Compiler Project

Introduction and Lexical Analysis

Florent De Geeter, Cyril Soldani & Pascal Fontaine

University of Liège

February 14, 2023

Outline

Outline

Compilers: from Theory to Practice . . . and Vice Versa!

- 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 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**

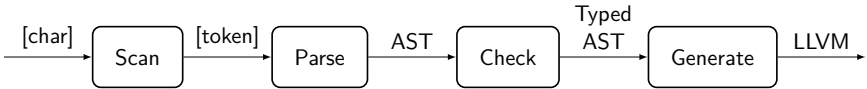
The Project: a VSOP Compiler

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

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

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

Four Project Phases



- Lexical analysis (28/02)
- Syntax analysis (21/03)
- Semantic analysis (18/04)
- Code generation and extensions (16/05)

Each phase has to be submitted on 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 !

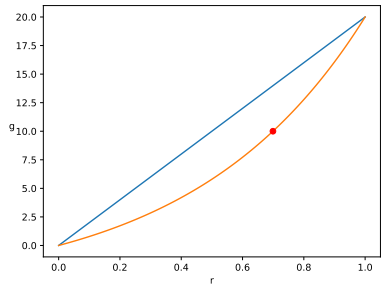
Automated Grading

20% **of your grade** comes from the automated tests ! (5% for each phase)

Your testing grade g will be derived from your completion rate r:

$$r = \frac{\text{number of passed tests}}{\text{number of tests}}$$

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



How to Implement your Compiler ?

Concerning the implement language, you can choose between **C++** or **Python**

We **encourage** you to choose C++, except if you have never coded this language (or you really hate it)

If you want to use a different language from these two, send us a mail

You can use compiler tools (and we encourage you to use them):

- You can use **lexer and parser generators**:
  - e.g. **flex/bison**, PLY, ANTLR
  - Only for lexing/parsing, **not for AST manipulation**
  - a small flex/bison example is available on eCampus
- You can use **LLVM bindings** for code generation

Target Environment

Your code should be able to run on a **64-bit Debian Bullseye**

A **reference Docker container** is provided (`cffs/compilers`), and a small tutorial about Docker will be given at the end of this presentation

The same container is used on the submission platform for the automated tests

See instructions in the assignments 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
```

- A `Makefile` is mandatory
- `make install-tools` should install anything required for building your compiler (you can use `sudo`)
- `make vsopc` should build your compiler (i.e. creating an executable named `vsopc`)
- Your compiler should be callable from **anywhere** !

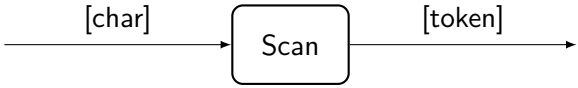
How to Deploy your Code

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

Questions and (Possibly) Answers

Outline

Lexical Analysis



Converts source input file (stream of characters) into a **stream of tokens**

Detects **lexically invalid** source code

`vsopc -l example.vsop` should:

- If the source file is lexically valid, return 0 and print on `stdout` the list of tokens with their position and their type
- Otherwise, return a non-zero value and print on `stderr` (at least) the first lexical error

Assignment

Due the 28th of February

Automated tests worth 5% of your grade

You can use a **lexer generator** (e.g. flex, PLY, ANTLR)

Pay attention to **error management**

You will likely need **starting conditions**, but avoid using too many

Output Format

```
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
```

Error Management

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

Error messages should be printed on **`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 to `stderr` is assumed to be an error

Interpreting Submission Platform Output

The submission platorm 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

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

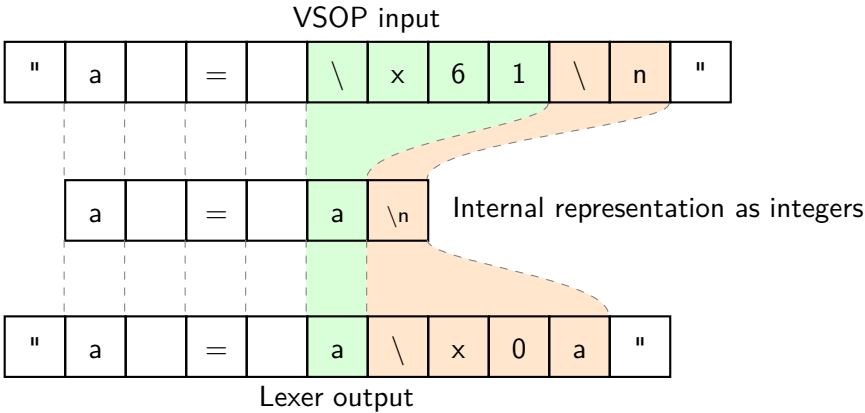
String encoding is a **complex** matter

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

You should print the characters following this format:

- **Non-printable** characters are replaced by `\xhh` where `hh` is their hexadecimal value
  - e.g., the *newline* character (`\n`) should be printed `\x0a`
- **Printable character** are simply printed, except `"` and `\` that are escaped, i.e. `\x22` and `\x5c`

3 Different Representations of Strings



Example of Incorrect VSOP String: Forbidden Line Feed

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

The \" is a valid escape sequence for character ", therefore the string is not terminated by this character

Two possibilities:

- Either the string contains a linefeed → the error is a **raw linefeed**
- Or the end of the file is reached → the error is an **unterminated string**

Questions and (Possibly) Answers

Outline

What is Docker

"*Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers*"

from Wikipedia

Here: Docker is used as a *lighweight VM* manager, to replicate the environment of the submission platform on your computer

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**:

```
docker pull cffs/compilers
```

Opening a shell in a container

You can then **start a container** from this image and open an **interactive shell**:

```
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 (thanks to the `--rm` flag)

You may have a warning telling you that the requested image's platform does not match the detected host platform

To silence it, add the argument:

```
--platform linux/amd64
```

Copying files to the container

The container is isolated from your file system, so you **cannot access your files** by default

To access your files, there are two possibilities:

- Use `docker cp` to **copy files** to/from your file system
- **Link** one of your directories to one of the container by adding the `-v` argument in the `docker run` command
  - e.g. `-v "$(pwd):/home/vagrant/compilers"` will link your current directory to the directory `compilers` of the user `vagrant`
  - Note that any modification made by the container in this directory will therefore **impact your directory**

Any file created in a directory of the container that is not linked to one of your directories will be **deleted** once the container is stopped

Final command

```
docker run --rm -it \
  --platform linux/amd64 \
  -v "$(pwd):/home/vagrant/compilers" \
  cffs/compilers \
  /bin/sh -c "cd /home/vagrant/compilers; su vagrant"
```

Use this command to:

- Start a container from the `cffs/compilers` image
- Open an interactive shell
- Tell Docker to stop and remove the container once you exit the shell
- Use the `linux/amd64` platform
- Link your current directory to the directory `compilers` of the user `vagrant`
- Go to the directory `compilers` of the user `vagrant`
- Log in as `vagrant`

Questions and (possibly) answers