back

Vapor Language Specification

A Vapor program is a list of functions and data segments. For example:

```
func DoSomething(a b)
  PrintInt(a)
  addr = call :LoadTableData(b)
  val = [addr+4]
  ret val

const Table
  2
  3
  5
  7

func LoadTableData(offset)
  addr = Add(:Table offset)
  ret addr
```

All values — integers and addresses — are four bytes long.

Identifiers

Identifiers are used for two things: variables and labels. Variables are always local to a function; variable names must be unique within a function.

There are three types of labels: data labels, code labels, and function labels. All label names must be unique across the entire program.

Identifiers consist of a sequence of letters, digits, dots ("."), and underscores ("_"), but the first character cannot be a digit or a dot.

Data Segments

Vapor has two types of global data segments. A "const" segment is for read-only data (like virtual function tables). A "var" segment is for global mutable data.

Each section starts with a data labels and is followed by static data values. For example:

```
const MinutesPerHour
  60

var MyClass.FunctionTable
  :MyClass.Start
  :MyClass.Finish
  -1
```

Each entry in a data segment is four bytes long. The entire first segment is four bytes long and contains the 2's complement representation of the number 60. It's a constant data segment and so memory write operations will fail at runtime.

The second segment is twelve byte long and consists of the address of the "MyClass.Start" function, followed by the address of the "MyClass.Finish" function, followed by the 2's complement representation of the number -1. This is a variable data segment and can be written to at runtime.

The two data labels, "MinutesPerHour" and "MyClass.FunctionTable", can be used in other places in the program.

Functions

The syntax for a function definition is:

```
func FunctionLabel(Params...)
  Body...
```

Each line of the body of a function is one of:

- code label: *Label*:
- assignment: *Location* = *Value*
- branch: `if` *Value* `goto` *CodeAddress*
- goto: `goto` *CodeAddress*
- function call: `call` *FunctionAddress* (*Args...*)
- function return: `ret` *Value*
- call to built-in operation: *OpName* (*Args...*)

Assignment

There are actually three distinct types of assignment. The first is variable assignment:

```
Var = Value
```

Here, "*Value*" is either an integer literal, a string literal, a variable name, or a label reference ("*:Label*").

For example:

```
a = 12              Store the value 12 into variable 'a'.
a = sum             Copy the value in variable 'sum' into variable 'a'.
a = :Factorial      Store the address of the label Factorial into variable 'a'.
```

The next two types of assignment are memory load and memory store. Memory operations always operate on 4-byte quantities and memory addresses must be 4-byte aligned.

```
Var = MemoryReference
MemoryReference = Value
```

A memory reference consists of a base address, which is either a label reference or a register, followed by an integer offset (either positive or negative).

- `[:MyArray+4]` refers to the address 4 bytes past the "MyArray" label.
- `[x-4]` refers to the address 4 bytes before the address stored in variable 'x'.

Some memory load/store examples:

```
x = [:FunctionTable+8]
[:GlobalCounter] = 15
[array-4] = length
```

Branch

There are two variants of the branch instruction:

```
if Value goto :CodeLabel
if0 Value goto :CodeLabel
```

The "`if`" jumps to *CodeLabel* if *Value* is non-zero and falls through to the next instruction otherwise. The "`if0`" does the opposite, jumping to the specified label if *Value* is zero.

Goto

The "`goto`" instruction is an unconditional jump to the specified target.

```
goto :CodeLabel
```

In addition to jumping to fixed labels, the "`goto`" instruction can also jump to a computed address read in from a variable.

```
goto Var
```

Function Call

```
Var = call :FunctionLabel (Args...)
```

The "*Var* =" is optional.

The "*Args...*" list is a whitespace-separated list of "*Value*" entries (either integer literals, variables, or label references). The return value of the function is stored in the *Var* variable.

Like "`goto`", "`call`" can also use a function address loaded from a variable:

```
Var = call Var (Args...)
```

Function Return

The "`ret`" instruction returns from a function. The return value is optional.

```
ret Value
ret
```

Built-In Operations

In addition to the core language, the Vapor interpreter also supports a set of built-in operations for things like arithmetic, memory allocation, and displaying output.

- Basic arithmetic: `Add`, `Sub`, `Mul`, `Div`, `Rem`, `MulS`, `DivS`, `RemS`, `ShiftL`, `ShiftR`, `ShiftLA`. The "`-S`" variants operate on signed integers.
- Comparison: `Eq`, `Ne`, `Lt`, `Le`, `LtS`, `LeS`. The "`-S`" variants operate on signed integers.
- Bitwise boolean operators: `And`, `Or`, `Not`.
- `HeapAlloc` and `HeapAllocZ` take an integer — the number of bytes of memory to allocate — and returns the address of newly-allocated memory. The "`-Z`" variant also initalizes the memory to all zero.
- Output: `PrintInt` and `PrintIntS` print out unsigned and signed integers, respectively. `PrintString` prints out strings. These do not return a value.
- `Error` is for abnormal program termination (for errors like null pointer deferences, etc). It takes a string message to display to the user.
- `DebugPrint` is only for debugging. It accepts any number of values and prints out the interpreter's internal representation of the value. This can be useful for getting information about pointers.

```
a = HeapAlloc(20)
b = 12
a = Add(a b)
s = "Hello"
PrintInt(13)
PrintString(s)
```