Compiler Construction (Winter 2025)

Project Assignment 1

Programming Languages and Compilers
Prof. Dr. Roland Leißa
Al-Harith Farhad, M.Sc

Project Task A General Information

In the practical project you will implement a compiler for a subset of C. The milestones of the project roughly reflect the structure of a compiler, which also structures the lecture. Although the milestones will not be graded, you will receive feedback that hints at existing shortcomings in your compiler. We strongly suggest that you create your own test suite for the project and not only rely on the feedback from our test runs. Each milestone comes with a soft deadline that indicates the expected time schedule. The final submission will be graded based on the percentage of tests passed and a code review where the group members have to justify their work.

Technical requirements and restrictions:

- The compiler itself must be written in C++20.
- Your project must either be buildable with `make` or

  ```
  mkdir build
  cmake -S . -B build
  cmake --build build
  ```

  and produce a binary `c4` in `build/bin`.
- Upon acceptance of a source program, `c4` must terminate with return code `0`. Rejection of a source program must be indicated by return code `1` and an error message with the following format on `stderr`:

  ```
  <filename>:<line>:<column>: error: <message>
  ```

  Here, `<filename>` is the name of the input file as provided to your compiler, `<line>` and `<column>` are the line number and the column number of the error's location, and `<message>` is some meaningful error message.
- The individual assignments will refine the requirements and specify the subset of C that we consider for this project. If you feel like it, you can implement more than this subset in your compiler. Generally, we require you to accept and correctly compile all programs that are part of the specified subset and to reject all programs that are not a part of the full C language. Programs that are not in those two classes, i.e. that are valid in C but not part of our subset, you may either reject with an error message or implement correctly according to the C language specification.

Project Task B Lexer

Implement the lexical analysis. The relevant section is §6.4. The lexer uses the maximal munch strategy (§6.4p4). The project does not incorporate a preprocessor. Therefore we are only considering *token*s, not *preprocessing-token*s. Furthermore, *universal-character-name*s (§6.4.3) are omitted. For *constant*s (§6.4.4) we process only *decimal-constant*s, 0 and *character-constant*s. In this connection we ignore all *integer-suffix*es and prefixes for the encoding (`L`, ...). *character-constant*s are restricted to single characters. For *string-literal*s (§6.4.5) the *encoding-prefix* is also omitted. Also we elide *octal-escape-sequence*s and *hexadecimal-escape-sequence*s. Finally, we ignore digraphs, `#`, and `##` in *punctuator*s.

The phases of translation are presented in §5.1.1.2. We leave out several parts. In phase 1 trigraphs are elided. For the input and execution encodings we use ISO-8859-1. Phases 2, 4 and 6 are left out. It is not necessary to implement the remaining phases in detail. Some of them may be combined.

To test this part, output the tokens. For this, use the switch `--tokenize`, e.g. `c4 --tokenize test.c`. Output one token per line: First the position in the source (terminated by a colon), a single space, then the kind of token, another single space, and last the textual representation of the token. The position is a tuple of line and column in the source file, both counting from 1. New lines are started by a `CR LF` sequence (`"\r\n"`), or by `LF` (`"\n"`. Make sure to output string- or character-constants exactly like they appear in the input stream, so do not replace escape sequences in the output. For naming the tokens, use the non-terminals on the right hand side of the productions of *token* (§6.4p1). For example, consider the input file `test.c`:

```
42␣␣if
⎵"bla\n"x+
```

Output:

```
test.c:1:1: constant 42
test.c:1:5: keyword if
test.c:2:2: string-literal "bla\n"
test.c:2:9: identifier x
test.c:2:10: punctuator +
```

Files that contain lexical errors must be rejected with an error message showing the location of the error. For error locations, choose the location of the first character that makes the input invalid.[1]

The running time of your lexer must be proportional to the input size.

Remarks and hints:

- Write more test cases, especially some that test the rejection capabilities of your lexer.
- §6.4.2.1p4 suggests a neat way for the implementation of the detection of keywords.
- Do not intermingle lexing with the output of the tokens. The interface to read tokens will be used again by the parser.
- It will prove helpful in the following phase to end the token stream of your lexer with a (not printed) end-of-file token.
- The soft deadline for this milestone is 2025-11-09.

[1] While this is not necessary the most helpful error location for developers using the compiler, it is better for automated testing.