globals

stdin
stdout
stderr
pi

Within a Method

this
_next

Library

```
new Object
new Object( [arg [, arg]…])
Object.Class()
Object.Clone()
Object.Exists(symbol)
Object.ExistsLocally(symbol)
Object.Send(selector [,arg]…)
Object.Show()

Method.Decode([file])
Method.Apply([arg [,arg]…])

new Vector
new Vector(size)
Vector.Clone()
Vector.size
Vector.size = newSize
Vector.Push(value)
Vector.PushFront(value)
Vector.Pop()
Vector.PopFront()

Symbol.printName

new String
new String(size)
String.size
String.Intern()

Integer.toInteger()
Integer.toFloat()
Integer.toString(fmt="%d")

Float.toInteger()
Float.toFloat()

new File(name,mode)
File.Print( [arg [, arg]…])
File.Display( [arg [,arg]…])
File.GetC()
File.PutC(ch)
File.Close()

Type(value)

toString(value)
```

Load(filename)
LoadObjectFile(filename)

Quit()
gc()

abs(n)
sin(n)
cos(n)
tan(n)
asin(n)
acos(n)
atan(n[,m])
sqrt(n)
ceil(n)
floor(n)
exp(n)
log(n)
log2(n)
log10(n)
pow(x,y)

rand(n)