Compiler Construction Project - Assignment 1: Basics

September 16, 2022

1 Introduction

In this course you will be building your very first compiler, using the theory provided during the lectures. The goal of developing this simple compiler is for you to get familiar with the compilation process, involving a number of different pipeline stages. This assignment in specific will help you get acquainted with the build system and framework used during the practical sessions of this course. The assignments are structured in the following sequence:

1. Basics (this assignment)
2. Conditional statements
3. Types and variables
4. Loop statements
5. Stack management
6. Bonus

Implementing all features up to "Stack management" will allow you to get a grade up to a 9 for the practical component of the course. In the "Bonus" assignment, 2 extra points can be added to this grade.

At the end of all the practical sessions we expect a final report in PDF where you explain each of the pipeline stages separately, encompassing all developments you have made while working on the compiler. We recommend that you work on this final report during the practical sessions, making updates as you walk through the assignments. The deadline for the final report is set to the 16th of December at 23:59. In addition to the final report, we expect a `README` file containing a discussion for each of the separate assignments, see Section 5.

In this document, we cover the basics of the framework in Section 2, to help you out with the assignment given in Section 3. Section 6 lists what should be handed in to complete this assignment, the deadline of handing in the result of this first practical is set to the 30th of September at 8:59.

2 Framework

In this Section, we cover some basics of the framework. In particular we cover its structure and some quick guidelines to get you started.

In this framework we use the build-system `Meson`. For more information please refer to `docs/MESON.md` in your assignment.

2.1 Structure

The framework is structured as follows:

- *docs*: documentations with more information about the concepts used in this framework
- *src*: source files of the compiler.
  Within each subdirectory (grammar, intermediate-code, machine-code) you find two directories: main and test. The main directory contains the source code for the compiler. The test directory contains unit tests which are specific for this subdirectory. In particular:
  - general/src: general code for the whole project. Do not change.
  - grammar/src: the source code for the parsing stage
  - intermediate-code:/src the source code for intermediate code stage
  - machine-code/src: the source code for the machine code stage
  - testutils/src: code for the unit tests.
- *subprojects*: several third-party projects used by this framework
- *test*: test files used to unit-test the compiler
- *meson.build* & *meson_options.txt*: `Meson` build files

2.2 Quick Start

In this Section we provide 'Quick Start' steps, which may be useful if you already are a bit familiar with `Meson`. If you are not, or if you would like more information, we recommend you read `docs/MESON.md`.
Main steps:

1. Install `Meson` and `Ninja` using for example `pip3`. Please verify you have at least Meson version 0.59.1.
2. (optional) add `~/.local/bin` to your `PATH` environment variable
3. Make sure you have `flex` and `bison` installed
4. Instantiate Meson using the following commands in the root directory of the project:
   `mkdir build; cd build; meson ..`
5. Build by calling `ninja` in the build directory

This should have generated all executables and libraries you will need in this assignment.
To run the compiler, execute:
`build/src/machine-code/coco_compiler_machine_code -f [FILE]`, with `FILE` the path to the source file you want to parse.

2.3 Testing

If you followed the steps of the previous subsection, then the executable `src/testutils/coco_compiler_tests` should exist. Simply calling this executable will perform the tests.

It is possible to run only a subset of the tests. For this, you can use the flag
`--gtest_filter=POSTIVE_PATTERNS[-NEGATIVE_PATTERNS]`
, where pattern is of the form `testsuite.testname`, or run –help for more information. To list all tests use
`--gtest_list_tests`

3 Assignment

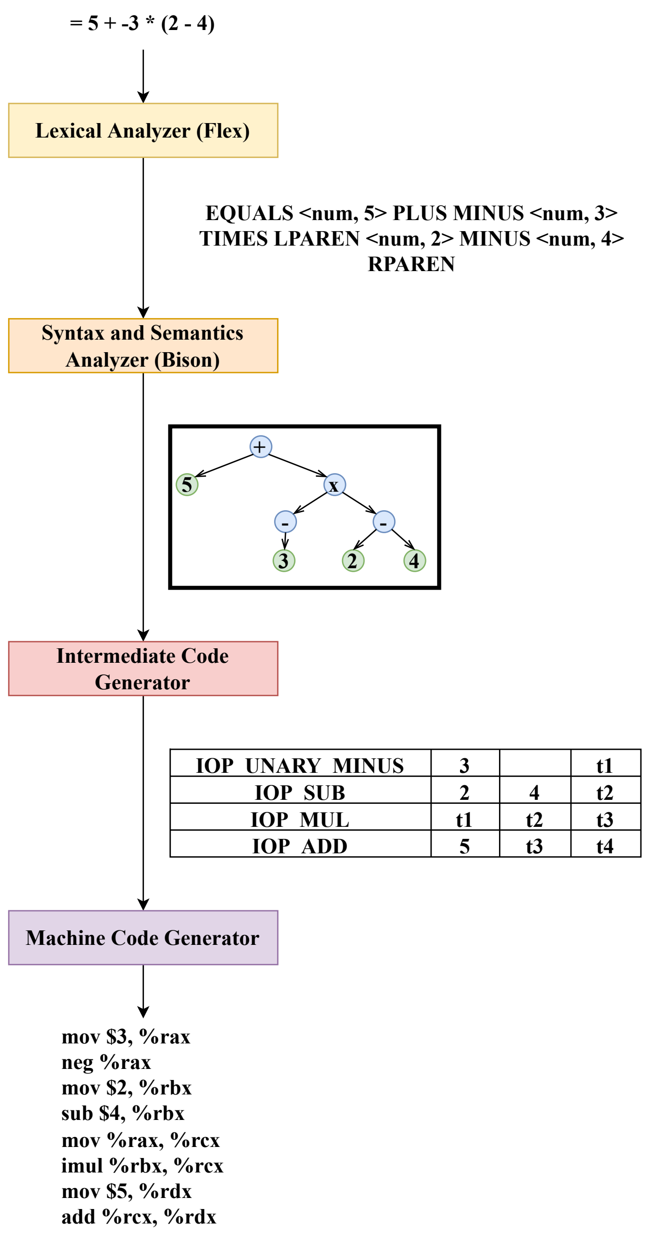
In this Section we cover what we expect from you in this first assignment.

In this first assignment, you will build a simple compiler with a very limited accepted language. You should correctly parse files consisting of a single expression. An example of such an expression is:

```
= 5 + -3 * (2 - 4)
```

Your task is to produce `x86-64` assembly that computes above expression. If required, you may assume the types of the numbers are `uint8_t`.

Figure 1 show an overview of the compiler phases.



EQUALS <num, 5> PLUS MINUS <num, 3> TIMES LPAREN <num, 2> MINUS <num, 4> RPAREN

IOP UNARY MINUS	3		t1
IOP SUB	2	4	t2
IOP MUL	t1	t2	t3
IOP ADD	5	t3	t4

```
mov $3, %rax
neg %rax
mov $2, %rbx
sub $4, %rbx
mov %rax, %rcx
imul %rbx, %rcx
mov $5, %rdx
add %rcx, %rdx
```

Figure 1: Overview Assignment

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

3.1 Grammar

The first phase of your compiler is parsing the file and generating a `SyntaxTree` from it.
In this Section we only consider the directory `src/grammar/src/main`. Note that during compiling the source code you will get a warning about the print statement not being used, this is alright as we will use it from the second assignment.

As a first step, you will need to define your grammar and its rules with `flex` and `bison`. In the directory `flex_bison` you can find the files `compiler.l` and `compiler.y` to help you get started.

3.1.1 Flex

In `compiler.l` you define the language of your grammar and provide rules to produce tokens to be used by `bison`.
We provided you the `newline` and `whitespace` rule. As a next step, you need to define rules to produce tokens. We left two examples, starting with:
`"=" { return ASSIGN; }`
Here we specify that if `flex` sees the string "=", then it should produce the token `ASSIGN`.
For the provided number-rule, we had to do something extra, as we want to pass the value of the number to `bison` as well. So, before we return our token, we pass a duplicate of the string `yytext` to the variable `yylval.number`. We use `strdup` for this.
You are going to add similar rules for the operators and parenthesis. The tokens you should produce are the following: `NUMBER, MINUS, PLUS, TIMES, LPAREN, RPAREN`. You can also find these tokens in `compiler.y`

3.1.2 Bison

In `compiler.y` you define the rules and actions to take for you grammar. For this, you use the tokens generated by `Flex`. The rules look as follows:

$$rule : T\{A\}$$

Here, T is a string of rules and tokens, and A is a sequence of actions. One rule may contain multiple actions.
As an example, we left several rules to parse a number. The entry-point of this first assignment is the expression-rule. Here we first expect an ASSIGN (=), followed by an `additive_expression`. The generated SyntaxTree starts with a NODE_PRINT node which prints the provided expression.
The rules we left are able to parse any program of the form "ASSIGN NUMBER". You will need to create rules to correctly parse other expressions, using the right precedence rules (First unary operators, then parenthesis, then multiplication and finally addition and subtraction).

Next, you are going to define the actions to perform for each rule. Often, you will want to pass information to your 'parent' rule. For this first assignment, you can return three types: `char* (number), Node* (node), NodeType (nt)`. To return one of these types simply set `$<type-name>$`. For example:
`$<nt>$ = NODE_SIGNPLUS;` returns the `NodeType 'NODE_SIGNPLUS'`.
Within a parent-rule you can reference the value of the kth-child by `$<type>k`. For example: suppose `otherrule` returns a `char*`, then you write the following:
`rule : TOKEN otherrule { char* value = $<number>2; }`

For the actions of each rule, you may make use of the `GrammarBuilder`. In particular, this class has several functions which you may use, but you will also need to implement some of them. Please look at `cpp/grammarbuilder/grammarbuilder.h` for a description of these functions. Whether you use them first, or implement them first is completely up to you. To understand the various components of this class, we advice you familiarize yourself with the files in the `public` folder. Note that you may implement the functions on-demand. Meaning that if you do not need a function in this assignment, you may leave it unimplemented.

Finally, we would like to note that `flex` and `bison` could give warnings. You need to fix all these warnings or you will receive a lower grade. If your `bison` version is high enough, you can make use of the `-Wcounterexamples` or `-Wcex` flag. As far as we know, this flag is available from version 3.7 onwards. Please let us know if you find anything different.

3.2 Intermediate Code

In the second phase of your compiler, you will need to convert the `SyntaxTree` to a sequence of `Intermediate Code`.

For this phase, you only need to implement a few functions. However, before you can implement them, you need to understand the idea behind a `Visitor Pattern`. We advice you to first read about them at `https://cpppatterns.com/patterns/visitor.html`.

Your goal in this module is to set the vector of `IStatement`s in the class `IntermediateCode` in the file `src/intermediate-code/src/main/public/intermediatecode.h`. For this, you should create and assign an ICGenerator to the correct assignment-variable in the SyntaxTreeVisitor at `src/intermediate-code/src/main/cpp/intermediatecode/icsyntaxtreevisitor`. Here you make use of the visitor-pattern. We already implemented the `accept` functions in the `Node` classes and created an `AbstractSyntaxTreeVisitor` class in `src/grammar/src/public/syntaxtreevisitor.h`.
Your custom generator should inherit from the abstract ICGenerator in `src/intermediate-code/src/main/cpp/intermediatecode/icgenerators/icgenerator.h`.

An `IStatement` has four members:
`IOperator operand1 operand2 result`

- `IOperator`: specifies the type of the instruction. See: `src/intermediate-code/src/main/public/ioperator.h`.
- `IOperand`: represents the Operands (operand1, operand2, result) in an `IStatement`. There are various types as you can find in `src/intermediate-code/src/main/public/ioperand.h`. Please note that it is your responsibility to make sure that all `SymbolIOperand`s have unique identifiers.

3.2.1 Tips & Tricks – SyntaxTreeVisitor

In this Section we provide some tips and tricks for implementing your visitor.

- You may add generic visit-functions to the abstract ICGenerator, such that generators from later assignments can use them.
- The `visit` functions return `void`, thus you are unable to return anything. However, you will need to use some `IOperand`s twice. Therefore, we recommend to use the `last_temp std::shared_ptr<IOperand>` member variable to keep track of your last generated `IOperand`. You may find this variable in the struct `SyntaxTreeVisitorInfo` in the file `src/intermediate-code/src/main/cpp/intermediatecode/icsyntaxtreevisitor.h`

3.3 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`.

In this module, you need to convert each `IStatement` to an `Instruction`, which consists of several components: `type is_signed mOperator src dst`, which are represented by the following types:

- `OperatorType`: specifies the width (in bytes) of the instruction. In particular these are: Byte (1), Word (2), Double (4), Quad (8). See also `src/intermediate-code/src/main/public/ioperator.h`.
- `bool`: specifies if the instruction has a signed-type or not (unsigned).
- `MOperator` (mOperator): specifies the operator of instruction.
  See also `public/moperator.h`.
- `MOperand` (src, dst): represents an instruction operand. In this first assignment, this can only be an immediate or a register. See also `public/moperand.h`. In particular, a `MOperand` consists of a flag (`MOperand_flag`), and a value (`MOperand_mem`). The value is either a `RegisterName`, an `uint64_t` or an `off_t`, and the flag indicates which of the three types (or neither) the value represents. Please check `public/register.h` for all the `RegisterName`s. You may assume there are always enough registers available (number of registers $\geq$ number of temporary variables).

Similar as for intermediate-code, you will need to create your own MCGenerator and assign it to the right variable in the IntermediateCodeVisitor constructor in the file `cpp/machinecode/intermediatecodevisitor.h`. The custom generator should inherit from the generic MachineCodeGenerator in `cpp/machinecode/mcgenerators`. You may also extend this generic MachineCodeGenerator with generic functions used by multiple generators.

Additionally, you will need to create your own RegisterAllocator as specified by the abstract `cpp/registerallocators/registerallocator.h`, and assign it to the registerAllocator member variable of the IntermediateCodeVisitor.

When implementing the `visit` functions, 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/`.

4 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.

5 Report

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.

6 Submission

This submission is handled through Brightspace. Go to the course website, and hand in assignment 1.

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.