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

January 25, 2021

Acknowledgements

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

Declaration

In Prague on January 25, 2021 . . . .

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.

[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

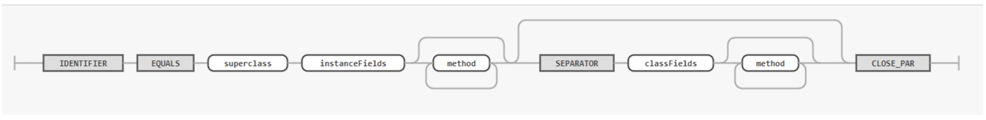

Figure 2.1: Railroad diagram for `classDefinition` rule.

```
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.

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.

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:`)

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.

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 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

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.
```

```
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;
```

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.

2.6 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.* As seen from the grammar snippet on figure TODO, expressions are recursive structures.

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).

```
"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.

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 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: Comparison 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

Protocol of an `Object` class is (this will probably be used somewhere further on):

- `class` - returns the class of an object,
- `=` - value equality comparison,
- `==` - reference equality comparison,
- `isNil` - check, if the object is `nil`,
- `asString` - converts the object into a string,
- `value` - evaluate (interesting for blocks),
- `print, println` - prints the object,
- `error:` - error reporting,
- `subClassResponsibility` - can be used to indicate the method should be implemented in the subclass of a given class,
- `doesNotUnderstand:arguments:` - can be used for error handling when a method is not implemented.

2.8 Interpretation

Once the source code is parsed, the next step is executing it – this step is called *interpretation.* Interpretation is As per [4], an interpreter for a language L can be defined as a mechanism for the direct execution of all programs from L. It executes each element of the program without reference to other elements.

It is however very rare that any language is interpreted directly. In most cases of non-trivial languages, the interpretation process is preceeded by parsing or compiling into some form of *intermediate representation.* According to [4], this process removes lexical noise (comments, formating), elements can be abstracted/combined (into keywords, operations etc.) and reordered into execution order (for example operators in an algebraic expression).

The choice of intermediate representation is therefore vital. It can determine a lot of aspects of interpretation - from the way of distributing the interpreted program to time and space complexity of the interpreter.

2.8.1 AST interpretation

Abstract syntax tree (AST) is a tree representation of the source code of a computer program that conveys the structure of the source code. Each node in the tree represents a construct occuring in the source code [5].

As the name suggests, AST represents the source code in the form of a tree. During the transformation from the source code to AST, some information is ommitted. Information that is vital for AST's according to [5] is:

- variables – their types, location of their definition/declaration,
- order of commands/operations,
- components of operators and their position (for example left and right operands for a binary operator),
- identifiers and corresponding values.

2.8.2 Bytecode interpretation

Using a form of bytecode. Effective, requires:

- designing the bytecode (instructions, bytecode file formats),
- AST to bytecode translation (AST -¿ bytecode instructions),
- actual bytecode interpretation.

Bytecode interpretation permits easier optimization.

2.9 Optimization

- dead code elimination,
- constant propagation,
- others...

2.10 Virtual Machine

Decide on memory hierarchy, garbage collection...

2.10.1 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 [6]. Most modern high-level programming languages implement some form of garbage collection.

Chapter 3

Realisation

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] Wolczko, M. Execution mechanisms Part I: Interpretation. [online], 2015, [cit. 2020-10-31]. Available from: `https://www.dropbox.com/s/lfav564dvx20qsw/2%20AST%20Interpretation.pdf`

[5] DeepSource Corp. Abstract Syntax Tree. [cit. 2020-11-4]. Available from: `https://deepsource.io/glossary/ast/`

[6] 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/`

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
```