Insert here your thesis' task.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master’s thesis

SimpleObjectMachine implementation

Bc. Rudolf Rovňák

Department of Theoretical Computer Science
Supervisor: Ing. Petr Máj

May 2, 2021

Acknowledgements

THANKS (remove entirely in case you do not with to thank anyone)

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 2, 2021 . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Rovňák, Rudolf. *SimpleObjectMachine implementation.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

V několika větách shrňte obsah a přínos této práce v českém jazyce.

Klíčová slova Replace with comma-separated list of keywords in Czech.

Abstract

Summarize the contents and contribution of your work in a few sentences in English language.

Keywords Replace with comma-separated list of keywords in English.

Contents

List of Figures

Chapter 1

Introduction

In the last decades, a trend of dynamic programming languages [1] has been on the rise. As opposed to static programming languages (usually compiled) dynamic ones offer a higher level of abstraction and allow faster and less error-prone development. Dynamic languages move a lot of actions traditionally done during compile-time to run-time. This creates the need for another layer, *a runtime environment.*

My goal in this diploma thesis is to implement a process virtual machine for a programming language called SOM, or Simple Object Machine. It is a dynamic, object-oriented programming language based on Smalltalk. It was originally implemented at University of Århus in Denmark to teach object oriented VMs [1]. There are several implementations in various programming languages, ranging in speed, optimizations etc.

My main focus in my work will be the clarity of implementation over performance. I will try to provide a basic implementation of a traditional runtime VM that can be built upon in the future. This will include the process of parsing, compiling the bytecode and then providing a runtime environment, along with an implementation for most basic principles (flow control, basic data types, loops).

[1] Not to be confused with *dynamically typed programming languages.*

Chapter 2

Analysis and design

Simple Object Machine (SOM) is a minimal Smalltalk dialect used primarily for teaching construction of virtual machines. Key characteristics according to official website ([1]) are:

- clarity of implementation over performance,
- common language features such as: objects, classes, closures, non-local returns
- interpreter optimizations, threading, garbage collectors are different across various implementations.

2.1 Grammar

To implement a parser for the language, I decided to use ANTLR. I will demonstrate language features and design on the following ANTLR grammar for SOM. For the sake of brevity, I ommited terminal symbols from the complete grammar as they are self-explanatory. All the terminal symbols in this grammar are named in uppercase letters.

```
grammar SOM;

classDefinition:
IDENTIFIER EQUALS superclass
instanceFields method*
(SEPARATOR classFields method*) ?
CLOSE_PAR
;
superclass: IDENTIFIER? OPEN_PAR;
instanceFields: (VBAR variable* VBAR)?;
classFields: (VBAR variable* VBAR)?;
```

```
method: pattern EQUALS methodBlock;
methodBlock: OPEN_PAR blockContents? CLOSE_PAR;
blockContents:
(VBAR localDefinitions VBAR)?
blockBody;
localDefinitions: variable*;
blockBody:
  RETURN result
| expression (PERIOD blockBody?)?;
result: expression PERIOD?;
expression: assignation | evaluation;
assignation: assignments evaluation;
assignments: assignment+;
assignment: variable ASSIGN;
evaluation: primary messages?;
primary: variable | nestedTerm | nestedBlock | literal;
messages:
  unaryMessage+ binaryMessage* keywordMessage?
| binaryMessage+ keywordMessage?
| keywordMessage;
unaryMessage: IDENTIFIER;
binaryMessage: binarySelector binaryOperand;
binaryOperand: primary unaryMessage*;
keywordMessage: (KEYWORD formula)+;
formula: binaryOperand binaryMessage*;
nestedTerm: OPEN_PAR expression CLOSE_PAR;
nestedBlock:
NEW_BLOCK blockPattern? blockContents? CLOSE_BLOCK;
blockPattern: blockArgs VBAR;
blockArgs: (COLON argument)+;
variable: IDENTIFIER;
pattern: unaryPattern | keywordPattern | binaryPattern;
unaryPattern: unarySelector;
unarySelector: IDENTIFIER;
binaryPattern: binarySelector argument;
keywordPattern: (KEYWORD argument)+;
binarySelector:
VBAR | PLUS | MINUS | EQUALS | MULT | DIV | MOD |
GREATER | GREATER_EQ | LESS | LESS_EQ;
argument: variable;
literal: literalNumber | literalString | literalArray | literalSymbol;
literalNumber: MINUS? (INTEGER | DOUBLE);
literalString: STRING;
literalArray: POUND NEW_BLOCK literal* CLOSE_BLOCK;
```

```
literalSymbol: POUND (STRING | selector);
selector: binarySelector | keywordSelector | unarySelector;
keywordSelector: KEYWORD+;
```

2.2 Class definition

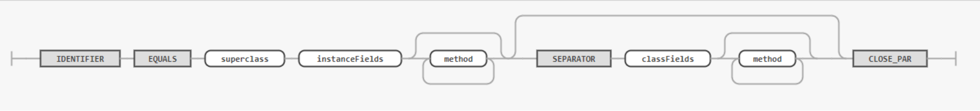

Figure 2.1: Railroad diagram for `classDefinition` rule.

```
SimpleHello = (
  | name |

  setName: aString (
    name := aString
  )

  printGreeting (
    ('Hello, ', name) print
  )
)
```

Syntax for class definition follows the official SOM grammar. The language supports single inheritance as apparent from the use of `subclass` token in the grammar. Not every class has explicitly specified superclass, therefore the actual identifier in the rule is optional.

Declaration of instance side fields follows, denoted by vertical bars. This token itself can be empty. Instance side methods definitions are next. Further details on *methods* and *messages* in SOM are discussed in TODO. Same syntax is used for class side fields and methods separated by a special token.

2.2.1 Variables

In Smalltalk, a variable is defined as *"a dynamically modifiable association (binding) of either a name or an index to a value. Each distinct variable has exactly one name (or index)"*[2].

A value of a named variable can be any object. Indexed variables are at the core also just an object. They represent an ordered sequence of objects as a single value. Example of those are arrays or Strings. Actual indices are always strictly positive (greater than zero), meaning the first element of an array corresponds to index of value one. This is standard in Smalltalk dialects, although uncommon in C-like languages. Retrieving the values belonging to an index is done via sending a message to the encapsualting object.

When creating a new variable, it is assigned a special value `nil`, meaning the variable is empty. This special value can also be explicitly assigned to a variable at any point.

SOM is a dynamically typed programming language (as is Smalltalk). As a result, there is no syntax to indicate a data type of a variable. One thing worth pointing out is that in the context of Smalltalk, a *data type* is defined differently than most programming languages. As stated in [2], a class is not a type. A Smalltalk type is defined as *"the power set of messages to which an object can meaningfully respond"*[2]. This is the definition I will be using in the context of SOM. As a result of this, any number of SOM classes can implement one data type.

2.2.1.1 Variable name scoping

Every variable has its scope, which determines the visibility of the variable. SOM follows the rules of Smalltalk when it comes to scoping, as defined in [2]:

- **Local variables** are accessible within the method or code block in which they are defined.
- **Formal method arguments** are accessible by the method wherein they are defined.
- **Formal block arguments** are accessible by the block wherein they are defined.
- **Instance variables** are accessible within all methods of a given object. Each object has its own instances of these variables.
- **Class variables** are accessible by all objects that are instances of the class or its subclasses. All the objects share the same instance of this variable.
- **Global variables** are accessible everywhere.

There is some more details on variable scoping, especially when talking about block closures. Further detail is discussed in 2.5.1.

2.2.2 Literals

Outside of variables, there is also a need to represent fixed values in a SOM source code.

Integer literal specifies a value of a decimal whole number, positive or negative. In my implementation, every integer literal is a representation of an object of class `Integer`. To achieve simple manipulation with integer numbers, all integers are internally represented by a C++ data type `int32_t` – therefore

32 bit signed integer number. Overflow and underflow is not addressed in my implementation, therefore the behaviour copies that of the underlying C++ type.

Floating point literal approximates a value of a real number. Syntactically, it consists of a decimal, possibly negative, integer literal representing the non-fractional part of the number. It is followed by a decimal point and another decimal (non-negative) integer representing the fractional part of the number. Precision is implicitly given and there is no way to change it. My implementation uses double precision (as defined in C++). Edge cases (such as rounding errors) are not addressed – the behaviour copies that of standard C++ double data type.

String literal represents a sequence of characters. String literals are objects of class `String`. Syntactically, they are delimited by single quotes (`'`). To include a single quote in a string, it needs to be escaped by another single quote.

In the provided implementation of SOM, String objects are not treated as a collection. It is an encapsulated object and every String object is unmutable. Every message, that somehow uses and modifies the value of its receiver creates a new object. This behaviour corresponds with Smalltalk and other SOM implementations.

Array literals specify a sequence of values encapsulated by a single object (that is an instance of class `Array`). Syntactically, the values of an array are surrouned by parentheses and preceeded by a hash sign. Note that because of the dynamic typing, elements of an array do not have to be instances of the same class.

Every Array literal results in an Array object instantiation. Arrays are mutable, unlike Strings.

2.3 Primitives

Even though SOM is purely object oriented, in order toto get any actual computations done, there is a point where some virtual machine primitives must be invoked. Following things are therefore implemented as primitives:

- memory allocation (`new` message),
- bitwise operations,
- integer arithmethics (`+`, `-`, `=` etc.),
- array accessing (`at:`, `at:put:`)

Primitives are implemented directly in the VM runtime, though not breaking the syntax or core principles of the language. Details on implementation are in section Primitives.

2.4 Methods and messages

As the SOM language is based on Smalltalk, the concept of messages (and the link to methods) is crucial to understand. *"The only way to invoke a method is to send a message – which necessarily involves dynamic binding (by name) of message to method at runtime (and never at compile time). The internals of an object are not externally accessible, ever – the only way to access or modify an object's internal state is to send it a message"* [2].

Execution of an invoked method ends with the execution of the last expression in it. Every method implicitly returns `self` (a reference to the object on which the method is invoked). Explicit return of a value is done with a special token `^`. Execution of an expression preceeded by this token will exit the method.

The [3] defines a helpful terminology for message passing:

- A message is composed of the message *selector* and the optional message arguments.
- Every message must be sent to its *receiver.*
- Message and its receiver together will be referred to as *message send.*

There are three types of messages (as defined in other Smalltalk dialects, Pharo as an example of one).

Unary messages are sent to an object without any additional information (argument). In the following example, a unary message `size` is sent to a string object.

```
'hello' size "Evaluates to 5"
```

Binary messages are a special type of messages that require exactly one argument. The selector of a binary message can only consist of a sequence of one or more characters from the set: +, -, *, /, &, =, <, >, —, and @. A very simple example of usage of binary messsage are arithmetic operations.

```
3 + 4 "Evaluates to 7"
```

Keyword messages require one or more arguments. From the syntactic standpoint, they consist of multiple keywords, each ending in colon (:). When sending a message, each keyword is followed by an argument. Note, that a keyword message taking one argument is different to a binary message.

```
| numbers |
numbers := #(1 2 3 4 5). "Simple array"
"Sending a keyword message at:put: to an object of class Array"
numbers at: 1 put: 6 "numbers is now #(6 2 3 4 5)"
```

When composing messages of various types, there are precedence rules (as defined for Pharo in [3]):

- Unary messages are sent first, followed by binary messages. Keyword messages are sent last.
- Messages in parentheses are sent before other messages.
- Messages of the same kind are evaluated from left to right.

These simple rules permit a very natural way of sending messages, as demonstrated on the next example. First, a simple array is created. Then, a unary message `last` is evaluated, returning the last element of the array. After that, binary message `+` is evaluated (to 2 in this example). Finally, keyword message `at:put:` is sent to an array, putting number 5 on the second position in an array.

```
| numbers |
numbers := #(1 2 3 4 5).
numbers at: 1 + 1 put: numbers last.
"numbers at: (1 + 1) put: (numbers last)"
```

Next example demonstrates sending messages from left to right when all of them are of the same type.

```
| numbers |
numbers := #(1 2 3 4 5)
numbers last asString print
"This is equivalent to the following message sends"
((numbers last) asString) print
```

There is a downfall to the simplicity of these rules. Arithmethic operations are all just a simple binary message sends, therefore to ensure proper precedence, it is necessary to use parentheses.

2.5 Blocks

Blocks provide a mechanism to defer the execution of expressions [3]. Blocks can be treated as an object – they can be assigned to variables and passed as arguments.

Blocks can also accept parameters – they are denoted with a leading colon. Parameters are separated from the body of the block by a vertical bar. Local variables can also be declared inside a block. Those are accessible only inside the block and are initialized each time a block is evaluated.

Block is executed by sending it a message `value`. However, this is a unary message and there is no way to pass parameters to a block. To solve this problem, a keyword message `value:` is implemented. So far, this gives a user to pass only one parameter to a block. To mitigate this issue, there are two posibilities. The first one is to implement a keyword message for every number of parameters (for example `value:value:`, `value:value:value:`). While this approach is simple, readable and relatively easy to implement for low numbers

Figure 2.2: Example of blocks usage in SOM.

```
| b0 b1 b2 b3 |
b0 := [ 1 + 2 ].
b1 := [ :x | x * x ].
b2 := [ :x :y | x * y ].
b3 := [ :x :y :z | x + y + z ].
"Evaluating the blocks"
b0 value. "Returns 3"
b1 value: 3. "Returns 9"
b2 value: 2 value: 8. "Returns 16"
"Message valueWithArguments: can be used with any number of parameters"
b3 valueWithArguments: #(1 2 3). "Returns 6
"The next expression is functionally identical to the previous one"
b3 value: 1 value: 2 value: 3.
```

of parameters, it is impossible for this solution to be exhaustive and the code using very long keyword messages would be bloated.

Another approach would be to implement a keyword message `value:` with an argument of array type. This would permit to use arbitrary number of arguments, though it would require to create arrays of objects before passing them to a block, which could impact readability and clarity of the code. In order to combine pros and cons of these 2 approaches, I have decided to follow the implementation in Pharo according to [3, p. 65]. There are keyword methods implemented for up to four parameters (`value:`, `value:value:`). For more than four parameters, a special keyword message `valueWithArguments:` is implemented, where an array of parameters is expected.

2.5.1 Non–local return and block scoping

Block closures are an essential feature to SOM. They allow the implementation of conditionals and loops as messages rather then them being baked in the language syntax. Blocks however bring some dynamic runtime semantics that is not straightforward. When used to the extreme, blocks can introduce some confusion and generally ugly code. However when used correctly, they offer great way to improve readability and reusability of the code.

Every method has its defined context – a set of variables and objects accessible from the method at given point in an execution. Variables accessible by blocks are bound during runtime, in the context of where the block is *defined*, rather than executed. This is reflected in creation and handling of frames (which can be considered a representation of the context of given method). This also ties in to how returns from blocks function.

The context in which the block is created (and evaluated) is commonly called as *home context* of the block. The block home context is basically a

```
expression: assignation | evaluation;
assignation: assignments evaluation;
assignments: assignment+;
assignment: variable ASSIGN;
evaluation: primary messages?;
primary: variable | nestedTerm | nestedBlock | literal;
messages:
  unaryMessage+ binaryMessage* keywordMessage?
| binaryMessage+ keywordMessage?
| keywordMessage;
```

representation of a particular point in the program execution. When a return statement is executed, the execution steps out of the current context and returns to the caller. This can be an implicit return (every method or block implicitly returns the receiver of the message). User can decide to return a different value, denoted by explicit return statement, (ˆ token).

The behaviour of explicit return in blocks is where the term non–local return comes in. *Non local return returns to the sender of the block home context, i.e., to the method execution point that called the one that created the block* [4]. The important thing is that the home context is tied to the creation of the block, not its evaluation. The pitfall here is there can be a situation where home context of a block being evaluated could end before the block attempts a return to it. This will result in runtime error.

To ilustrate the non local behaviour better, consider a following example.

2.6 Expressions

According to [2], *an expression is a segment of code in a body of executable code that can be evaluated to yield a value as a result of its execution.* Expressions can contain another expressions.

Syntactically, an expression can consist of [2]:

- literal,
- variable/constant reference,
- message send,
- nested expression.

2.7 Control structures

In Smalltalk, there are no built-in control structures, unlike for example C++ or Java. SOM follows this principle from Smalltalk, therefore there are no grammatical rules for branching or loops.

The way controlling the flow of program works in SOM is, again, by sending messages. One big advantage of this approach is that the programmer can define their own control structures, simply by implementing classes and methods as needed.

To make working with SOM easier and faster, my implementation provides multiple message implementations, corresponding to the most used control structures in other programming languages. Syntax of these messages corresponds to other Smalltalk dialects.

2.7.1 Conditional branching

There are 3 messages that function as an if control structure. Selectors for these messages are `ifTrue:`, `ifFalse:`, `ifTrue:ifFalse:`. As apparent, they are keyword messages, the receiver is an instance of a Boolean class. All of these messages take blocks as arguments, then evaluating or not evaluating them based on the Boolean value. Figure 2.3 shows a simple example of usage.

```
"Subtracts b from a only if a is greater then b"
a > b ifTrue: [ a - b ].
a <= b ifFalse: [ a - b ].
"Subtracts the smaller number from the bigger one"
a < b
  ifTrue: [ b - a ]
  ifFalse: [ a - b ]
```

Figure 2.3: Example of messages functioning as *if*-control structures.

2.7.2 For loops

The simplest example of a for loop is iterating over a range of integers. There are 2 messages, `to:do:` and `to:by:do:`. The receiver of the message is an integer. The receiver of the message is the lower bound of the iteration, the argument for `to:` keyword is the upper bound, `by:` specifies a step of iteration, `do:` takes a block that is evaluated (note that the block has to have exactly one parameter, so it is possible to capture the value of index in every step).

This way of looping is also usable when iterating over arrays (or any indexable collection). As seen on figure 2.5, this method is not very concise, therefore a message `do:` is implemented. Array class implements a method corresponding to this message, iterating over every element of the array. It

```
"Prints all numbers from 1 to 10"
1 to: 10 do: [ :index | index asString printLn ].
"Prints all the even numbers between 1 and 100"
0 to: 100 by: 2 do: [ :index | index asString printLn ]
```

Figure 2.4: Example of simple for loops.

takes a block as an argument. The block has to have one parameter – that is the element of the array of the given step of the iteration.

```
| array |
array := #(1 2 3).
"Printing the elements by iterating over index"
1 to: array size do: [ :index |
  (array at: index) asString printLn ].
"Printing the elements by iterating over array"
array do: [ :element | element asString printLn ]
```

Figure 2.5: Examples of different ways of iterating over an array.

2.7.3 While loops

While loops are implemented as a unary message sent to a block that returns a boolean value. There are actually two messages, `whileTrue` and `whileFalse`. The first one repeats the evaluation of a receiver (a block) as long as it returns `true`. The second one, as the name suggests, does the same thing if the block returns `false` value. Example in figure 2.6 shows printing numbers from 0 to 10 using `whileTrue` message.

```
| index |
index := 0.
[ index asString printLn.
index := index + 1.
index < 10
] whileTrue
```

Figure 2.6: Example of while loops.

2.7.4 Class hierarchy

SOM supports single inheritance and it is a vital aspect of the language. As a purely object-oriented programming language, it takes full advantage of polymorphism.

Single inheritance permits only one superclass per class. The superclass can be explicitly defined. If it is not defined, every class is a subclass of special Object class. This therefore means that every object is an instance of the Object class or its subclass.

Tied with inheritance is a principle of late binding. The method to invoke is decided during runtime. The method lookup is simple – it follows the chain of inheritance, from the subclass to superclass (always ending at Object class).

2.8 Abstract syntax tree

2.9 Bytecode

The next step after constructing the AST is to compile it into a bytecode. The bytecode is saved in a binary file that can be interpreted. The structure of bytecode files and semantics and syntax of operation codes is described in the following sections.

2.9.1 Program structure

SOM program has a very simple structure consisting of:

1. **The constant pool:** This is a list of all the entities of the program. The choice of the word *entity* over *object* is intentional to avoid confusion with what objects are in OOP languages. Each entity can be accessed by its index.

2. **Entry point:** An index to a Method that is executed on program start. There can only be one entry point to a program. It is a unary method with selector `run`. It can be a member of any class of the program.

2.9.2 Program entities

All entities in the constant pool are one of these types:

1. **Nil entity** represents an undefined value.

2. **Int entity** represents a 32 bit signed integer number. It is used for LIT instrucions.

3. **Double entity** represents a double-precision floating point number.

4. **String entity** represents a value of string of characters of arbitrary length. It is used for constants in the program as well as to store all the identifiers to classes, method selectors and variables.

5. **Field entity** represents a variable in an object. It consists of one index to a string value that represents the name of the slot.

6. **Method entity** represents a method of an object. It holds and index to a string representing the selector, number of arguments (arity of the corresponding message), number of local variables and an array of instructions.

7. **Primitive entity** is a method that needs an implementation in the VM. These are used to handle constructs that cannot be expressed in the base language.

8. **Block entity** is a block of code. It holds the number of arguments and an array of instructions (similar to a method).

9. **Class entity** represents the structure of objects. It consists of an array of indices to all the fields of the object. Each one of these fields point either to a Field entity or a Method entity.

2.9.3 Instructions

- `LIT i` retrieves a constant value from the constant pool at the index `i` and pushes it on the stack. The item can be either integer, double or string value.

- `GET SLOT i` pops a value from the operand stack, assuming it is an object. Then it retrieves a value with index `i` from the constatns pool, assuming it is a string. It then retrieves the value stored in the slot with the name specified by the string and pushes it onto the stack.

- `SET SLOT i` pops a value from the stack. This value is then assigned to an instance variable with identifier at index `i` in the constants pool.

- `SEND i n` sends a message to an object, which in most cases results in calling a method. A new frame is created on the execution stack, arguments are pushed and the execution jumps to the first instruction of the method.

- `GET LOCAL i` retrieves a local variable with an index `i` and pushes it to the top of the stack.

- `SET LOCAL i` pops a value `x` from the top of the stack and then assignes the `x` into a local variable with the index `i`.

- `GET SELF` retrieves the callee of executed method. The object is pushed to the top of the stack.

- `GET ARG i` retrieves the i–th argument of the current message from the stack and pushes it on top.
- `BLOCK i` creates a code block object. The argument `i` points to a block value in the constant pool. The block object is instantiated on the heap and pushed to the top of the stack.
- `RET` is used to return from a method call. The value from the top of the stack is returned. The address to return to is retrieved from the current frame, then the frame is popped and execution jumps to an instruction after the `CALL` that invoked the method.
- `RETNL i` - non local return. The value at the top of the current frame is used as the return value. Argument `i` specifies the number of frames to be popped, then the return value is pushed to the top of the current frame.

2.9.4 Garbage collection

The process of *garbage collection* performed by *garbage collector (GC)* is the process of allocating and freeing memory during application runtime. The main advantage of this mechanics is to prevent *memory leaks* – parts of a program that allocate memory without freeing it when it is not needed [5]. Most modern high-level programming languages implement some form of garbage collection.

There are multiple possible algorithms that solve this problem. I have decided to implement a mark and sweep algorithm, due to its simplicity. The expectation is that this algorithm will not be particularly fast, though it leaves a lot of room for possible improvements and it can be used as a demonstration of performance effects of a garbage collector.

2.9.4.1 Mark and sweep

According to [6], the algorithm consists of two main phases:

- Mark phase – discovery of all the reachable objects.
- Sweep phase – clearing the heap of all unreachable objects.

Every object allocated on the heap holds a mark bit. This represents the reachability of the object – 0 (false) for unreachable, 1 (true) for reachable. Upon object creation, the mark bit is set to zero. In the sweep phase a simple graph traversal algorithm can be used (such as DFS – Depth First Search) to mark all the reachable objects. Every objects can be considered a node and variables serve as neighbour lists. Root nodes would be local variable and fields that are directly accessible.

In the sweep phase, the whole heap can be traversed linearly and objects with mark bit set to zero are deleted. As the [6] states, main advantages of this algorithm are:

- algorithm handles cyclic references and therefore cannot end up in an infinite loop,
- no additional overheard during the execution of the algorithm (such as extra data structures etc.).

On the other hand, the simplicity of the algorithm means there are some disadvantages:

- normal program execution is suspended during the garbage collection process,
- the algorithm does not address memory fragmentation.

The problem of memory fragmentation means that after several runs of the algorithm, the reachable objects will be separated by chunks of free memory. This can be solved by shuffling the objects, though at a cost of further performance degradation.

To demonstrate the execution of the algorithm, consider the following object hierarchy allocated during a program execution. Each node represents an object allocated on the heap, with marked bit set to false. Arrows represent variable references from on object to another. In reality, there will be multiple root objects – therefore the algorithm can run for every root object.

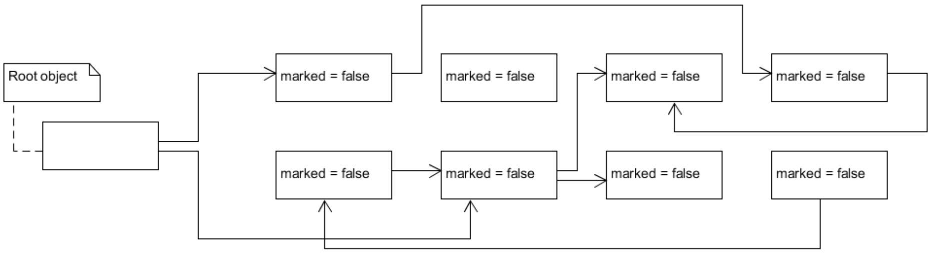

Figure 2.7: Illustration of the object hierarchy before GC algorithm run.

The heap is then traversed from the root object (figure 2.8). Each colour represents one step of the traversal. This example represents breadth first search.

After the run of the mark phase, all the objects reachable from the root objects (or multiple root objects after a run for each one) are marked. One linear traversal over objects can then remove all of the unmarked ones (figure 2.9).

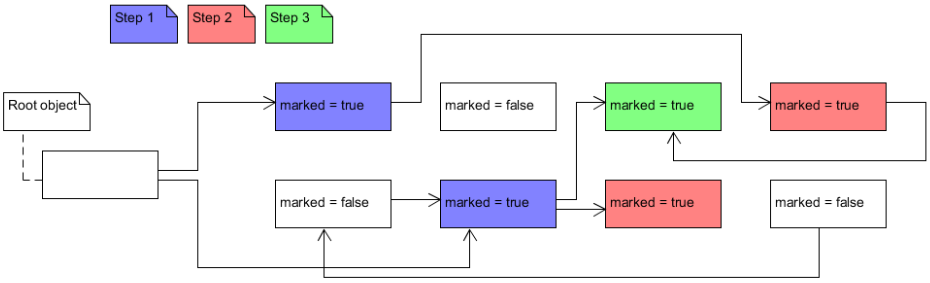

Figure 2.8: Traversal of the heap objects.

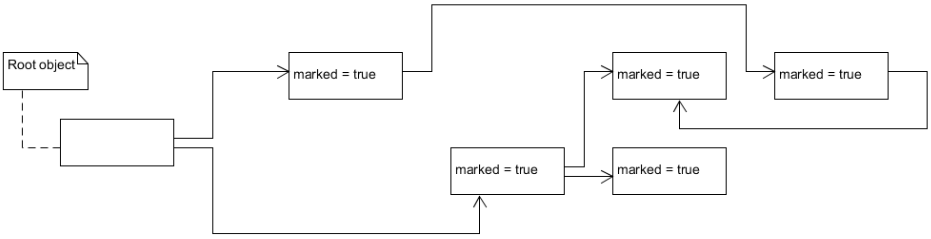

Figure 2.9: State of the heap after the GC run.

Chapter 3

Realisation

3.1 Program overview

The program I have implemented provides a way to compile SOM source code and execute it.

```
<som_executable> [OPTION] [SOURCE]
```

Figure 3.1: Interface of the program.

The interface of the application is simple and consists of two user provided arguments.

The argument `OPTION` can have two values and alters the mode the app will function in:

- `-c` is compile mode. The argument `SOURCE` is a folder containing the source code of the program. This folder is searched for SOM source files – those are recognized by their file extension, which should be `.som`. The folder is searched non recursively. Every SOM source file is compiled and one binary file containing bytecode is created. The name of the file is the same as the provided folder name.
- `-r` loads and runs a compiled bytecode. The `SOURCE` argument is a file name of the compiled bytecode.

3.2 Abstract Syntax Tree

After the parsing is complete, Abstract Syntax Tree (AST) is constructed. AST is, by definition, stripped of many syntactic detail. It mainly represents the structural and content-related aspects of the code.

The conceptual design of the AST is depicted on figure 3.2.

3.2.1 AST Nodes

Class node represents a class in the program, while the program itself is basically an array of different classes. A class holds its name, its member fields (member variables in C++ terminology), instance-side methods (member functions), class-side fields (static member variables) and class-side methods (static member functions).

Method node represents a method – instance or class side. It consists of a pattern, local variable definitions and a block to be executed.

Pattern represents a message corresponding to the method. There are 3 types of messages in SOM, therefore there are 3 distinct types of patterns. The simplest one is **unary pattern** – consisting of only one identifier as there are no arguments. **Binary pattern** is treated as a separate pattern. It consists of identifier and exactly one argument. There are special requirments for binary pattern identifier – there is a special set of characters permitted that can form a binary pattern. **Keyword pattern** then consists of one or more keywords and same number of arguments, each correpsonding to one keyword. Concatenation of keywords form a selector of the method. **Keywords** holds the string value of the keyword, always ending in colon (:).

Variable node represents instance/class side variables, arguments to messages or blocks. It holds the identifier of the variable as a string value.

Block represents a block of executable code with its own scope. The simplest block consists of local variable definitions and an array of expressions to be evaluated. While this is enough to represent a method block, other uses may require more information, therefore there is another similar node discussed later.

3.2.1.1 Expressions

Expression is an abstract term in the context of the AST - there are two types. The common thing is they can be evaluated – therefore forming the actual executable code of the program.

Evaluation is the first form of expression - it represents a message sends to an object, thus returning a single value when evaluated. This node consists of messages (optional) and a *primary*.

Primary is another abstract concept. In its core, a primary represents an object, though there are multiple ways to reference an object in SOM. There are four AST nodes that can be classified as a primary:

- **Literal** – a constant basic value (of integer, floating point, string or array type). Each of these have their dedicated literal node holding the value as seen on figure TODO.

- **Variable** is self explanatory – a reference to an object accessed via the identifier.

- **Nested term** is an expression that needs to be evaluated to retrieve the reference to an object. Syntactically, the nested terms are enclosed in parentheses.

- **Nested block** is a block of expressions returning the reference to an object. It is enclosed in square brackets in the syntax. Nested blocks consist of the same elements as block discussed with methods with addition of a block pattern – nested block can have their arguments.

The second part of the evaluation node is the message sends to the primary. There are three types corresponding to three types of messages in SOM. `UnaryMessage` node is self explanatory – there are no arguments, only the message selector. `BinaryMessage` holds its selector too with addition of the argument. The argument of the binary message send can be a primary, along with unary message sends (because unary message sends take precedence). `KeywordMessage` is made up of the keywords (forming the selector) and something called *formulas*. Formulas are binary (and also unary) message sends, that take precedence over keyword messages.

Assignation is the second form of an expression. The name suggests this node represents assigning a value into a variable. Therefore the node consists of the `Variable` node to assign to and an `Evaluation` node returning the value to assign.

3.2.2 AST construction

The AST is constructed by visiting over the ANTLR–generated parse tree. The visitor is implemented in class `CParseTreeConverter`. This is a subclass of `SOMParserBaseVisitor`, which is a base visitor implementation provided by ANTLR that perform depth–first traversal over the parse tree. Some member functions in `CParseTreeConverter` are not overriden and make use of this default behaviour (which is just iterating over child nodes and visiting them).

3.3 Bytecode

After constructing the AST it is turned into a bytecode. The bytecode definitions are located in source files `Bytecode.h/Bytecode.cpp`. The actual compilation is implemented as a depth–first traversal of the AST, therefore a visitor pattern is used. There is an abstract class `ASTVisitor` that defines the interface of any AST visitor. This can be used to further implement visualisations of the AST or to add support for different bytecode instructions sets, for example Java bytecode to add support for running inside a Java runtime.

3.3.1 Values

Every constant value in the code is saved as a `Value` struct. Each value contains the one byte tag and the actual value to hold. There are method implementations to print every value into human readable format for better visualisation of the compiled bytecode. Additionally, every instruction struct is able to serialize itself – write the data needed in binary format to a file.

3.3.2 Instructions

Similar to values, every instruction is represented by a struct holding the relevant information (such as operation codes and arguments). Every instruction is capable of printing itself in human readable format and serialize itself to binary format, the same as all the values.

3.4 Interpretation

The process of interpreting the loaded bytecode is handled by class `CInterpret`. The class diagram of the interpretation environment is in the figure 3.3. The diagram serves as a reference point, all the parts are described in further detail in the following sections.

3.4.1 Program counter

The class `CProgramCounter` is an implementation for a program counter for the VM. During initialization of this object, the entry point of the program is loaded. The program counter also saves the ”end” address. The addresses are implemented as iterators to a vector of instructions – that is possible simply because all of the executable code will be contained within a method, and all the method are loaded into a vector of instructions. The entry point of the program is therefore the first instruction of the `run` method, while ”end” address is the end iterator of the same vector. This means that the end address is not an executable instruction.

3.4.2 Execution stack

The execution stack is responsible for management of method calls (and nested block evaluations). It is implemented as a LIFO (last in, first out) structure. For every method call or block evaluation, a new frame is created and pushed on the stack. A frame contains:

- Arguments accessible by their index, assigned at compilation. For the sake of stack, the callee is also considered an argument and is always the last element of the arguments array.

- Array of local variables, accessible by their index assigned during compilation.
- Return address. The program counter is set to this address when a `RET` instruction is executed.
- Local data for the method/block execution. This part is used for all the temporary object creation (e.g. literal values) and argument passing.

Sending a message or evaluating a block therefore can be split into multiple steps:

- The receiver of the message is pushed to the stack.
- The arguments are then pushed to the stack, in the order from left to right.
- When the `SEND` instruction is executed, new stack frame is created. The return address is initialized to the address of the instruction following the `SEND` instruction. Array of arguments is initialized with the values from the top of the stack, number of arguments is provided as an argument of the instruction.
- The new frame is pushed to the stack. The receiver is then accessed, added to the end of the argument array and given the responsibility to invoke the method corresponding to the sent message.

When returning, the top of the stack contains the return value of the method or block. The frame is popped from the stack, program counter is set to the return address and the top value is pushed to the new top of the stack.

3.4.3 Objects

Every object created during runtime (implicitly or explicitely) is represented by the `VMObject` class. Every value is represented by an instance of this class or its subclass. Every object holds a pointer to its class and a map of the instance values – the identifier as a key, object pointers as a value. Upon creation, instance field values are initialize to the `nil` value.

Every class is represented by a `VMClass` instance, which is a subclass of `VMObject`. This allows us to manipulate every class as an object. There is only one class instance per class definition during runtime and instantiating new class object is not possible. Before the execution of the code, the bytecode is traversed and for every `CLASS` value, a singleton object is created and initialized. These objects are accessible globally by the identifier – the class name.

Every class holds all the information needed to create and manipulate its instance. This includes the method dispatch.

3.4.3.1 Object creation

One of the main responsibilities of the class object is to handle dynamic creation of its instances. This is done via a `new` message send. Upon the `new` message send:

- new `VMObject` is created on the heap and its class is set,
- instance fields of the new object are initialized,
- pointer to the new object is returned.

3.4.4 Core library

SOM programming language is very light on features. In order for it to be usable, there needs to be an implementation of some fundamental principles provided. For example, the language itself does not provide a way to manipulate numbers, character strings, standard input and output or control structures. All of these can be implemented in the VM itself while preserving the consistency of rules of the language.

In order to achieve that, a keyword `primitive` is defined. This keyword is used for methods that require an implementation in the VM runtime. From the outside, caling a primitive method is no different than calling a method implemented directly in SOM.

While the user is able to implement their own primitives, a lot of them are already provided in the core library. This library is a set of classes loaded every time a SOM code is interpreted. This library provides implementations for:

- strings,
- numbers – integers and doubles,
- boolean values with messages that provided control flow features,
- code blocks,
- arrays.

3.4.5 Primitives

Every class that marks a method primitive has to provide its implementation in the VM. This is done by creating a new subclass of `VMClass`. This subclass then has to implement the primitive methods (in the form of `void` member functions that take `CInterpret*` as an argument). Resolving of method selector (which is a simple string) and the function to invoke is then done via member function `dispatchPrimitive`.

3.4.5.1 Strings

Every literal string value is represented by an instance of `String` class during runtime. The corresponding class implementing the primitives is `VMClass`. In SOM, every string object is immutable, therefore every string manipulation results in creating a new string object.

This approach can however lead to inefficiencies when constructing strings. To combat this, a class handling dynamic string creation could be implemented, similar to streams.

When looking at String implementation, there is a big difference to how they are implemented in SOM and Smalltalk. When a string object is created in Smalltalk, it is handled as an object with indexed fields, each containing a character. My implementaion of SOM does not conform to this, as strings are simply handled in the VM runtime. Possible extension of the language could be developed, allowing for creation of objects with indexed fields.

This could be the applied to arrays, thus creating a unified ways to handle collections.

3.4.6 Messages

In SOM, almost everything is handled via message sends. Message sends invoke a method, implemented either directly in SOM, or as a primitive. The process of invoking a method on an object depends on what kind of message is sent. Additionally, a small number of special cases needs to be handled for code blocks, to allow for iteration.

When a `SEND` instruction is encountered, the receiver of the message is retrieved from the stack. As the receiver is pushed to the stack first, before the argument, it needs to be retrieved using the message arity. To achieve the late binding, the selector (a string value) is then retrieved from the constants pool.

Reference to the class of the object than handles the method dispatch. Slight differences in method dispatch between primitives and native methods are discussed in further detail in section TODO.

Every method invokation results in a new frame being created on the execution stack. Because the execution stack is implemented as a stack of separate objects representing the frames (and not a continuous array), every new frame needs to be initialized. This handles the argument passing, as well as scoping of local variables when sending messages to code blocks.

When the new frame is created, multiple actions take place:

- Return address on the frame is set to the instruction following the `SEND` instruction. As every block of code must end with return (simple or non – local), the existence of the address is guaranteed.

- The array of arguments is initialized. Arguments had been pushed to the stack in the order of corresponding keywords (therefore left to right). As a result, the arguments are reversed in the actual array – member functions access them from the back. The last argument is set to the reciever – that is the object accessed by `self` keyword (comparable to `this` pointer in C++).

- Array of local variable is initialized. This step is skipped for every object outside of code blocks – instances of `Block` class. This is due to the scoping rules – code blocks can access the local variables of the method, in which they are evaluated.

After the new stack frame is created, the execution of the method can take place. For primitive values, the corresponding member function is called directly in the VM. For native methods, program counter is set to the address of the method and interpreting loop continues.

Method execution should always end with the return value at the top of the stack – for primitives and native methods. Every method should also end with the `RET` instruction. When this instruction is executed, the top value of the current stack is taken. The top frame of the stack is removed, program counter is set to its return address and the top value is pushed to the top of the underlying frame. Interpreting loop can then continue.

3.4.7 Block evaluation

From the outside, block evaluation is not very different to method execution. Actual work with blocks is also done entirely through standard messaging system. To achieve the functionality in the implementation as – is, some special cases need to be handled.

The block evaluation is done via `value` message (or the variants with arguments). The method itself is primitive, as full access to interpreting objects is needed.

As new frame is created on the stack on `value` send, the actual evaluation is done in that context. This provides one advantage in the form of handling arguments – the arguments provided to the message sends can be directly used by the actual block code, without any modification.

As per scoping rules defined in SOM, code blocks can access local variables of method that created them. However, every frame stack holds its own local variables. For that, a special case is handled when sending messages to block (`value messages`, `whileTrue:` etc.). The array of locals is first initialized with the values from the underlying stack. Every local variable of the block itself is then placed after that. The block object itself (`self` in this scope) is at the end, as standard.

3.4.7.1 Iteration

To implement iteration easily and without the need for a lot of recursive calls, method `restart` is implemented for blocks. This message restarts the execution of the block, without the need to call `self value`, which would result in a new frame creation, along with possible arguments and local variables initialization.

In order for this method to work, every frame needs to hold a starting address of the corresponding method. Then the program counter can be set to start of the block and executed again.

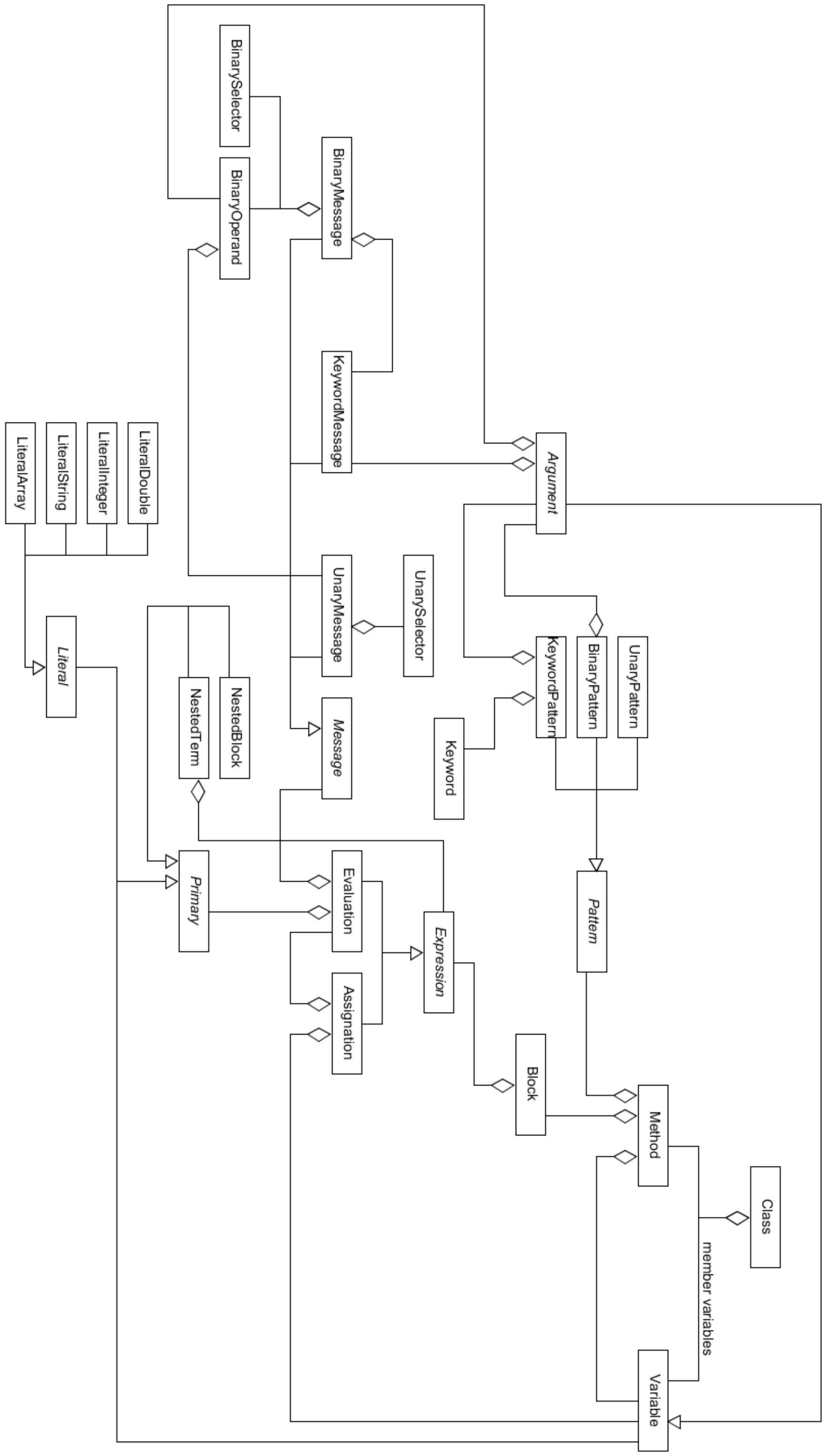

Figure 3.2: Conceptual class diagram for SOM abstract syntax tree.

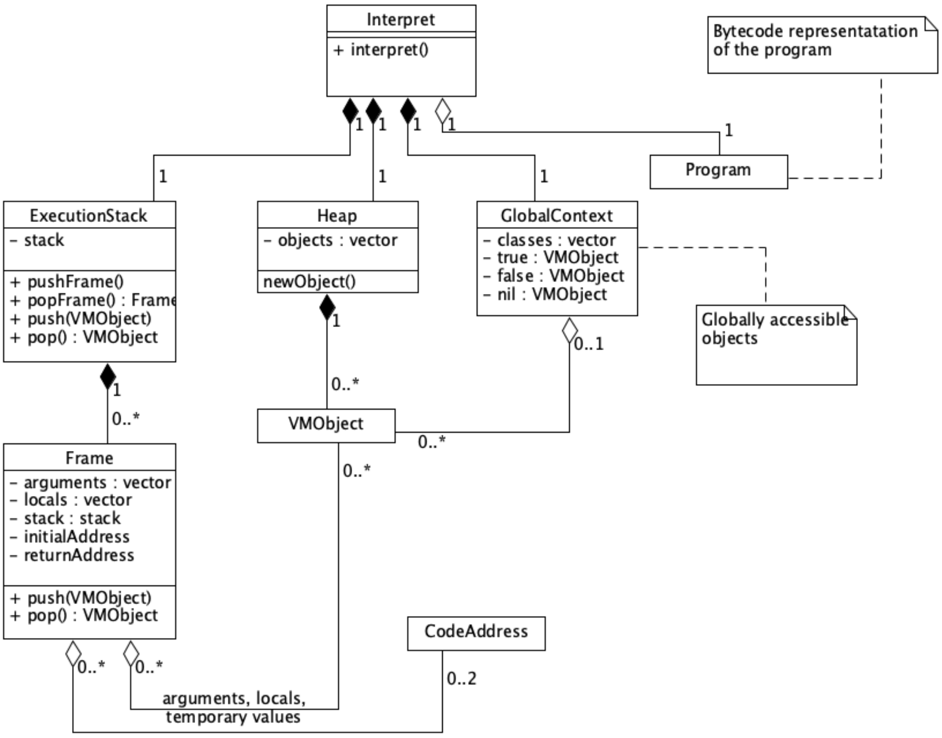

Figure 3.3: Conceptual class diagram for interpretation environment.

Conclusion

Bibliography

[1] SOM. SOM: A minimal Smalltalk for teaching and research on Virtual Machines. 2020, [cit. 2020-11-17]. Available from: `https://som-st.github.io/`

[2] Lovejoy, A. L. Smalltalk: Getting The Message. 2007, [cit. 2021-1-19]. Available from: `http://devrel.zoomquiet.top/data/20080627141054/index.html`

[3] Ducasse, S.; Chloupis, D.; et al. Pharo By Example 5. 2017.

[4] Bera, C. Blocks: A Detailed Analysis. [cit. 2021-4-30]. Available from: `http://pharobooks.gforge.inria.fr/PharoByExampleTwo-Eng/latest/Block.pdf`

[5] Boersma, E. Memory leak detection - How to find, eliminate, and avoid. January 2020, [cit. 2020-11-5]. Available from: `https://raygun.com/blog/memory-leak-detection/`

[6] Agarwal, C. Mark-and-Sweep: Garbage Collection Algorithm. [cit. 2021-5-1]. Available from: `https://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/`

Appendix A

Acronyms

AST Abstract syntax tree

GC Garbage collector

SOM Simple Object Machine

VM Virtual machine

Appendix **B**

Contents of enclosed CD

- `readme.txt` the file with CD contents description
- `exe` the directory with executables
- `src` the directory of source codes
  - `wbdcm` implementation sources
  - `thesis` the directory of LaTeX source codes of the thesis
- `text` the thesis text directory
  - `thesis.pdf` the thesis text in PDF format
  - `thesis.ps` the thesis text in PS format