Compiler Construction Project - Assignment 2: Conditional Statements

September 29, 2023

1 Introduction

In this assignment, you are going to make your first extension to your compiler!
In particular, you will be adding `if-then(-else)-statements` and solve the `dangling else` problem. The rest of this document describes the details of this assignment.

The assignment is described in Section 2. Section 5 lists what should be handed in to complete this assignment. Also, make sure to update your final report with the new changes.
In order to run the framework for assignment2 you have to run the following command in the build directory:
`meson -Dwith-assignment=2 --reconfigure`

2 Assignment

In this Section we cover what we expect from you in this assignment. You will extend the compiler you build in the first assignment with a basic program body and `if-then(-else)-statements`. An important aspect of this assignment is solving the `dangling else` problem.

Example 1 shows a program which your compiler should correctly parse and execute. Note that in this assignment, we expect this program to print '10'.

In the rest of this Section, we describe all steps you need to take to perform this task.

2.1 Grammar

The first phase of your compiler is parsing a file and generating a `SyntaxTree` from it.
In this Section, we only consider the directory `src/grammar/src/main`.

As a first step, you need to extend the grammar and its rules with `flex` and `bison`, in the files `compiler.l` and `compiler.y`.
In `compiler.l` you will need to add the rule for the `MAIN` token, as follows:
`"int main()"  return MAIN;`
Other rules you will need to implement are *statements*, *relational operators*. You should also start thinking

```
    int main() {
        if (3 > 2)
            if (3 != 3)
                print 6 * (3+4);
            else
                print 3 + 7;
    }

```

Listing 1: Example with Dangling Else

how to handle braces and what they represent in code. We leave it up to you to implement the rest of the rules in `compiler.l` and `compiler.y`. You may again implement the functions in the `GrammarVisitor` on-demand.

2.2 Intermediate Code

In the second phase of your compiler, you will need to convert the `SyntaxTree` to a sequence of `Intermediate Code`.
Similar as in assignment 1, you will need to create your own ICGenerator and assign it to the right member variable of the SyntaxTreeVisitor in the file `src/intermediate-code/src/main/cpp/icsyntaxtreevisitor.h`.

2.3 Machine Code

In the final phase of your compiler, you will convert the sequence of `Intermediate Code` to a sequence of `Machine Code Instructions`. In this Section we only consider the directory `src/machine-code/src/main`.

Additionally for this module, you will need to create your own MCGenerator and assign it to the right variable of the IntermediateCodeVisitor in the file `cpp/machinecode/intermediatecodevisitor.h`. Again, please take care to produce valid assembly instructions. For this, you will need to find out which instructions expect which types. While the page is not official, we used `https://www.felixcloutier.com/x86/`.

3 Modifying/Extending the framework

If you wish to add **separate '.cpp' files** you will need to add them to the corresponding **meson file**. For example if we look in the folder **src/intermediate-code/src/main/**, we see a **meson.build** file. In this file we have added a path to all the corresponding '.cpp' files we want to compile. Do this if you need to add extra files at any of the compilation steps.

4 Report

You are responsible for proving that a feature works. This means you should hand in (one or more) example program(s), in separate file(s), that use the feature and compile correctly. If the feature concerns optimization, make sure you can toggle the optimization on and off. Then show that with your optimization on, the output code is smaller or more efficient. Describe this and other additional features in your `README`.

We require you to provide a `README` file including any design choices you have made during this assignment. This also includes a summary of the functionalities of each file you have created or modified. Furthermore, the `README` includes a paragraph on what you have learned from this assignment, what the most challenging parts were and how you dealt with these challenges. It is important that you include a discussion on the *dangling else* problem. Why it is a problem, and why your solution solves it.

5 Submission

This submission is handed in through Brightspace. Go to the course website, and hand in Assignment 2. Beware that the deadline for the Final Report is on the same day.

In Brightspace, hand in

1. a tarball:
   - named `assXgroupY`, with X the assignment number and Y your group number. Name your main folder in the same way (so do not leave it as `assX`).
   - with *all source code* (not only the modified files).

    - **without** the build directory. In general: do not hand in larger submissions than required.

2. the `README` reporting on the assignment.

Failing to adhere to these instructions will result in a penalty to your grade. Please also be aware of the fact that we will not grade work that does not compile. Warnings will result in a penalty in your grade, even if you get warnings when building the framework as is (e.g. unused variable/function warnings).
We use `huisuil` as a reference. So, make sure that your submission compiles on `huisuil`!

You will be graded on the quality of your `README` file, the layout of the code including the modularity and quality of comments, and the functionality of your implementation.

For the date **and time** of the deadline, please check the schedule and submission tab for this assignment on Brightspace.