Implementing a VSOP Compiler

Cyril SOLDANI, Pr. Pierre GEURTS

March 2, 2016

Part II

Syntax Analysis

1 Introduction

In this assignment, you will use the parser generator of your choice to produce a parser for the VSOP language, according to the syntactic rules described in the VSOP manual. You will reuse the lexer developed for the previous assignment (you can of course improve it if needed).

To allow easy testing of your generated parser, your program will dump the *abstract syntax tree* (AST) corresponding to given input VSOP source file on standard output, in the format described in section 2.

As VSOP source code files can contain syntax errors, you will need to detect those and print error messages when they occur. The way syntax errors should be handled is not precisely described in the VSOP manual. This documents gives some more information about the way your compiler should handle those errors in section 3.

Some guidance regarding how you should test your parser is given in section 4.

Finally, the way you should submit your work for evaluation is described in section 5.

This assignment is due at the latest for Wednesday the **16th of March** 2016 (23:59 CET).

2 Output Format

You are free to use the structure of your choice to represent your AST, but your parser output should be structured as follows.

Preliminaries

You can use spaces and line feeds for legibility in your output, if you want.

To improve output legibility, you will not print AST node positions, but it is a good idea to keep them in your AST data structure, for semantic error messages and debugging.

Lists will be enclosed in square brackets (`[]`), with items separated by commas (`,`).

The following of this section describes how the different elements of the AST should be printed, referring to them in the same way as in the grammar in the VSOP manual.

It concludes with an example output.

Program

```
program = class { class };
```

Print the list of classes, in the same order as in the input file.

Class

```
class = "class" type-identifier [ "extends" type-identifier ] class-body;
class-body = "{" { field | method } "}";
```

Each *class* will be printed as

```
Class(<name>, <parent>, <fields>, <methods>)
```

where `<name>` is the class name, `<parent>` its parent class, or `Object` if none was provided[1], `<fields>` is the list of class fields, and `<methods>` the list of class methods. Both fields and methods will be listed in the same order as in the input file (but separately).

Field

```
field = object-identifier ":" type [ "<-" expr ] ";";
```

Each *field* will be printed as one of

```
Field(<name>, <type>)
Field(<name>, <type>, <init-expr>)
```

depending on whether or not a field initializer was provided.

Method

```
method = object-identifier "(" formals ")" ":" type-identifier block;
```

Each *method* will be printed as

```
Method(<name>, <formals>, <ret-type>, <block>)
```

where `<formals>` is the list of formal parameters, `<ret-type>` the method's return type, and `<block>` the method body.

Types

```
type = type-identifier | int32 | bool | string;
```

Types should be printed literally, as they would appear in VSOP source code.

[1]`Object` is the ancestor of all classes in VSOP.

Formals

```
formals = [ formal { "," formal } ];
formal = object-identifier ":" type;
```

Each *formal* argument will be printed as

```
  <name> : <type>
```

Blocks

```
block = "{" expr { ";" expr } "}";
```

A block will be printed as

```
  Block(<expr-list>)
```

if it contains multiple expressions, and simply as

```
  <expr>
```

if it contains a single expression.

Expressions

```
expr = "if" expr "then" expr [ "else" expr ]
```

```
  If(<cond-expr>, <then-expr>)
  If(<cond-expr>, <then-expr>, <else-expr>)
```

```
expr = "while" expr "do" expr
```

```
  While(<cond-expr>, <body-expr>)
```

```
expr = "let" object-identifier ":" type [ "<-" expr ] "in" expr
```

```
  Let(<name>, <type>, <scope-expr>)
  Let(<name>, <type>, <init-expr>, <scope-expr>)
```

```
expr = object-identifiers "<-" expr
```

```
  Assign(<name>, <expr>)
```

```
expr = "not" expr
     | "-" expr
     | "isnull" expr
```

```
  UnOp(not, <expr>)
  UnOp(-, <expr>)
  UnOp(isnull, <expr>)
```

```
expr = expr ("=" | "<" | "<=") expr
     | expr ("+" | "-") expr
     | expr ("*" | "/") expr
     | expr "^" expr
```

```
  BinOp(<op>, <left-expr>, <right-expr>)
```

where `<op>` is represented literally as it would appear in VSOP source code.

```
expr = object-identifiers "(" args ")"
     | expr "." object-identifiers "(" args ")"
```

```
  Call(<obj-expr>, <method-name>, <expr-list>)
```

A call of the form `someFunc(arg1, arg2)` is just *syntactic sugar* for `self.someFunc(arg1, arg2)`. `self` denotes the current object in VSOP. Print them using the explicit version, *e.g.*

```
  Call(self, someFunc, [arg1, arg2])
```

```
expr = "new" type-identifier
```

```
  New(<type-name>)
```

```
expr = object-identifier
```

Print *variable names* literally, as they would appear in VSOP source code.

```
expr = literal;
literal = integer-literal | string-literal | boolean-literal;
boolean-literal = "true" | "false";
```

String literals should be escaped as in your lexer output, *i.e.* enclosed in double quotes ("), escaping \ and " with \\ and \" (respectively), and non-printable characters as `\xhh` where `hh` is the numerical value of the byte in hexadecimal.

Integer literals should be represented in decimal, and booleans will be denoted by `true` and `false` as in VSOP source code.

```
expr = "(" expr ")"
```

Simply print the expression, as described above. There is no need for the parentheses, which are now implicitly conveyed by the structure of the tree, *e.g.*

```
(a + b) * c
```

will give

```
  BinOp(*, BinOp(+, a, b), c)}
```

correctly conveying the fact that `a` and `b` should be added first, and then only multiplied with `c`.

Example

For the source code of the linked list example in the VSOP manual, your parser should output (except for whitespace differences):

```
  [Class(List, Object, [],
     [Method(isNil, [], bool, true), Method(length, [], int32, 0)]),
   Class(Nil, List, [], []),
   Class(Cons, List, [Field(head, int32), Field(tail, List)],
     [Method(init, [hd : int32, tl : List], Cons,
        Block([Assign(head, hd), Assign(tail, tl), self])),
      Method(head, [], int32, head), Method(isNil, [], bool, false),
      Method(length, [], int32, BinOp(+, 1, Call(head, length, [])))]),
   Class(Main, IO, [],
     [Method(main, [], int32,
        Block(
          [Let(xs, List,
             Call(New(Cons), init,
               [0,
                Call(New(Cons), init, [1, Call(New(Cons), init, [2, New(Nil)])])]),
             Call(
               Call(Call(self, print, ["List has length "]), printInt,
            [Call(xs, length, [])]),
          print, ["\x0a"])),
      0]))])]
```

3 Error Handling

Your parser should detect and report syntax errors in input VSOP source file. Error reporting is part science, and part *black magic*. You are free to implement it as you see fit (or as you can). The only constraint is that error messages should begin with

```
<filename>:<line>:<col>: syntax error
```

Generally the reported position should be the one of the first token that cannot be part of a valid grammar production, but this may be hard to do depending on which parser generator you use. In some cases, it might also be more interesting to report the error elsewhere, if it is more likely to be the error position. *E.g.* in code

```
{ if cond then e1; else e2 }  // Parsed as { (if cond then e1); else e2 }
```

the first token in error is the `else` (which cannot be part of previous *if-then* which is ended by the semicolon), but reporting the error at the semicolon could arguably be more likely to help the developer.

It is generally nice when a compiler can report several errors in the same execution. However, this generally requires to discard some tokens after a syntax error, in order to resynchronize in some sensible place. If not done smartly, it results in a great number of spurious error messages after an error, greatly diminishing the advantage of reporting several errors at once. Finally, this may be more or less difficult to do depending on your parser generator and/or AST structure.

Don't forget that lexical errors can still happen. When a lexical error occurs, you can either stop processing, or try to recover in one way or another to try detect some more potential lexical and/or syntax errors.

Finally, you can do what you want with the standard output in case of errors. You can print a partial AST up to the error, print a full AST with some dummy values in the areas of errors, or print nothing at all (which has the merit of making clear that an error occurred).

Due to the great freedom left in error reporting, the testing script will be quite lax, and only check that your parser exits with non-zero status and prints some error message in case of error. However, we will check your parser manually to see how good your error messages are.

We also ask you to provide a short report, describing the general overview of your error reporting strategy. Are you reporting a single or several errors? How do you detect syntax errors? How do you recover if you do? What position are you reporting? *etc.*

4 Testing Your Parser

You are expected to test your parser by writing some VSOP input files, and calling your parser on them. Use both *valid* input files, which are syntactically correct, and *invalid* ones, that should trigger errors. Try to test all language features, and all specific errors you should detect. Don't forget to try a few nested constructs, like a while loop inside an if-then-else branch, or *vice versa*. Also mix and match some operations to ensure that precedence and associativity rules are respected.

Don't assume your lexer is bug-free. If you encounter some strange parsing errors, or if the output is different from what you expected, it may be worth calling `vsopc` again with the `-lex` argument to check if generated tokens are correct.

Finally, check your parser generator documentation to see if it comes with built-in support for debugging. With most parser generators, the parser can report many information such as read tokens, content of the parsing stack, transitions between states, *etc.* once debugging support has been activated.

5 How to Submit your Work?

Your work should be submitted through the submission platform.

You should submit an archive named `vsopcompiler.tar.xz`. That archive should contain a `vsopcompiler` folder with your code, and a `Makefile` to build it. A skeleton `vsopcompiler` folder and `Makefile` is provided on the assignments web page. You can actually build your compiler with whatever tool you want (if you need building at all), but still provide a `Makefile` with the targets described here.

Your archive should also contain in that folder a short report describing your approach to error reporting, either in `report.txt` or `report.pdf`. It should also give an overview of how you implemented the parser (which generator did you use, how does it interface with your lexer, what were the main difficulties, *etc.*?). Finally, if something in the implementation is not obvious from your **commented** code, also mention it in that report.

Finally, provide your test VSOP input files in a `vsopcompiler/tests` sub-folder. Any file with `.vsop` extension is that folder will be tested against the reference parser by the testing script.

Your code will be built as follows. From inside your `vsopcompiler` folder, `make install-tools` will be run. This should install any software needed to build your compiler (`make` and `llvm-dev`) are already installed. You can use `sudo` if you need administrator privileges. Then, `make vsopc` will be issued to build your compiler, which should, obviously, be named `vsopc`. Appending the letter `c` (for **C**ompiler) to the name of the language is a common convention for compilers.

Your code will be executed as follows. Your built `vsopc` executable will be called with the arguments `-parse <SOURCE-FILE>` where `<SOURCE-FILES>` is the path to the input VSOP source code. Your program should then output the parsed AST on standard output, and eventual error messages on standard error, as described above.

The reference platform is an amd64 machine, powered by Debian Jessie (a GNU\Linux distribution). You can use the virtual machine provided on the assignments web page. Alternatively, you can use `debian/jessie64` with vagrant or `debian:jessie` with docker, or your own Debian-based machine.

We will try to add test scripts to the submission platform that check that your submission is in the right format, and test your parser. If you want to be able to use that feedback, don't wait until the last minute to submit your parser (you can submit as many times as you want until the deadline, only the last submission will be taken into account).