INFO-0085: Compilers

Implementing a VSOP compiler

Maxime Goffart
180521

Olivier Joris
182113

Academic year 2021 - 2022

1 Introduction

In this report, we provide a broad overview of our compiler implementation along with more detailed explanations on some parts of the implementation.

2 Tools

For some parts of the project, we have used external tools.

For the lexical analysis part, we used Flex along with Bison. Specifically, Bison was used only to define the tokens used in Flex.
For the syntax analysis part, we continued to use Bison since we already used it for the lexical analysis part.
For the code generation part, we used the LLVM library. We have followed this tutorial to implement this last part. This choice has been motivated by the fact that the library can optimize the generated code through its optimization passes.

3 Organization of the code

Our code is split in multiple files. The files are the following:

- `Makefile` is a Makefile to build the compiler by simply using the command `make`.
- `vsopc.l` is the file for the lexical analysis part using Flex.
- `vsopc.y` is the file for the syntax analysis part using Bison. It also contains the `main` function of the project.
- `*.cpp` and `*.hpp` are the implementation and header files for the possible tokens.
- `runtime` is a folder containing the files necessary for the code generation part. It contains code for the external linkage with the object class and the power operation in respective `.c` and `.h` files.

The classes defined in the `*.cpp` and `*.hpp` files are organized in an inheritance-based approach using dynamic dispatch because, in our opinion, it is the most natural approach while using an object-oriented language as C++. The hierarchy of classes can be represented by the figure 1.

4 Modules' responsibilities

Most `*.hpp` files contain the headers associated to at least one sub-type of expression. For instance, the file `while.hpp` contains the header associated to the `while` construct.
For each `.hpp` file, there is an associated `.cpp` file with the same name containing the implementation of the header.

The `LLVM.cpp` file is a singleton responsible to make the link between our tokens and the LLVM library.

As already explained, `vsopc.l` contains the rules for the lexical analysis with Flex and `vsopc.y` contains the rules for the syntax analysis with Bison along with the `main` function.

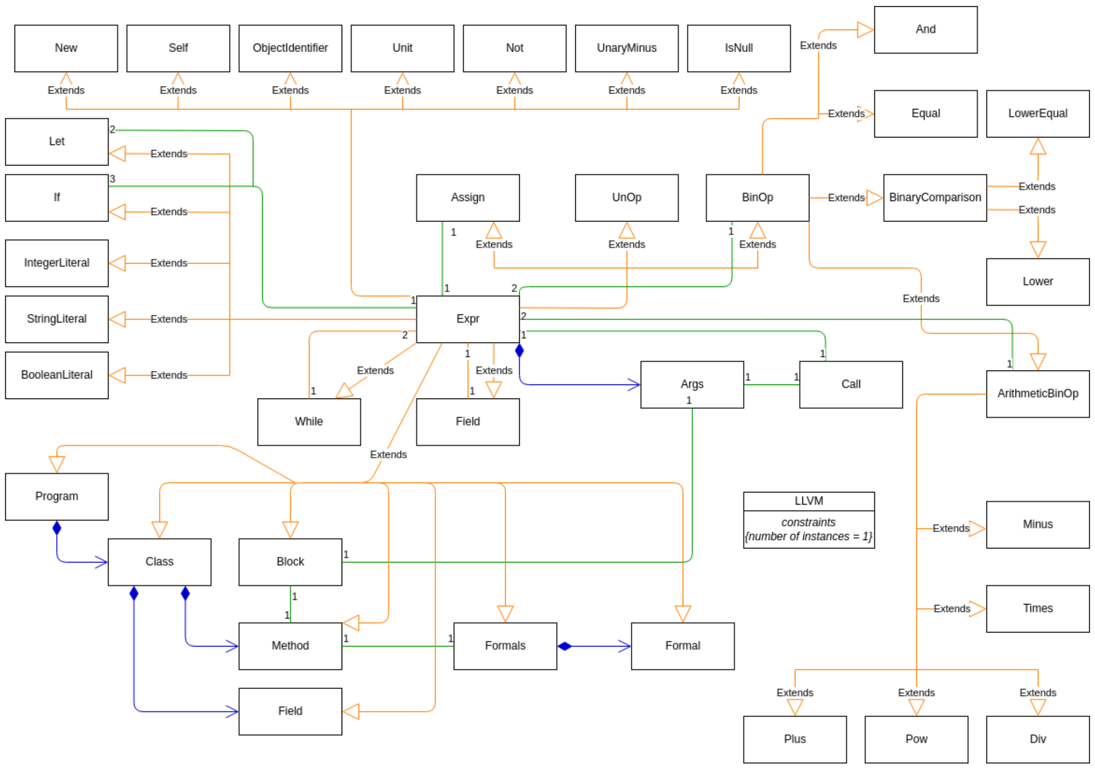

Figure 1: Class hierarchy as a UML diagram

5 Data structures and algorithms

Regarding data structures, we mostly use `std::vectors` defined in C++ standard template library (STL).

For algorithms, we do not make use of algorithms that require any additional explanations than what is provided as documentation in the various source files.

6 Implementation choices

We decided to use Flex and Bison due to the strong recommendations made during the first presentation of the project.
Since we chose Flex and Bison, we were limited to C or C++ as programming languages. We decided to use C++ because we thought that having classes would come handy and, in our experience, it is easier to work with strings in C++ compared to C, even thought we have more experience in C than in C++. Finally, as already mentioned in section 3, we decided to use an inheritance-based approach because, in our opinion, it is the most natural approach while using an OO language such as C++.

7 Conflicts in the grammar

Our grammar does not contain any shift/reduce or reduce/reduce conflicts.

8 Language extensions

Unfortunately, we did not have the time to implement any extension of the language.

9 Limitations, possible improvements, and retrospective

Our compiler support all the basic operations of the VSOP language and pass all tests of the submission platform. If we had more time, we would have implemented more features such as: a garbage collector, the support of decimals, some syntaxic sugar for instructions, etc.

We did not regret the choice of C++ and the tools proposed by the professor and the assistant which saved us time. We just find it a pity that the LLVM library is poorly documented. Fortunately, the tutorial mentioned in the section 2 as well as the example code provided by the assistant have almost filled this gap. We also did not regret to use the inheritance-based approach using dynamic dispatch to implement our abstract syntax tree: we find this approach very easily readable and understandable.

10 Time spent

Unfortunately, as we did not know that we would be asked the time spent at the end of the project, we did not keep track of it nor we can provide an approximation since we are always going back and forth between multiple projects specially with the integrated project.