Compiler Documentation

Milla Kortelainen

March 2022

1 Program overview

In this section I will discuss how my implementation is structured. During the project I had some personal scheduling issues and due never using C earlier it gave me some times struggles to implement in that language. This lead compiler to be what I consider maybe a skeleton as it could be a fully functional compiler for the Mini-PL language.

1.1 Structure of the implementation

All the executable code is in the file src/Program.cs. File is divided in four compiler specific classes `Scanner`, `Parser`, `SematicAnalyzer` and `Compile`. There are also three classes for the tree date structure called `Node`, `Edge` and `Tree`. The main class that is executed when the program start is called `MainClass`.

Compiler's structure follows the one taught in class as it is divided in different phases scanning, parsing, sematic analyzis and finally executing the complete version in the compilation phase. Scanner and parser are the most final features in this project where sematic analyzing and compilation are missing some core features. Parser and scanning are missing a error handling so they don't detect users input errors very well. This can be seen as a major missing features as well but at least the tokens and the parse tree is working fine if the user's input is correct code defined by Mini-PL language.

1.2 Executing the program

Run the program by first navigating to the folder Compiler/src with the command line and then run command `dotnet run` which runs the program's main class when in the folder. Input for the compiler is given as a string of text defined in the `MainClass`'s method `Main`. There are multiple strings called `str` defined beforehand. You can switch between these strings by commenting out all but one `str`, the one not commented out will be used as a input for the compiler. You can also make your own input string in the `str` variable in similar manner.

1.3 Missing features

List of features that are missing or not working properly

1. No error handling in any phase. Correct code is detected but there is no way to detect input code's sematic errors.
2. `SematicAnalyzer` doesn't analyze the semantics of the code, it only creates and updates the symbol table. The returned AST is actually just the parse tree.
3. Compilation only handles some math expressions. Not really anything is fully compiled.
4. This documentation also lacks definitions of the code as it goes through the compilation (Regex, LL(1), AST).

2 Work hours

4.3. 1h started implementing scanner

5.3. 4h scanner implementation continued

6.3. 4h scanner implementation ready

7.3. 5h started parser

11.3. 6h finished parser

12.3. 8h starting the compiler and syntactical analysing

13.3. 12h compiler and syntactical analysing

14.4. 2h documentation
Total work hours: 42

3 After math

I greatly underestimated the hours I needed to finish this project. Also in the final steps I lost my changes to the `Compiler` class which in it's most final form could compile even `print`, `vars` with stings and `read` statements. When moving the files to new folder I managed to remove the files and when restoring the files I got only the older version back.

When implementing the program I struggled with the Tree structure I had implemented and over all some syntactical concepts in the compiling gave some hard times. These leaded to delays in the actual implementation. I also would like to give some more time to code's overall structure (for example dividing the classes in different files) but because this was my first program in C I didn't dare to start refactoring the code in such tight schedule.