Implementing a VSOP Compiler

Florent DE GEETER, Pascal FONTAINE

February 7, 2022

1 Introduction

In this project, you will implement a compiler for the VSOP language. VSOP stands for ***Very Simple Object-oriented Programming*** language. It is **object-oriented**, **expression-based** and **statically typed**. All the required information about this language in order to implement your compiler is written in its **manual**, available on eCampus.

More precisely, you will implement the front-end part of a compiler, whose goal is to check the correctness of a VSOP source file and to generate the intermediate representation of this input program. As seen during the theoritical courses, the front-end part of a compiler is composed of four steps:

1. the **Lexical analysis** part, where the source file (a stream of **characters**) is splitted into a stream of **tokens**;
2. the **Syntax analysis** part, where this stream is turned into a **syntax tree**;
3. the **Semantic analysis** part, which **annotates** the syntax tree;
4. the **Code generation** part, which generates the **intermediate representation** from the annotated syntax tree.

Once the intermediate representation is obtained, you will use an external compiler to generate the final executable.

This project can be done alone, by group of two (highly preferred option), or exceptionally by group of three (ask for permission, stating the reason). We **strongly recommend** that you do it in groups, because this is quite a large project.

2 Implementation

Concerning the language to implement the compiler, you can choose among **C/C++**, **Java** or **Python**. We **encourage** you to use C++, as it provides high-level structures like classes, while still giving you a low-level control about what it does (unlike Python). It is also compatible with well-known lexer and parser generators that you can use in the two first parts. However, for those who do not know this language (or those who really hate it), you can choose among the proposed alternatives. Finally, if you want to use your favorite language not listed here, contact the assistant and we will discuss about that.

Note also that there are some tutorial sessions about the compiler tools for C++, but we will not be able to provide in-depth support for all tools in the various programming languages.

Part	Deadline
Lexical analysis	22/2
Syntax analysis	15/3
Semantic analysis	19/4
Code generation	15/5

Table 1: Deadlines for the different parts of the project.

3 Submission

You will have **four deadlines** for this project, each one concerning one part of the compiler. The different deadlines are given in Table 1.

Unlike the past years, we have decided to make the last part, the Code generation one, **optional**. This means that you will not get a grade of 0 if you do not submit something for the last deadline. However, doing so will **limit the maximal grade** you can get for this course to 13 points out of 20.

For each deadline you will receive a specific statement, which will tell you what you are supposed to do for that part. Globally, each part will have its own project on the submission platform, and once your code is submitted, it will be tested with different input files (both correct and incorrect). A small part of your final grade will be determined by these automated tests.

4 Docker container

All the tests will be run on a Docker container, to which you have access. This allows you to test locally your code on the same environment than the one of the submission platform.

To use it, first install docker on your computer. Once this is done, you can obtain the docker image with:

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

Note: on Linux, to execute docker commands with your user, you might have to add your user to the docker group

```
> usermod -aG docker
```

and you might have to logout and in again.

Once downloaded, a container can be started from the image with:

```
> docker run --rm -it cffs/compilers /bin/sh
```

This basically starts the container and opens a shell (`-it` option and `/bin/sh` argument) in the container. The option `-rm` tells Docker to remove the container when it will be stopped. You are connected by default as `root`. It is usually not a good idea to stay `root` in a terminal, thus you can switch to the other user, `vagrant`, whose password is also `vagrant`:

```
> su vagrant
```

You can then reach its home directory (`/home/vagrant`) with:

```
> cd /home/vagrant
```

When you are done with the container, you can exit the shell or close your terminal, this will stop the container and remove it (thanks to the `-rm` option).

The container has its own filesystem, which means that it does not have access to your files. You can use `docker cp` to copy files to the container, but a better solution would be to link a directory of yours to a directory of the container. This can be done when starting the container:

```
> docker run --rm -it -v "$(pwd):/home/vagrant/compilers" cffs/compilers /bin/sh
```

This links your current working directory (obtained with the `pwd` command) to the directory of the container named `/home/vagrant/compilers`. Note that any modification made by the container in this directory will impact your directory.