Implementing a VSOP Compiler

Florent DE GEETER, Pascal FONTAINE, Cyril SOLDANI

February 14, 2023

Part I

Lexical Analysis

1 Introduction

In this assignment, you will use the lexical analyzer generator of your choice to produce a scanner for the VSOP language, according to the lexical rules described in the VSOP manual. You can also write your scanner by hand if you wish.

To allow easy testing of your generated lexical analyzer, your program will dump the tokens corresponding to the input VSOP source code on the standard output, in the format described in section 2.

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

Finally, the way you should submit your work for evaluation is described in section 4.

This assignment is due at the latest for the **28th of February**, 23h59 CET.

For those who have chosen C++ to implement their compiler, a small C++ example using the lexer generator *flex* and the parser generator *bison* is provided on eCampus. This example will be useful for the two first parts of this project. You are free to reuse the code of the example for your implementation.

2 Output Format

You will convert the sequence of characters in the source code input file into a stream of tokens. Each generated token will then be printed on its own line on standard output, according to the following format:

```
token-output-line = LINE, ",", COLUMN, ",", TOKEN-CLASS, [ "," TOKEN-VALUE ]
```

where

LINE is the line of the first character of the token in the source code (counting from 1).

COLUMN is the column of the first character of the token in the source code (counting from 1).

TOKEN-CLASS is the kind of token read. You will use the same names as in the VSOP manual, *e.g.* `integer-literal` if the token is an integer literal, `object-identifier` if the token is an object identifier, *etc.* If the token is a keyword, use the keyword itself as name for the token class.

TOKEN-VALUE is the value of the token.

- For integer literals, output the decimal value (whatever the input format was).
- For type and object identifiers, it is the identifier itself.
- For strings, it will be the character string escaped as follows. The string will be enclosed in double-quotes and each non-printable character (in the ASCII sense, *i.e.* with a byte value lower than 32 or greater than 126) will be replaced by a `\xhh` escape sequence, as described in the VSOP manual (don't bother with `\n`, `\r` and the like). Additionally, the double-quote and the backslash must themselves be escaped (as `\x22` and `\x5c`, respectively).
- Other token types have no associated value.

For example, the input source code

```
class MyClass {
    s : string <- "One\n\x1b[33;mTwo and \
                  three\r\n"
    i : int32 <- 0x1b
}
```

should give as output

```
1,1,class
1,7,type-identifier,MyClass
1,15,lbrace
2,5,object-identifier,s
2,7,colon
2,9,string
2,16,assign
2,19,string-literal,"One\x0a\x1b[33;mTwo and three\x0d\x0a"
4,5,object-identifier,i
4,7,colon
4,9,int32
4,15,assign
4,18,integer-literal,27
5,1,rbrace
```

3 Error Handling

All lexical errors will be printed on the standard error stream. They will be prefixed with

`FILE_NAME:LINE:COLUMN: lexical error:`

where `FILE_NAME` is the input file name, as given on the command-line of your program, and `LINE` and `COLUMN` are the line and column of the position of the error as defined below.

It is an error when a character is read that is not a valid VSOP character. Report the error at the position of the faulty character.

It is an error if a multi-line comment is not terminated when end-of-file is reached. Report the error at the position of the opening `(*` tag for that comment.

It is an error if you see an integer literal such as `0xgZ`. Report the error at the start of the literal. Note that integer literals can be directly followed by some tokens, *i.e.* `0x8g9Z` should report that `0x8g9Z` is not a valid integer literal (rather than returning `integer-literal` `0x8` followed by `object-identifier` `g9Z`), but `0x2a+` should return a token for `integer-literal` 42, followed by token `plus`.

It is an error if a string literal contains a raw line feed. Report the error at the position of the faulty character.

It is an error if a string literal is not terminated when end-of-file is reached. Report the error at the position of the opening double-quote.

Is is an error if a string contains an invalid escape sequence. Report the error at the beginning of the escape sequence (*i.e.* the position of the backslash).

4 How to Submit your Work?

You will use the submission platform. You will need to register (if not already done) and subscribe to the course `INFO0085`. You will then be able to create or join a group, and submit your work for evaluation.

You should submit an archive name `vsopcompiler.tar.xz`. That archive should contain a folder named `vsopcompiler` with your code, and a `Makefile` to build it. You can build your compiler with whatever tool you want (if you need building at all), but still provide a `Makefile` with the targets described below.

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 (some libraries and programs are already installed, see below). You can use `sudo` if you need administrator privileges. Then, `make vsopc` will be issued to build your compiler, which should, logically enough, be named `vsopc`.

Your submissions will be evaluated in a reference container based on Debian Bullseye (a GNU\Linux distribution) on an amd64 architecture. To obtain the reference environment, you can use docker with the image `cffs/compilers` (user and password are both `vagrant`). Alternatively, you can use a real Debian (or Debian-like) system. The following packages are pre-installed in the reference environment: bison, build-essential, ca-certificates, clang, curl, flex, libllvm-ocaml-dev, llvm-dev, locales, ocaml, ocamlbuild, openssh-server, python3, python3-ply, sudo, zlib1g-dev.

Your code will be executed as follows. Your built `vsopc` executable will be called with the arguments `-l <SOURCE-FILE>` where `<SOURCE-FILE>` is the path to the input VSOP source code.

Your program should then output the read tokens on standard output, and possible error messages on standard error, as described above.

The submission platform will check that your submission is in the right format, and test your lexer. If you want to be able to use that feedback, don't wait until the last minute to submit your lexer (you can submit as much as 40 times, only the last submission will be taken into account).

Also note that **5% of your final grade** will be determined directly by the automated tests for this assignment. It is thus doubly in your interest to ensure that your code passes all the tests (and early enough).

Plagiarism. Cooperation (even between groups) is allowed, but all cooperation must be clearly referred to in the final report, and we will have **no tolerance at all for plagiarism** (writing together, reusing/sharing code or text). Code available on the web that serves as a source of inspiration must also be properly referred in the report.

Participation. **Each student has to participate** in the project, and do her/his fair share of work. Letting others in the group do all work will be considered as plagiarism.