Università
della
Svizzera
italiana

Facoltà
di scienze
informatiche

Bachelor Project

Sapphire

Scripting Smalltalk

Daniele Sciascia

supervised by
Prof. Dr. Laura Pozzi
Prof. Dr. Michele Lanza
Paolo Bonzini

Abstract

GNU Smalltalk is an open source project whose goal is to create a scripting environment based on the Smalltalk programming language.
The problem of creating such a scripting environment is that in Smalltalk there is no syntactic construct to define a class. This is an elegant solution when the system can rely on a development environment. In a typical Smalltalk environment it is possible to interactively define classes and store them on the virtual machine's image. This becomes a problem when trying to develop a scripting environment. In our case it is not possible to rely on them. That is, the programmer must be able to write down his classes on normal text files.
GNU Smalltalk faces this problem by adopting the so called bang-separated chunks format. It adopts this format as its language syntax. This solution actually solves the problem; but this format was intended to be used as a machine-readable format for sharing source code. As a result, reading and writing GNU Smalltalk scripts is hard.
We propose an extended Smalltalk syntax specifically designed to write GNU Smalltalk scripts. By doing so we provide a better programming experience. In particular, the aim is to make it easier to write code, but also to improve readability.

Contents

Abstract i

1 Introduction 1
1.1 Structure of the Document 2

2 Project Description 3
2.1 Project Context 3
2.1.1 Smalltalk-80 3
2.1.2 GNU Smalltalk 5
2.2 Scripting Smalltalk 6
2.2.1 Overview and problem statement 6
2.2.2 Goals 7
2.2.3 Solution and benefits 7

3 Implementation 9
3.1 Syntax 9
3.1.1 Class definition 9
3.1.2 Class extension 9
3.1.3 Instance variables 10
3.1.4 Class variables 10
3.1.5 Methods 10
3.1.6 Metaclass 11
3.1.7 Namespaces 11
3.1.8 Summary 11
3.2 BNF Grammar 12
3.2.1 Analysis and disambiguation 13
3.2.2 Parsing Table 14
3.3 Testing 14
3.4 Syntax Converter 15
3.5 Example 16

4 Discussion 18

5 Conclusions 19
5.1 Contributions 19
5.2 Future Work 20

Chapter 1

Introduction

The use of scripting languages represents today a convenient way of developing a variety of software systems. In this document we refer to *scripting language* as a programming language that is usually interpreted and dynamically typed. In some cases these languages are based on the Object Oriented paradigm. Examples are: Perl, Python and Ruby[1].
Trends of the past few years show that the adoption of such languages is increasing. Scripting languages are successfully used for web applications, system administration, prototyping, simulation and all sorts of other activities.
The reasons behind this trend cannot be easily summarized. Some of them relate to the concept of rapid application development; that is the ability to write a complete software system in a small amount of time. This is usually due to the fact that these languages come with a large set of available libraries. They represent, in some cases, easy to use high level languages that lead to easily maintainable software. The choice of whether to use a system programming language such as Java[2], or a scripting language such as Ruby strictly depends on the application being developed.

With this project, we propose an alternative scripting environment based on the Smalltalk programming language. Although it is not as widely adopted as others, Smalltalk is extremely well designed and introduced many concepts later adopted by scripting languages, including dynamic typing itself. Adopting Smalltalk as the base of a new scripting language represents an attractive alternative to the existing ones, especially for its expressiveness.

One of the problems when creating a scripting environment based on Smalltalk is that its design was heavily influenced by the introduction of IDEs (Integrated Development Environment). In fact the syntax of the language itself does not include syntactic constructs for the basic structures of a program. For example, there is no specific syntax for declaring a class. In a Smalltalk system, this is typically done interactively in the development environment. This is an elegant solution, because in this way the language exposes just a minimal syntax, while declaration of classes can be easily ”hidden” in the development environment.

[1] see respectively: www.perl.org, www.python.org and www.ruby-lang.org
[2] www.java.sun.com

In particular, for this project, we focus on GNU Smalltalk[3], an open source implementation of Smalltalk, whose aim is to provide such a scripting environment. As the GNU Smalltalk manual[4] explains:

> "[...] the goal of the GNU Smalltalk project is currently to produce a complete system to be used to write your scripts in a clear, aesthetically pleasing, and philosophically appealing programming language."

GNU Smalltalk addresses the problems discussed earlier by using the so called *bang-separated chunks* format as its language syntax. This format was originally designed for exporting source code created in a development environment so that it was possible to share it. It is not intended to be used as a language to create Smalltalk programs (even if it is possible to do it), nor it is designed to have a comfortable and easy-to-use syntax.

The main goal of this project is to provide an alternative syntax, specifically designed for writing GNU Smalltalk scripts.

1.1 Structure of the Document

Chapter 2 starts with a description of the basic concepts behind Smalltalk-80 and the differences found on GNU Smalltalk. While the second part of the chapter presents a complete description of project.

Chapter 3 describes the implementation details of the project. In particular it presents the new syntax and the how the system was implemented and what was achieved.

Chapter 4 discusses the project in terms of strengths and weaknesses in the design and implementation.

Chapter 5 draws conclusions about the whole project and indicates possible future directions.

[3]www.gnu.org/software/smalltalk/
[4]www.gnu.org/software/smalltalk/manual/

Chapter 2

Project Description

2.1 Project Context

2.1.1 Smalltalk-80

This section should not be taken as a complete description of Smalltalk-80. It focuses on those parts that are relevant for the understanding of the rest of this document. We assume that the reader knows at least the basics of the object oriented programming paradigm.

Smalltalk and its environment

When referring to Smalltalk, we can think of an environment made of:

- a language
- a class library
- a development environment

Smalltalk, the language, is the ground on which everything else is based on. It provides the basic framework for communicating objects. Basically, the language allows to create objects described in classes, and exchange messages between the created objects. The rest of the features, from the most basic ones like predefined data types and control flow, are entirely delegated to the class library. The language is therefore quite small and elegant.

As in almost every language, the class library provides a set of predefined classes. In the case of Smalltalk, the entire class library is written in Smalltalk itself. This is a great benefit, because Smalltalk programmers have the entire source code of the class library in their hands.

As already said, Smalltalk provides a complete development environment. It is not only the place in which you see, write and debug your code, but it also shows the source of the class library. The development environment plays an important role and is considered an essential part of a Smalltalk system. Since the language is small and has a minimal syntax, the development environment provides a way to interactively

define programs and classes. Figure 2.1 shows the *System Browser* included in the development environment of Squeak[1].

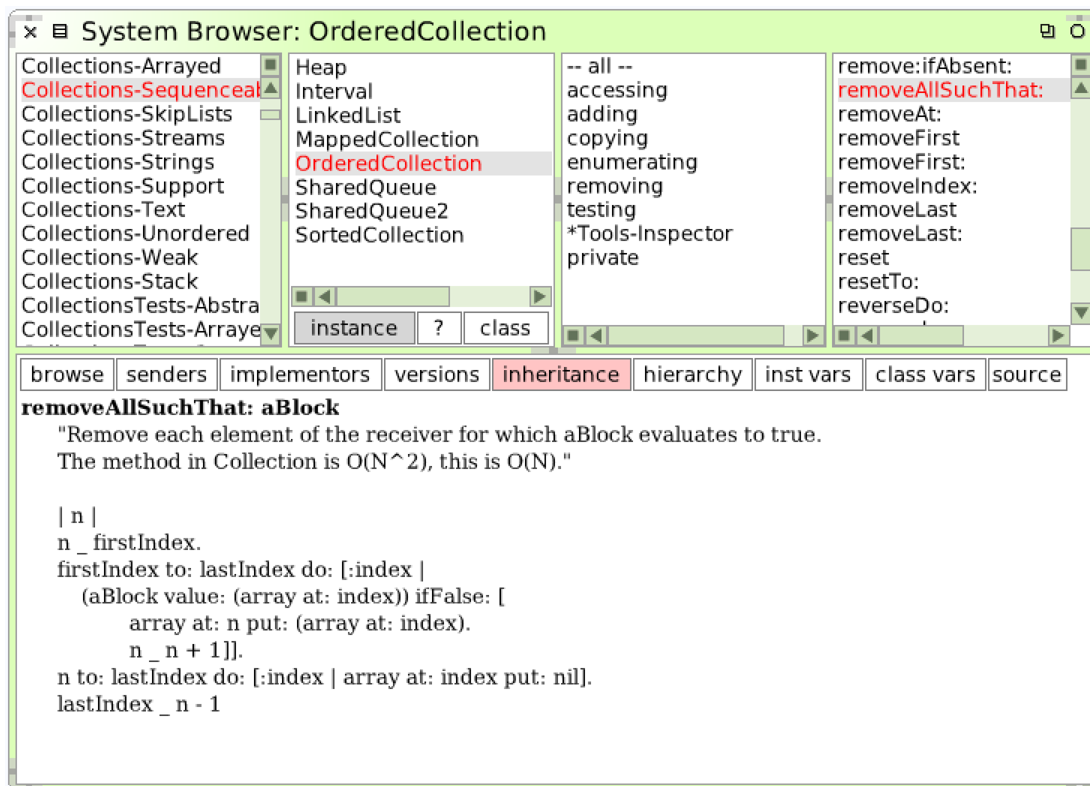

Figure 2.1: Squeak's System Browser

This is a typical code browser of a Smalltalk system. It provides five different panes:

- Leftmost pane - Shows categories under which available classes are grouped.
- Left center pane - Shows classes of the selected category.
- Right center pane - Shows categories of methods of the selected class.
- Rightmost pane - Shows methods of the selected category.
- Bottom pane - Shows the source code of the selected method.

The virtual machine and the image

Smalltalk runs on top of a *virtual machine* that isolates Smalltalk programs from the actual hardware. Programs are therefore truly portable. The virtual machine basically implements a bytecode interpreter.
However, another important piece of a Smalltalk system is the so called image. An image is a file that is used by the virtual machine. As figure 2.2 shows, the image integrates the class library, the complete development environment but also user defined classes. In particular, user defined classes are directly compiled into bytecode by the development environment.

[1]www.squeak.org

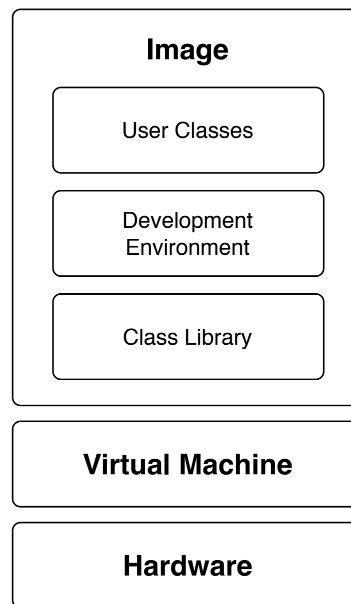

Figure 2.2: Smalltalk virtual machine and image

2.1.2 GNU Smalltalk

GNU Smalltalk is an open source implementation of the Smalltalk-80 programming language. This implementation strictly follows the design proposed in [GR83].
However the environment that GNU Smalltalk offers is fairly different from the usual IDE provided by other Smalltalk implementations like VisualWorks[2] or Squeak.
GNU Smalltalk offers a minimal working environment. Its environment is mainly a command line virtual machine. There are two ways to operate with this tool. The first one is to use as an interactive interpreter. The other one is to use it to launch a script previously written on a file.

The virtual machine and the image

In the case of GNU Smalltalk, the virtual machine is not only a bytecode interpreter, but it is also able to compile code. Instead of working on the image, in this case the virtual machine works on files that contain the user code. This solution does not prevent the virtual machine to take advantage of the image. In fact it is able to reconstruct an image that contains the class library and is used as a cache during execution.

File-out syntax

As already mentioned, GNU Smalltalk uses the file-out format or bang-separated chunks format as its language syntax. This format was described in [Kra83].
However, this format was intended to be used as an exporting format. The goal when creating this format was to allow programmers to share their source code across different implementations of Smalltalk development environments.

[2]www.cincomsmalltalk.com

Namespaces

A missing feature of Smalltalk-80 is the possibility to work in multiple namespaces. Classes are defined in one global namespace. Instead GNU Smalltalk implements multiple namespaces so that different classes and global variables, may have the same name given that they are defined in different namespaces.

2.2 Scripting Smalltalk

2.2.1 Overview and problem statement

We believe that using the bang-separated chunks format as GNU Smalltalk's language syntax is not an adequate choice. Some reasons of why a new syntax is needed can be intuitively seen by comparing a simple program written in a scripting language like Ruby and its corresponding implementation written for GNU Smalltalk:

```
# Ruby
class Person
  attr_accessor :name, :age

  def initialize(name, age)
    @name = name
    @age  = age.to_i
  end

  def inspect
    "#@name (#@age)"
  end
end


# GST (bang-separated chunks format)
Object subclass: #Person
       instanceVariableNames: 'name age'
       classVariableNames:    ''
       poolDictionaries:      ''
       category:              ''!

!Person class methodsFor: 'instance creation'!
    name: aString age: anInteger
        ^self new name: aString; age: anInteger; yourself!
!

!Person methodsFor: 'accessing'!
    name
        ^name!

    name: aString
        name := aString!
```

```
    age
        ^age!

    age: anInteger
        age := anInteger!
!

!Person methodsFor: 'printing'!
    printOn: aStream
        aStream << ('%1 (%2)' % {name. age})!
!
```

This example clearly demonstrates that the *bang-separated chunks* format is more uncomfortable, error prone and long than the corresponding Ruby code. Moreover, in GNU Smalltalk, there isn't a convention that implies one class for each file as in other languages. That is, a file could possibly contain more than one class. In such cases, the bang-separated chunks format results hard to understand, because there's no clear notion of scope.

2.2.2 Goals

The main goal of this project is to provide an alternative syntax for GNU Smalltalk. This new syntax should address the shortcomings of the bang-separated chunks format. The main problem lies in the fact that in Smalltalk there is no syntactic notion of a *class*. The new syntax should introduce this notion, with the only reason that this will help structuring source code in a better way. This should be done in a way that:

- writing code is natural and intuitive for programmers. The syntax should feel like a natural extension of the Smalltalk language, appropriate for writing scripts.
- it improves the readability by introducing scoped definitions. That is, classes and methods should have their own scope.
- it tries not to affect the rest of the syntax. Remember that this is an extension not a complete remake.
- the current parser can be extended, without the need of rewriting it completely.

A problem that arises when changing the syntax of the language is that software already written in GNU Smalltalk has to be migrated to the new syntax. That is why the second goal of the project is to provide a converter tool that takes as input a source file written in bang-separated chunks format and translates it into the new syntax. This tool has been written using GNU Smalltalk, and its new syntax. Notice that also GNU Smalltalk's class library is entirely written using the file-out format, and therefore has to be converted.

2.2.3 Solution and benefits

The following code snippet shows the same example program we used in Section 2.2.1, but this time written using the proposed syntax:

```
Object subclass: Person [
    | name age |

    Person class >> name: aString age: anInteger [
        ^self new name: aString; age: anInteger; yourself
    ]

    name [ ^name ]

    name: aString [ name := aString ]

    age [ ^age ]

    age: anInteger [ age := anInteger ]

    printOn: aStream [
        aStream << ('%1 (%2)' % {name. age})
    ]
]
```

This example does not cover all the syntactic elements provided by this project. These will be covered in Chapter 3. However, by comparing this example with the one shown in Section 2.2.1, we can notice the following:

- Increased readability.
- Instead of separating different pieces of code using exclamation marks, scopes are clearly delimited for both classes and method definitions.
- Apart from class and message definitions Smalltalk's syntax remains completely unchanged.
- Instance variables are defined in the same way as you would define local variables inside a Smalltalk method.
- All of the features of Smalltalk are still supported.

Chapter 3

Implementation

3.1 Syntax

This section presents a complete overview of the syntactic elements introduced in the grammar of GNU Smalltalk. Each element is described with a simple and commented example.

3.1.1 Class definition

In order to define a class, the new syntax provides the following construct:

```
Object subclass: MyClass [
]
```

where *MyClass* is the new class that has to be created, and *Object* is the class from which to inherit. An important fact to notice is that with the new syntax, we introduce also the notion of scoping. In fact, the ”contents” of *MyClass* have to be written between the ’[’ and ’]’ brackets.

3.1.2 Class extension

Smalltalk provides also a way to redefine the ”contents” of a class that was defined earlier. Classes can be extended in the following way:

```
MyClass extend [
]
```

where *MyClass* is an already defined class. In the same way, it is possible to extend the metaclass of an already defined class:

```
MyClass class extend [
]
```

3.1.3 Instance variables

Instance variables are defined in the same way local variables are defined inside a method. That is instance variables names must be listed between '|' and '|':

```
Object subclass: MyClass [
    | anInstanceVariable anotherOne |
]
```

3.1.4 Class variables

Class variables are included in the class scope as the other kinds of variables with the only difference that they can be initialized on the fly. This choice is due to the fact that class variables are usually used for constant values. For example:

```
Object subclass: MyClass [
    AClassVariable := nil.
]
```

3.1.5 Methods

Smalltalk provides three method patterns: unary, binary and keyword methods. GNU Smalltalk obviously provides all of them, with the only difference that methods get their own scope, between angle brackets, containing local variables and method statements:

```
Object subclass: MyClass [
    aUnaryMethod [
    ]

    < arg [
        "This is a binary method"
    ]

    aKeywordMethod: arg1 with: arg2 [
    ]

    MyClass class >> aClassMethod [
    ]

    ASuperClass >> overriddenMethod [
    ]

    ASuperClass class >> overriddenClassMethod [
    ]
]
```

In addition, the example above shows another feature, that is the ability to define methods in a superclass. This feature becomes useful in a number of situations, for instance assume you are defining a new data type *Foo*. In that case you might want to create the method *#isFoo* in class Object, that simply returns *false*. And then override that method in class Foo so that it returns *true*.

3.1.6 Metaclass

In order to define a class instance variable, we have to ”access” the metaclass. This can be accomplished in the following way:

```
Object subclass: MyClass [
    MyClass class [
        | aClassInstanceVariable |

        aClassMethod [
        ]
    ]
]
```

Notice that the metaclass scope behaves exactly as the class scope. Whatever can be done in the class scope, can also be done in the metaclass scope. This means, as the example shows, that we can define a method inside that scope.

3.1.7 Namespaces

As we explained in Section 2.1.2, GNU Smalltalk supports multiple namespaces. Using the new syntax, the way to switch the current namespace is:

```
Namespace current: MyNamespace [
]
```

Every statement or declaration between the angle brackets is referred to namespace *MyNamespace*.

3.1.8 Summary

Putting it all together, a complete class using the new syntax looks like this:

```
SuperClassName subclass: SubClassName [
    | instanceVariables |

    ClassVariable := initExpression.

    SubClassName class [
        | classInstanceVariables |

        methodPattern [ methodStatements ]
    ]

    methodPattern [ methodStatements ]
]
```

3.2 BNF Grammar

Section 3.1 showed the details of all the syntactic elements available in GNU Smalltalk. In this section we will discuss the grammar needed to represent those elements. The following grammar is written using a BNF-like notation and represents only the addition made to the original GNU Smalltalk grammar.

```
scoped_definition : class_definition
                  | namespace_definition

class_definition : qualified_name "subclass: " identifier "[" class_body "]"
                 | class_name "extend" "[" class_body "]"

namespace_definition : "Namespace" "current: " qualified_name "[" doits "]"

qualified_name : [identifier"."]* identifier

class_name : qualified_name
           | qualified_name "class"

class_body : [class_body_elem]*

class_body_elem : instance_variables
                | method
                | class_metadata
                | class_variable
                | class_protocol_definition

instance_variables: "|" [variable_name]+ "|"

method : [class_name ">>"]? message_pattern "[" method_body "]"

class_metadata : attributes

class_variable : variable_name ":=" expr

class_protocol_definition : class_name "[" class_body "]"

message_pattern : unary_pattern | binary_pattern | keyword_pattern

unary_pattern : identifier

binary_pattern : binop identifier

keyword_pattern : [keyword identifier]+

method_body : local_variables attributes statements

local_variables : "|" [variable_name]+ "|"
```

3.2.1 Analysis and disambiguation

The grammar we defined in Section 3.2 is in the class of *LL(k)* grammars. That is, a grammar which is parsable in a top-down and left to right manner. Since the pre-existent parser of GNU Smalltalk is a hand-written recursive descent parser, we are interested in such a grammar because we can simply extend it to fit our needs.

However the grammar suffers from a common problem, namely it is ambiguous. In particular the presented grammar is LL(3); this means that when using a recursive descent parser, three lookahead tokens are needed in order to correctly recognize the language. The following example clarifies the problem.

Assume that the following snippet of code has to be parsed:

```
Object subclass: Test [
    | anInstanceVariable |
]
```

Suppose that the first line, the class declaration, has been already parsed. According to our grammar we are now in the *class_body* rule:

```
class_body : [class_body_elem]*

class_body_elem : instance_variables
                | method
                ...
```

The first token that has to be parsed is the character |. At this point, we don't know the production we have to follow. As far as a parser could know there are two productions that can be followed. For instance, it could follow the *instance_variables* rule:

```
instance_variables: "|" [variable_name]+ "|"
```

or alternatively it could reach the *binary_pattern* rule by going through the *method* and the *message_pattern* rules:

```
method : message_pattern "[" method_body "]"
       | class_name ">>" message_pattern "[" method_body "]"

message_pattern : unary_pattern
                | binary_pattern
                | keyword_pattern

binary_pattern : binop identifier
```

It is clear that one single token is not sufficient to recognize the fact that we are parsing an instance variable. Similarly, we can apply the same reasoning made before, and discover that even if we fetch the second token, that is the identifier *"anInstanceVariable"*, we are faced with the same problem. In order to choose the right rule, the parser would have to get the third token to disambiguate this case.

3.2.2 Parsing Table

In Section 3.2.1, we just showed that our grammar is *LL(3)* and also one case in which three lookahead tokens are needed. However, by applying the same reasoning, it is possible to find out that there are other cases in which the grammar needs more than one token to recognize which production to follow. Table 3.1 can be used as a reference for all those cases in which the grammar is ambiguous.

token 1	token 2	token 3	possible productions
|			instance_variables or method
|	|		instance_variables
|	identifier		instance_variables or method
|	identifier	identifier	instance_variables
|	identifier	[	method
identifier			method or class_variable or class_protocol
identifier	identifier		method or class_variable or class_protocol
identifier	identifier	binop	method
identifier	identifier	[	class_protocol
identifier	assignment		class_variables
identifier	binop		method
identifier	[		method
identifier	.		method
binop			method
>			method
keyword			method
<			method or class_metadata
<	keyword		class_metadata
<	identifier		method

Table 3.1: Parsing table

3.3 Testing

While developing the parser, part of the time was spent on writing regression tests. Producing a complete coverage of all possible cases would be a waste of time, because they are too many. When writing our regression tests, we put emphasis on valid cases, that is, cases in which the parser should recognize the given input.
Taking into account that GNU Smalltalk has a good test coverage of the class library which is about forty thousand lines of code. Once the class library is converted to the new syntax, if everything still works and all of those test cases pass, it means that the parser does what it should do.

3.4 Syntax Converter

As we said, the second goal of the project was to provide a syntax converter. That is, a program that takes as input a source file written in old syntax and produces a corresponding implementation written in the new syntax.
In order to write this converter, we took advantage of a package called STInST that comes with the GNU Smalltalk project. STInST provides, along with other utilities, a parser for Smalltalk written in Smalltalk itself. Figure 3.1 depicts the class diagram of the converter.

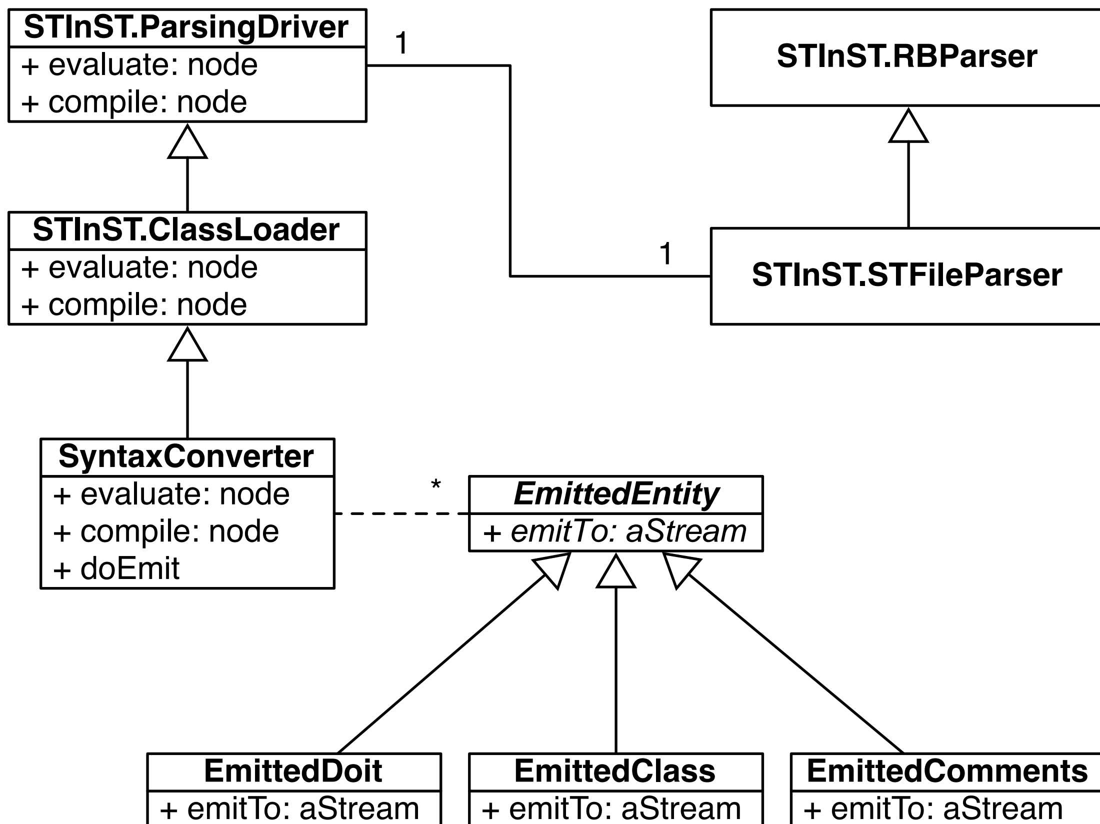

Figure 3.1: UML class diagram of the converter

As we can see from the diagram, the *STParsingDriver* hierarchy cooperates with *STFileParser* through the "strategy" pattern. In turn, *STClassLoader* is a superclass for drivers that need an in-memory model of the classes being parsed. This is a lightweight representation that is not executable, but still mostly polymorphic with the standard smalltalk classes that implement reflection (Class, Metaclass, CompiledMethod, Namespace).

Our *SyntaxConverter* inherits from the class loader in order to grab this representation. In addition, after every chunk the *SyntaxConverter* tries to understand what kind of content will have to be emitted. As chunks are processed, the syntax converter creates *EmittedEntity* objects for each section in the output, and inserts them into an ordered list. At the end of this process, the *SyntaxConverter* iterates over it. On each *EmittedEntity*, *#emitTo:* is called so that the output is formatted on the given stream.

3.5 Example

In this section we provide an additional example of using the new syntax. The original example was taken from [Bec97].

```
Object subclass: Money [
    | amount currency |

    Money class >> amount: aNumber currency: aSymbol [
        ^self new
              setAmount: aNumber
              currency: aSymbol
    ]

    amount [ ^amount ]

    currency [ ^currency ]

    setAmount: aNumber currency: aSymbol [
        amount := aNumber.
        currency := aSymbol
    ]

    + aMoney [ ^aMoney addMoney: self ]

    addMoney: aMoney [
        ^currency = aMoney currency
            ifTrue:
                  [self class
                        amount: amount + aMoney amount
                        currency: currency]
            ifFalse:
                   [MoneySum monies: (Array
                        with: self
                        with: aMoney)]
    ]

    addMoneySum: aMoneySum [
        ^aMoneySum addMoney: self
    ]

    printOn: aStream [
        aStream
            print: amount;
            space;
            nextPutAll: currency
    ]
]
```

```
Object subclass: MoneySum [
    | monies |

    MoneySum class >> monies: aCollection [
        ^self new setMonies: aCollection
    ]

    monies [ ^monies ]

    setMonies: aCollection [
        monies := aCollection
    ]

    + aMoney [
        ^aMoney addMoneySum: self
    ]

    addMoney: aMoney [
        ^self class monies: (monies copyWith: aMoney)
    ]

    addMoneySum: aMoneySum [
        ^MoneySum monies: monies , aMoneySum monies
    ]

    printOn: aStream [
        monies do:
            [:each| aStream
                        print: each;
                        nextPutAll: ' + '].
        aStream skip: -3
    ]
]

Eval [
    | m1 m2 sum |

    m1 := Money
            amount: 5
            currency: #USD.

    m2 := Money
            amount: 7
            currency: #GBP.

    sum := (m1 + m2) + (m1 + m2).
    sum printNl.
]
```

Chapter 4

Discussion

Extending the syntax of Smalltalk is an arguable choice. However, since we do not rely on a development environment, we fulfill part of its functionality with the support of a more complete syntax on the language side. This solution is a compromise. The new syntax does not propose a pure Smalltalk solution, but for the same reason, even the bang-separated chunks format is a compromise.

From a theoretical point of view, both implementations are equivalent in power, but our syntax is meant to be specifically used for scripting. Adopting the new syntax is therefore a way to improve the GNU Smalltalk project. The examples we showed in Section 2.2.3 and 3.5, look promising.
Our specialized syntax provides a concrete way to improve the scripting activity on GNU Smalltalk. In section 2.2.3 we already mentioned the increased usability when writing and reading scripts.

Another consideration is that our syntax is also editor-friendly and easiliy manageable. Constructs such as classes and methods are well scoped. Therefore, features like folding or syntax highlighting can be easily implemented. It clearly depends also on how the editor itself is implemented.

On the other side, in respect to other languages, there might be some missing features. For example, Ruby provides a succinct way to define accessor methods. By using the keywords *attr_reader*, *attr_writer* and *attr_accessor* it possible to define respectively a getter, a setter or both getter and setter for a given attribute of a class. (see the *Person* example written in Ruby in Section 2.2.1). Using our syntax, accessors must be explicitly written by hand. With the drawback that these still occupy a relevant part of a source code file.

Chapter 5

Conclusions

A scripting environment based on Smalltalk represents an attractive alternative to the existing scripting languages. Although the goal of GNU Smalltalk is to produce such a scripting environment, the improvements proposed in this project considerably increased its value. Defining an extended Smalltalk syntax is crucial for our purpose, for at least two reasons:

1. In Smalltalk there is no syntax to define a *class* or a *program*. Because they can be defined interactively inside the development environment and stored inside the image. A scripting environment does not rely on a development environment, so we introduce these notions inside the syntax.

2. The bang-separated chunks format does not represent an alternative which is good enough. Remember that it was introduced as a machine-readable format to enable sharing of source code.

Migrating to a new syntax is not an easy task. Providing tools, such as the automatic converter, is important. The class library of GNU Smalltalk has been easily ported to the new syntax. By using the same converter, users of GNU Smalltalk, can easily port their own code base.

5.1 Contributions

We provide the following contributions to the GNU Smalltalk project:

- the definition of a new syntax for GNU Smalltalk scripts.
- an extended parser that recognizes the new syntax.
- a tool that converts source code from old to new syntax.

These contributions were already accepted and committed to the official GNU Smalltalk repository[1].

[1] http://savannah.gnu.org/projects/smalltalk

5.2 Future Work

GNU Smalltalk comes with a number of tools. Some of them are based on the STInST package discussed in Section 3.4. One of those tools is a documentation tool. That is, a tool that loads source code and automatically creates documentation for it.
In order to support the new syntax, the documentation tool should use a new kind of *STFileParser*. That is, a parser that recognizes the new syntax.
Having such a parser would be useful also when implementing new tools that need to analyze source code. For example, it could be used in combination with the syntax converter to implement a pretty printer.

List of Figures

2.1 Squeak's System Browser 4
2.2 Smalltalk virtual machine and image 5

3.1 UML class diagram of the converter 15

List of Tables

3.1 Parsing table 14

Bibliography

[Bec97] Kent Beck. *Smalltalk: best practice patterns.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[Kra83] Glenn Krasner, editor. *Smalltalk-80: bits of history, words of advice.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.