Compilers Project

Introduction and Lexical Analysis

Florent De Geeter, Cyril Soldani

University of Liège

February 15, 2022

Outline

Outline

Compilers: from theory to practice . . . and vice versa!

This course is largely **project-based**

- You will write a compiler for a simple programming language
- You will get credits based on the assignments
- Exam will be largely about your project

Compilers theory is very practical, and practising writing a compiler will help understanding the theory

Don't sleep at the lectures!

- Contents directly **useful** for the project
- You will be **evaluated also on theory**

The project: a VSOP compiler

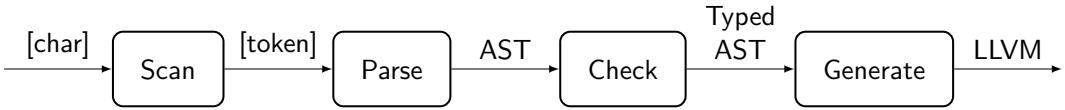
VSOP stands for **Very Simple Object-oriented Programming** language It is:

- simple
- general-purpose
- **object-oriented**
- **expression-based**
- **statically typed**
- explicitly typed
- strict
- ... see manual introduction for details

The language is described in the **VSOP manual** available on eCampus

This manual is subject to changes during the year

Four project phases



- Lexical analysis (Feb 22 2022)
- Syntax analysis (Mar 15 2022)
- Semantic analysis (Apr 19 2022)
- Code generation and **extensions** (May 15 2022)

To be submitted through the submission platform:

- Basic automated tests in place
- Submit early, submit often
- Still **do your own tests**!

You should put **reasonable** effort in the project as a whole, we do not expect your compiler to be *perfect* for every phase!

Last part is optional

This year we have decided to make the **last part optional**, so you will not get a grade of 0 if you don't submit anything.

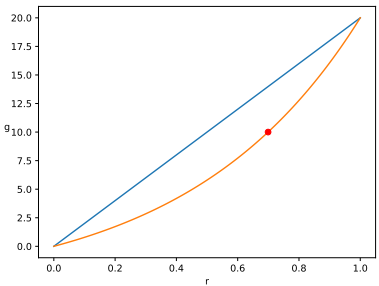
But doing so will **limit the maximal grade** you can get for this course to 13 **out of** 20 **points**.

Automated grading

20% of your grade comes from the automated tests!

Your testing grade g will be derived from your completion rate r (*i.e.* number of passed tests / number of tests) according to the following formula:

$$g = 20\frac{6^r - 1}{5}$$



How to implement your compiler?

You can choose the implementation language among **C/C++**, **Java** and **Python**, but we **encourage** you to choose C++. Send an email if you want to use a language not listed here.

You can use compiler tools:

- You can use **lexer and parser generators**
  e.g. flex, jflex, bison, cup, PLY, ANTLR
- Only for lexing/parsing, **not for AST manipulation**
- You can also use LLVM bindings for code generation

Target environment:

- Your code should be able to run on a **64-bit Debian Bullseye**
- A reference container is provided (`cffs/compilers`)
- The same container is used for automated tests
- See instructions in the assignment briefs

How we build and use your compiler

```
tar xJf vsopcompiler.tar.xz
cd vsopcompiler
make install-tools
make vsopc
cd /path/to/tests
/path/to/vsopc -l a_test_file.vsopc # not -lex
```

A `Makefile` is mandatory (it can call something else)

`make install-tools` should install anything required for building your compiler (you can use `sudo`)

`make vsopc` should build your compiler (if needed)

Your compiler should be callable from anywhere!

How to deploy your code

- C/C++: just build a `vsopc` executable
- Python: add a *#!/usr/bin/env python3* shebang line, and make `vsopc` executable
- Java: make a jar and make `vsopc` a wrapper script

```
#!/bin/sh
set -eu
java -jar "/usr/local/lib/vsopcompiler.jar" "$@"
```

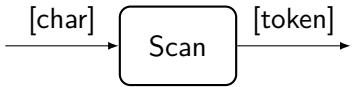
Install your jar in your `make vsopc` rule, or use a relative path:

```
#!/bin/sh
set -eu
DIR="$(dirname "$(readlink -f "$0")")"
java -jar "$DIR/vsopcompiler.jar" "$@"
```

Questions and (possibly) answers

Outline

Lexical analysis



- Converts source input file into a stream of tokens
- Due the 22rd of February
- Automated tests worth 5% of your grade
- Can use a **lexer generator** (*e.g.* flex, jflex, PLY, ANTLR)
- Might need a parser generator (*e.g.* bison, cup) for token definitions
- Pay attention to error management
- You will likely need **starting conditions**, but avoid using too many

Output format

Details in the assignment brief, but basic format is:

```
LINE, ",", COLUMN, ",", TOKEN-CLASS, [ ",", TOKEN-VALUE ]
```

Example:

```
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"
...
```

Printed tokens should go to `stdout`

A word about error management

The brief describes lexical error conditions, and how they should be reported

Error messages should go to `stderr`, and look like

```
input_file.vsop:4:12: lexical error: description
```

Automated tests don't check the description, but we do!

One (or more) lexical error(s) should cause your compiler to fail, with non-zero exit code

Do not write warnings, as `stdout` is used for tokens and any write on `stderr` is assumed to be an error

Interpreting submission platform output

The submission platform tests use `cat -v` to display files. This will print most non-printable characters using escape sequences, so that:

- you can see them
- they are not destroyed by the e-mail transfer

E.g. the carriage return character (`\r`) would be displayed as `^M`

You have to replace those characters when trying to replicate the tests (using a decent editor, or an hexadecimal one)

The `cat -v` format is unlikely to be the same as the one expected by your lexer generator, *e.g.* a CR character is represented as `\r` in flex, not as `^M`

A word about string encoding

String encoding is a surprisingly **complex** matter

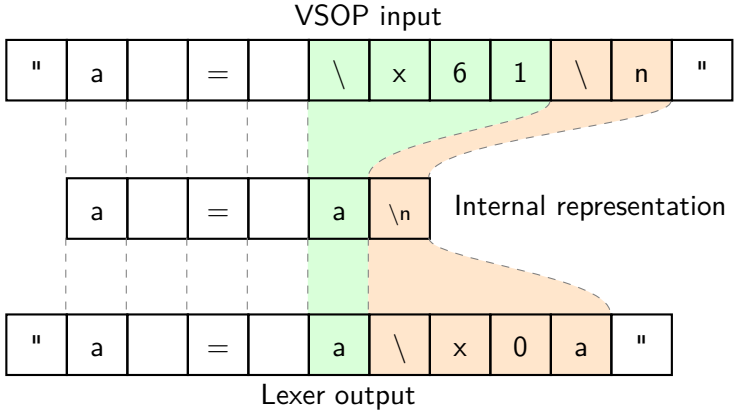
You *can* assume VSOP strings only contain ASCII, except for escaped characters

Token output format is simpler than VSOP:

- Non-printable **bytes** are replaced by `\xhh` where `hh` is their hexadecimal byte value
- Each printable character is itself, except `"` and `\`, which are escaped, *i.e.* `\x22` and `\x5c`

Think about how you will use the strings later on

There are 3 different representations of strings



Forbidden line feed

What is the error to report for the following file?

```
1 "Forbidden raw line feed\"
```

The \" is a valid escape sequence for character "

What is the last character of the file?

- None (end-of-file)? Then it is an unterminated string
- A linefeed? Then the error is the raw linefeed

Most text/code editors will add a final linefeed at the end-of-file

Questions and (possibly) answers

Outline

Using the docker image

A docker container is used to perform the tests on the submission platform. You can also run it on your computer to test locally your code on the same environment than the submission platform's one. First install docker on your computer and then download the image with:

```
> docker pull cffs/compilers
```

Then you can start the container and opens an interactive shell with:

```
> docker run --rm -it cffs/compilers /bin/sh -c "cd /home/vagrant; su vagrant"
```

The container will be automatically stopped and removed when you exit the shell

Copying files to the container

Note that any file created in the container will be deleted when you stop it

- You can use `docker cp` to copy files to/from your container,
- or link one of your directories with one of the container by adding the `-v` option when executing `docker run`:

For instance, `-v "$(pwd):/home/vagrant/compilers"` will link your current directory to the directory `compilers` of the user `vagrant`

Questions and (possibly) answers

Outline

Flex, the lexer generator

Flex is a **lexer generator** written for generating lexical analyzers in C (by default) but also in C++ (by setting the correct options)

Its manual is available on eCampus and some examples are given in the theoritical slides

Here we will see some tricks about **how to use flex with C++**

Wrapping the lexer inside a C++ class

Flex allows to **wrap the lexer inside a C++ class** defined in an other header file
$\Rightarrow$ Any field of this class can be used in the flex lexer

To do that, add `%option c++` and `%option yyclass="MyLexer"` at the beginning of your flex file. The first option tells flex to **generate the scanner in C++**, the second one tells it to **put the lexer in the `MyLexer` class**.

Also, the header file that declares the class must be `include` **in the *Declarations*** part of your flex file.

What about the header file ?

In order for your class to be **compatible with the flex lexer**, there are several things to add in the header file:

- It has to **include `<FlexLexer.h>`**, a file created when flex is installed. However, you could have an error if you just include it. The solution is to put it between some guard:

```
#if !defined(yyFlexLexerOnce)
#include <FlexLexer.h>
#endif
```

- Your class has to **inherit the `yyFlexLexer` class**
- You must **declare the public method `yylex()`**:

```
virtual int yylex();
```

Questions and (possibly) answers