1 Usage

The usage of the compiler is quite simple, to compile it go to the root of the project and execute `make -C src/`, then you just need to use the given script `./jlc <sourcefile>`.

2 Language specification

The language is specified using `BNFC`, just open the `Javalette.cf` file in order to check all the grammar rules of the language.

3 Shift/reduce conflicts

- **The dangling else problem**: Caused by the rules `Cond` and `CondElse`. With these we define that an `if` statement can optionally have an `else` branch. The problem is that the compiler does not know how to deal with constructions such as `if cond then if cond2 then st1 else st2` since it is impossible to know to which `if` belongs the `else`.

4 About the compiler

The compiler consists of the following pipeline:

1. **Lexer**: Generated with `BNFC`, this phase reads the source file and translate it into a set of tokens.
2. **Desugarer**: Removes syntax-sugar such as for loops and structure/classes definitions.
3. **Typechecker**: Checks the program is correct, and returns the typed program.
4. **Code generator**: Using the typed program, generated the `LLVM` code necessary to run the program.

Each of these phases are implemented in a `Haskell` module with the same name (except the lexer which is implemented by `BNFC`). In addition to those files, we have implemented another module called `LLVM.hs` which defines a small framework to build LLVM programs, this adds an extra layer of type checking (it tries to prevent some mistakes while generating the code).

5 Implemented extensions

We have implemented 5 extensions, which will be explained within the following sections.

5.1 Arrays

This was the first extension implemented, which was modified during the development of the second one. The initial idea was to create a global structure (so the array can be passed by reference) which holded the address and size of the array (the type is not needed during runtime).

In order to hold the type of the array, we've created an internal type `DimT` which contains the type of the elements (`int`, `double`, etc) and the number of dimensions.

5.2 Multidimmensional arrays

We generalized the last extension extending the global structure by adding the number of dimensions and the length of each dimension. In this way, we don't need to allocate multimensional arrays using loops since we just use a 1-dimensional array which contains every dimension. Using a simple formula, we can address the correct element.

The main challenge here was the code generation, since although the structure is similar to the last extension, the assignment is a little bit tedious due to the adressing (and duplication/modification) of the fields of the structure.

5.3 Structures

This was an interesing extension since it provided us with a good framework in order to implement the following ones. We had to add some rules to the grammar, and some types to the program since we need to keep track of the fields of each structure. Moreover, we need to keep track of the synonyms (defined through `typedef`). Once the program passes the type checker, then the code generation is quite straight-forward as LLVM provides structures without any workaround.

5.4 Classes

For this extension, the desugaring phase did mostly all the job. The idea here was to translate objects to structures (containing the attributes) and functions. By doing this, we only needed to change the grammar and the desugarer, while the type checker and the code generator barely changed.

5.5 Classes 2 (with dynamic dispatch)

In order to get this extension working, we needed to add class descriptors (structures which hold the information of a class). Due to this addition, the apprach used for the previous extension was no longer valid.

This time, we splitted structures and classes. Structures remained the same, but now classes are processed in a more complicated way. There is a class descriptor for each class, in order to keep the hierarchy, each of these descriptors point to its parent class (or `null`, in case).

Knowing this, is quite easy to decide which method to execute. Using a function we defined in `C` we check (in runtime) which is the closest method to the object. This means, where in the hierarchy (from the very first parent to the type of the current object) were defined the method, trying to pick the deepest one.