

CSCI 355 - Compiler Construction, Fall 2025

Hans de Nivelle

October 29, 2025

1 Read the Syllabus!

This syllabus contains important information about the course CSCI 355. Read it completely. I will always assume that you are familiar with its contents.

2 Time/Location/People

Lectures are on Tuesdays and Thursdays from 15.00 until 16.15 in room 7.522. Lectures are on-site, unless something unexpected happens (e.g. a new pandemic.) Note that **Students are required to attend lectures**. If you decide that you won't attend lectures, you will miss important information and have less exposure to the course material, which means that you are choosing to learn less, and obtain a lower grade.

The preferred way of contacting me is always through Piazza. Don't send email to me directly. Questions related to your personal situation, or containing partial solutions of an assignment, should be marked as private.

General questions about the course contents should be public. If you are in doubt, make the question private, because I can always change it to public later.

3 Prerequisites

The course is intended for undergraduate students of Computer Science. Prerequisites are CSCI 235 (Programming Languages) and CSCI 272 (Formal Languages), both with a grade of *C* or more.

Since a course on compiler construction traditionally involves a lot of programming, you must have good programming skills, and find pleasure in programming. You must be self-motivated, and not procrastinate.

4 Course Contents

During the course, students will learn how high-level programming languages are translated into something that can be executed by the computer. Our focus

will be mostly on translation to low-level, native code, but we also consider languages that use a different intermediate representation.

Compilers typically perform the following tasks:

lexical analysis (also called *tokenizing* or *scanning*). Separate the input (just a sequence of characters) into the words of the programming language. These are typically keywords, identifiers, numbers, and special operators. In most languages, the tokenizer removes comments and whitespace.

parsing Using the result of lexical analysis, the parser analyzes the high-level structure of the program, and creates a tree representation, called *abstract syntax tree* (*AST*).

type checking/overload resolution Using the AST, check that all variables are declared, and that operations occurring in the AST have a possible translation. If necessary, insert conversions, and when an operation has more than one implementation, decide which one should be used. This is called *overload resolution*.

lowering Translate the type-checked AST into a flow graph containing low level instructions. In this flow graph, all operations are simple, and unambiguously defined. It uses local variables, that do not have an address.

optimization Using low-level representation, reorganize code so that it becomes more efficient: Remove repeated computations, remove unused computations, reorganize loops, simplify linear dependencies, replace expensive operations by cheaper operations.

instruction selection In the final stage, intermediate code is replaced by machine instructions. This is the only stage that is CPU dependent. Restrict the number of local variables so that they fit into CPU registers. After that, select instructions. This is usually done by matching expressions trees of the instructions against expression trees in the intermediate representation.

preprocessing Some languages (*C* and *C++*) use a preprocessor. This preprocessor can define abbreviations (macros), turn off parts of the code, and deal with include files.

In some sense, the preprocessor is like a separate compiler, because it has its own tokenizer, parser, variable lookup, and expression evaluation.

5 Course Organization

The course has two lectures of 75 minutes each. You are expected to attend the lectures. If you don't attend lectures, you may miss important information, which may negatively affect your understanding of the subject matter, your grade, your intellectual satisfaction, and your value on the job market. Per

university rules, you are required to attend lectures. I will always assume that you have received all information that was presented during lectures.

The course consists of four parts (parsing, type checking, lowering, optimization). The first two parts will have a midterm exam. The last two parts will be covered in the final exam, together with the first two.

Exams will be always on-site and require your physical presence. I fix the midterm exam dates in agreement with you, but once a date is fixed, it won't be moved, and no separate make up times will be given.

The final exam covers all three modules. The exam requires physical attendance, and its time will be decided by the registrar.

In addition to the midterms, there will be homeworks, which will be graded on the basis of effort.

There are three programming exercises, which will be graded on functionality and coding style. I may decide that some of the programming exercises will be made in groups. In that case, I will ask you to form groups of two or three students.

Late submissions of homeworks or programming assignments will not get the full number of points, and very late submissions will not earn any credit at all, unless a *sprawka* is presented to the school office.

We use Piazza for class discussions. Never write to me directly. I will also use Piazza for all announcements, and for uploading course materials. Piazza sends regular messages by email. Make sure that your email settings are such that you can see these messages. Make sure that they don't go into your spam folder.

If you are too ill to attend an exam or submit an assignment, you should **(1)** inform me as soon as you know and are reasonably able to, **(2)** get a *sprawka* and submit it to the school, **(3)** contact me again without delay when you are recovered.

Students are expected to read the uploaded course materials, and to reload when I announce an update. The course is mostly based on slides. You are not required to use any text books, but there is a list of recommended literature in Section 6.

6 Course Materials

The main course material are the slides, which will be regularly posted on Piazza. You must be aware that this course is constantly improving, errors are being made and corrected, so you must regularly reload the slides.

In addition, you will need *Maphoon* (a lexer and parser generator) developed by me, and available from [2]. The following literature is useful as background material:

The book *Compiler Design: Virtual Machines* ([5]) gives a very good overview of the compilation process for various languages, including functional languages and Prolog. In my view, it is a bit more formal than necessary, but it contains a lot of information.

The book *Compiler Design and Implementation* ([4]) is very detailed on optimizations, but at the same time a bit old-fashioned, mostly in the programming examples. Also the treatment of theory is bit old fashioned. It still contains a lot of good ideas.

The book *Compiler Design: Analysis and Transformation* ([6]) discusses compiler optimizations in depth. I am planning to use this book in the final part of the course.

The site *Compiler Explorer* ([3]) allows you to type code in different programming languages, and see the resulting assembler code. It is frequently used for checking if code has been efficiently optimized. It can also settle arguments whose code is better. I prefer by myself to use `clang` with `-emit-llvm -S` because I find LLVM more readable than assembly, and LLVM is machine independent.

The *Dragon Book* ([1]) is a classic source on compiler construction. It is very detailed on the parsing process, explaining the language theory behind it, and this theory is still valid now. At the same time it is a bit short on the rest of the compilation process, and became outdated.

The Compiler Design Handbook is very technical. It describes SSA (static single assignment) in a way that is not used in LLVM. In my view, it contains a lot of theoretical approaches which are not used in real compilers. It is still useful to have a look at it if you are brave enough.

7 Programming

A course on compiler construction traditionally has a large programming component. You will be implementing a tokenizer, a parser and a type checker, possibly also an intermediate code generator. You will need to use `C++` for this. Make sure that you have a compiler that supports `C++-20`. Check that you can compile `std::variant`, `std::string_view`, and concepts. Check if your compiler accepts the following includes:

```
#include <variant>
#include <optional>
#include <string_view>
#include <concepts>
```

8 Academic Integrity

The government of Kazakhstan has installed Nazarbayev University, because it believes that a western style university is essential for the economical, scientific and cultural development of Kazakhstan.

Two things are essential for the functioning of teaching at NU: **(1)** Students who graduated from NU are highly-qualified, independent workers whose judgement and integrity can be trusted without hesitation, and **(2)** Everyone, in Kazakhstan and in the rest of the world, has trust in Nazarbayev University.

In order to obtain these long term goals, Nazarbayev University and the School of Engineering and Digital Sciences have established high standards for academic integrity, using an approach in which students are trained to produce original work according to professional standards, and to properly cite and reference the work of others when it is appropriate to do so.

Any form of academic misconduct causes damage to the above mentioned long-term goals of Nazarbayev University. I will assume that you understand this and that you will not attempt any form of academic misconduct. In particular, do not try any of the following:

- Hand in work that you did not make completely by yourself. Never copy any work from another student. In some cases, which I will specify, some forms of cooperation are allowed, but blind copying and copying with small changes, are always academic misconduct.
- Show or send your work to other students. Again, it is usually allowed to explain things to other students, but it is never allowed to show or give them your solutions.
- In case of a group assignment, share or show your work with any student who is not in the same group as you.

Don't beg for a better grade. It is allowed to have a look at an exam with the purpose of improving yourself. It is also reasonable to expect that grading errors will be corrected, but it is not OK to ask for better grade just because you want it, or because you think that the exam was checked too strictly. We apply the same rules to everyone equally, and try to meet global standards. A grade is a measurement how well the student meets the learning goals of the course, nothing else.

Finally, we remind you that:

- If we discover academic misconduct such as plagiarism or other forms of cheating, the student will receive no credit for the work in question, and the event will be reported to the Dean of SEDS.
- Severe cases, or a repeated offense, may result in failure of the course and suspension or expulsion from the university. This is decided by the school.
- Programming exercises may be tested by an automated plagiarism detector, with or without suspicion. Discovered plagiarism will result in zero points, and a possible report to the Dean of SEDS.
- In the case a student argues very much about grades for a part of an exam, this could be interpreted as the student understanding less of the topic, than was initially assumed during grading.

9 Expected Behavior

9.1 During Lectures

You are expected to attend lectures, to be attentive during these lectures, and to arrive on time. You are expected to read and study the discussed material soon after the lecture (preferably on the same day), and try to read in advance for the next lecture. Relevant materials will be made available on Piazza.

9.2 During Quizzes or Exams

In the rest of this paragraph, 'exam' and 'quiz' are treated as synonyms. Exams will be announced at least one week in advance. You are expected to be on time on the indicated place. Being late does not guarantee make up time. During the exam you cannot speak with other students. You are not allowed to use any electronic devices, including your phone. You are not allowed to use additional materials (books or notes). You have to move place if an instructor asks this.

9.3 During Reviewing Sessions

We allow students to review exam results. Students are not allowed to take their work home or take pictures of it. Purpose of reviewing sessions is to allow the students to see what mistakes were made during the exam, with the purpose of having a last chance to understand the material before the lecture moves on to another topic. The purpose is not to argue about grades. Don't put pressure on instructors to change a grade. A grade is an objective measurement of how much the student understood of the course material. Giving in to pressure would be unfair towards other students, and harm the functionality of grades. If the student believes that a grade is unfair, the student can submit a grade correction request. Grades are never corrected in the presence of the student.

10 Grading

Final grades are calculated as follows:

Midterm Exams (two)	30
Programming Assignments (three)	30
Homeworks (probably three)	10
Class Participation	5
Final Exam	25
<hr/> Total	<hr/> 100

Work that is handed in late without medical excuse (sprawka) will not receive full credit. If the work is very late, it will not be graded at all, resulting in 0 points.

For calculation of the final letter grades, we use Nazarbayev University's standard cut off borders:

Grade	At Least
A	≥ 95
A ⁻	≥ 90
B ⁺	≥ 85
B	≥ 80
B ⁻	≥ 75
C ⁺	≥ 70
C	≥ 65
C ⁻	≥ 60
D ⁺	≥ 55
D	≥ 50
F	≥ 0

11 Schedule (Likely to Change)

Week	Dates	Topics
1	August 18 - 24	History of Compilers; Main Components of a Compiler
2	August 25 - 31	Short demonstration of LLVM and generated assembly code; Tokenizing; Some necessary C++ : <code>std::variant</code> ; moving; Writing a tokenizer by hand
3	September 1 - 7	Regular Expressions; Use of Maphoon for Tokenizing
4	September 8 - 14	Attribute Grammars; Parsing; Expressing Priorities; Top Down Parsing
5	September 15 - 21	Bottom Up Parsing; The Prefix Automaton
6	September 22 - 28	Bottom Up Parsing
7	September 29 - October 5	Lisp Language, Discussed Homework
	October 6 - 12	Fall Break
8	October 13 - 19	Type Checking and Overload Resolution
9	October 20 - 26	Type Checking and Overload Resolution
10	October 27 - November 2	TCOR, Intermediate Code Generation (1)
11	November 3 - 9	Intermediate Code Generation (2), Low Level Virtual Machine (LLVM)
12	November 10 - 16	Optimization : Alias Detection
13	November 17 - 23	Optimization : Elimination of Common Subexpressions
14	November 24 - 30	Review Week
	December 1 - December 11	Final Exam Period. There will be a final exam over the complete material.

References

- [1] Alfred Aho and Jeffrey Ullman. *Principles of Compiler Design*. Addison-Wesley Publishing Company, 1977.

- [2] Hans de Nivelle. Parser and tokenizer generator Maphoon, 2021. Can be obtained from [here](#).
- [3] Matt Godbolt. Compiler explorer. [Link](#).
- [4] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
- [5] Reinhard Wilhelm and Helmut Seidl. *Compiler Design: Virtual Machines*. Springer Verlag, Heidelberg, Dordrecht, London, New York, 2010.
- [6] Reinhard Wilhelm, Helmut Seidl, and Sebastian Hack. *Compiler Design: Analysis and Transformation*. Springer Verlag, Heidelberg, Dordrecht, London, New York, 2012.