Course Notes
Concepts of Programming Languages

H.A. Hiep and S.O. Brand
Leiden University

Fall 2023
Version: January 9, 2026

Contents

Overview

> "If you find that you're spending almost all your time on theory, start turning some attention to practical things; it will improve your theories. If you find that you're spending almost all your time on practice, start turning some attention to theoretical things; it will improve your practice." — Donald Knuth

Introduction

Computer programming is the activity of specifying what computers do, and programming languages are the main tool to achieve that. This course offers a kaleidoscopic view on the many concepts of programming languages, focusing on object-oriented programming and functional programming.

We will have a look at the most important concepts underlying (the theory of) programming languages. But, we will also practice with different programming languages, to get a firm grasp on the involved concepts. In one way or another, the concepts we shall discuss are present in almost all (high-level) programming languages. To keep the course accessible, some concepts will be merely introduced, thereby providing an overview of directions for further study.

This course comprises both theoretical and practical aspects. Theoretical topics include: methods to describe language syntax (BNF), semantics of programming languages (denotational and operational semantics), variables and expressions, imperative languages (and the **while** language), functional languages (and the λ-calculus), parameter passing mechanisms (call-by-value, call-by-name, and others), types and type systems, polymorphism, higher-order functions, object-orientation and generics, concurrency, and theorem proving. Practical topics include: language generators and string rewriting, advanced imperative and object-oriented programming, and basic functional programming.

Having followed the courses Foundations of Computer Science (that introduces concepts from discrete mathematics: sets, relations, induction), Introduction to Logic (that introduces propositional logic and first-order logic) and Programming Techniques (that gives the student an intuition of the practical activity of programming) is highly recommended, but not mandatory. However, basic programming ability (e.g. in a modern programming language such as Python, C++, Java, etc.) is necessary in order to be able to follow the course. Knowledge of any other programming language is a plus but not mandatory.

Objectives

The main goal of the course is to provide insights into why there is a need for high-level programming languages, the (relationship between) concepts of programming languages, and practice with how some of these concepts could be implemented. This enables the student to learn new programming languages (or related techniques employed in the programming context) faster, and make more informed decisions about what to use and when.

At the end of the course the student has acquired:

- basic knowledge of programming language theory,
- basic knowledge of imperative programming concepts (such as structured programming),
- basic knowledge of functional programming concepts (such as pattern matching, algebraic data types, and recursion),
- basic knowledge of operational and denotational semantics (needed for an advanced course in program correctness and computational models),
- basic knowledge of concurrent programming (needed for an advanced course in concurrency),
- basic knowledge of automated and interactive theorem proving (needed for an advanced course in logical verification),
- practical experience with advanced imperative and object-oriented programming features (such as designing class hierarchies and defining and using abstract data types),
- hands-on experience with implementing the (simply typed) lambda calculus, type checking and type inference (needed for an advanced course in type theory).

Mode of instruction & material

Instruction is structured around *lectures*, *exercises*, and *practical labs*. Lectures are digital and pre-recorded, 2 hours per week. During exercise classes, we discuss theory and work on a selection of exercises, also 2 hours per week. During the lab sessions, students work in groups on practical assignments, also 2 hours per week. Both exercise classes and lab sessions are organized on location.

The relevant lecture videos, per week, are linked in these course notes. Students are expected to watch lecture videos (and optionally additional material) on their own. These sources are useful for preparing for exercises in class and the group assignments. Students should be critical of the provided (video) material: any questions that arise while watching the videos, or reading the literature, can be discussed during class.

There main source for lecture videos is:

- Concepts of Programming Languages by drs. H.A. Hiep (dated 2021):
  YouTube playlist

Additionally the following lectures can be used as sources:

- Functional Programming by prof.dr. Jürgen Giesl (dated 2012):
  YouTube playlist
- Programming Methodology by prof.dr. Mehran Sahami (dated 2007):
  YouTube playlist

Additionally the following books can be used as sources:

- *Concepts of Programming Languages* (11th edition)
  by Robert W. Sebesta (2016)
  ISBN: 978-1-292-10055-5
  Online copy
- *Programming Language Concepts* (2nd edition)
  by Peter Sestoft (2017)
  ISBN: 978-3-319-60788-7
- *Concepts in Programming Languages*
  by John C. Mitchell (2003)
  ISBN: 0-521-78098-5

Additionally the following articles can be used as sources:

- *A Tutorial Introduction to the Lambda Calculus*
  by Raúl Rojas (2015)
  Open access
- *An Introduction to Lambda Calculus and Functional Programming*
  by Rodrigo Machado (2013)
  in *2nd Workshop-School on Theoretical Computer Science* (IEEE)
  Online copy
- *A short introduction to the Lambda Calculus*
  by Achim Jung (2014)
  Online copy

The purpose of the exercise classes is to actively discuss exercises, and learn how to be able to answer questions in a satisfactory manner. During the exercise classes, students use pen and paper (or anything that works like pen and paper). Attending and actively participating in exercise classes is a useful preparation for the final exam (discussed in the next section). At the start of each exercise class, the teacher may ask quiz questions (15 minutes) about the material provided in the lectures.

For students who want to practice more: these course notes contain short summaries of what was discussed during the exercise class, and additional exercises. Some of these exercises also come with solutions. Exercises look like:

Exercise 0.1. This is an exercise.

Exercise 0.2***.** This is a difficult exercise.

The labs are for getting started, collaborating, and requesting/obtaining feedback on assignments. In the first lab session, students form groups. In the remaining lab sessions, students mainly collaborate within these groups. Most assignments are worked on in groups. Students may also request feedback from fellow students: students are encouraged to engage in *peer reviewing*. All work should be publicly available through the Web, e.g. by using a service such as GitHub or GitLab. Students should make proper use of version control to keep track who has committed what. Students may be asked to bring their own device (e.g. if there are not enough computers available in the room).

Assessment method

To test that a student comprehends concepts in sufficient detail, there are two kinds of assessments: assignments, and a final exam.

The purpose of the assignments are:

- to learn concepts of unfamiliar programming languages,
- to learn how to implement programming concepts and thereby gain a better understanding of the involved concepts,
- to engage with, and critically reflect on, the scientific literature.

The purpose of the final exam is to test the individual ability of students to answer questions about programming concepts in a satisfactory manner.

Assignments will be mixed individual work, and group work in groups of a maximum of 3 people. The number of assignments will be around 4. Self-study and group work is necessary to finish the assignments. At the start of the semester, students should register their group using the following form (one registration per group is sufficient):

https://forms.gle/XBoAqyp7Tfcb5Bpt9

Teaching assistants will—at random intervals—look at the work and give feedback to each group on their assignments, including a rough estimate of the grade. There is only a single deadline to submit the final work (of all assignments) at the end of the semester, after which the final grade of the assignments is determined.

The final exam is a written exam (using pen and paper), and consists of a selection of (the kind of) questions that are discussed during the exercise classes. The exam consists of two parts:

- quizzes about basic knowledge,
- hands-on exercises about the lambda calculus.

The first part of the exam tests basic knowledge in a similar way as the quizzes and is based on the material in the lectures. The second part of the exam tests the ability of students to solve questions concerning the lambda calculus. These course notes also comprise, for each exercise class, a list of additional exercises, including some of their solutions. These course notes are updated several times during the semester—students can request more exercises if needed. ++ The final grade is calculated from the following components:

- Assignments (50%)
- Final exam (50%)

It is required to achieve at least 5.5 points on average for the assignments, and also for the final exam in order to pass the course. Grades of previous years may be reused this year, but only for one of the two components, and not the individual grades of assignments. The teacher will inform the students how the inspection of the graded assignments and final exam will take place.

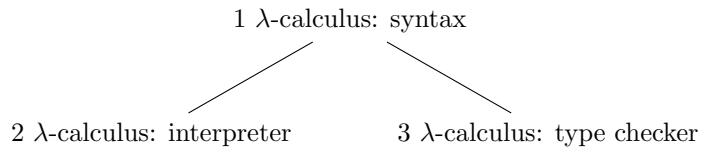

Figure 1: Overview of dependencies between programming assignments. An assignment connected from bottom to top means that work on the assignment below depends on completing the connected assignment above.

There are 4 practical assignments in total, and students must complete all assignments. Note that implicitly there are dependencies between some of the assignments: completing an earlier assignment is sometimes necessary before working on the next assignment makes sense. See Figure 1 for a dependency graph between the assignments.

Students are *allowed* to use OpenAI's ChatGPT or GitHub's CoPilot (or similar tools) to assist them while working on the assignments (signing-up and applying for a student pack is the student's own responsibility). In doing so, all interactions (input submitted and output received) with the tools *must* be recorded and stored alongside submitted work in a separate file included in the work submitted. Much trust is placed in students to responsibly use the involved tools and keep a record of their usage.

There is one individual assignment: a critical literature review. In this assignment, the student will ask ChatGPT (or any similar large language model) to summarize concepts and critically analyze the output of the chat bot by relating it to the scientific literature.

Part I

Exercise classes

Exercise class 1

Overview

We discussed three major ideas that underlies programming languages:

- programming languages can be divided in roughly three paradigms:
  - *imperative programming*,
  - *functional programming*, and
  - *logic programming* (or *theorem proving*).
- in the 1970s the *software crisis* started, and as a result *high-level* programming languages were developed to help tackle the problems in *software engineering*.
- that one can understand programming languages from different points of view. We distinguish the following viewpoints: *syntax* and *semantics*.

Exercise 1.1. Give an overview of the most widely-used languages, and classify them according to the three paradigms. What programming languages are considered low-level or high-level? What programming languages can be translated into what other programming languages (e.g. by means of a compiler)?

Exercise 1.2. What makes a programming language high-level or low-level? What are the highest-level programming languages, and what are the lowest-level?

Exercise class 2

Syntax

We discussed a general method for describing the *syntax* of a programming language: unrestricted grammars.

Rules of unrestricted grammars can be restricted to obtain different classes of grammars. One such class is the context-free grammar. One way of writing down grammars is the so-called Backus-Naur form (BNF).

Exercise 2.1. Given the following unrestricted grammar (starting at S):

$$S \longrightarrow (S)$$
$$S \longrightarrow SS$$
$$S \longrightarrow$$

This grammar describes the so-called Dyck language of balanced parentheses.

- Give a derivation of the string (()(()())()).
- Give a derivation of the string ((()))(()).
- Draw the derivation tree (containing all derivations), up to four steps deep. What terminal strings can be derived using at most four steps?

Exercise 2.2. Given the following abstract grammar (in BNF):

$$E ::= x \mid n \mid (E + E) \mid (E \times E)$$

where x is any variable and n is any number.

- Write the abstract grammar in the previous format where the production rules are clear.
- Why are the parentheses necessary around the constructors for plus and times?
- Show an abstract syntax tree obtained when parsing the string: $((1+x)\times y)$

Exercise 2.3. Given the following abstract grammar (in BNF):

$$F ::= (E = E) \mid \neg F \mid (F \wedge F)$$

where E is as given in the previous exercise. Are the following strings part of the language? If yes, draw the abstract syntax tree. If not, explain why.

- $((5 = y) \wedge \neg(5 = y))$
- $((x = y) \wedge (y = z) \rightarrow (x = z))$
- $\neg\neg(x = y) \wedge (y = x)$
- $(x = 5 + 5) \wedge ((5 + 5) = x)$

How do you deal with ambiguity? Explain what is syntactic sugar, and how to introduce $\vee$ and $\rightarrow$ in such way?

Exercise class 3

Variable binding

Exercise 3.1. Given the following lambda expression

$$\lambda y(\lambda y(y\ y))$$

1. To which λy are the variables $(y\ y)$ bound in this expression?
2. Is $\lambda y(\lambda y(y\ y)) \to_\alpha \lambda x(\lambda y(y\ y))$ a valid α-conversion?
3. Are the expressions $\lambda y(\lambda y(y\ y))$ and $\lambda x(\lambda y(y\ y))$ equivalent? I.e. $\lambda y(\lambda y(y\ y)) \stackrel{?}{\equiv} \lambda x(\lambda y(y\ y))$. If so, prove this.

Answer 3.1.

1. The variables $(y\ y)$ are bound to the right-most λy.
2. This is not a valid α-conversion because it does not follow the rule $(\lambda x M) \to_\alpha (\lambda y M[x := y])$. Renaming the outer λy using this rule would give $\lambda x(\lambda x(x\ x))$.
3. The expressions can be shown to be equivalent through α-conversions by renaming the inner λy first: $\lambda y(\lambda y(y\ y)) \to_\alpha \lambda y(\lambda w(w\ w)) \to_\alpha \lambda x(\lambda w(w\ w)) \to_\alpha \lambda x(\lambda y(y\ y))$.

Exercise class 4

Substitution

Exercise class 5

Reduction

Exercise 5.1. Keep performing reductions on the following expressions:

1. $\lambda x((\lambda x\ x)x(\lambda z\ x))z$
2. $\lambda y(\lambda z.y\ z)(z\lambda x.x)$

Answer 5.1.

1. $\lambda x(\underbrace{(\lambda x\ x)}_{\beta}x(\lambda z\ x))z \to_\beta \lambda x(x(\underbrace{\lambda z\ x}_{\alpha}))z \to_\alpha \underbrace{\lambda x(x(\lambda w\ x))z}_{\beta} \to_\beta z(\lambda w\ z)$
2. $\lambda y(\lambda z.y\ z)(z\lambda x.x) = \underbrace{\lambda y(\lambda z(y\ z))(z\lambda x(x))}_{\beta} \to_\beta \lambda w((z\lambda x(x)w)$

Exercise class 6

Type checking

Exercise class 7

Type inference

Exercise class 8

Encoding arithmetic

(discuss successor function, addition, multiplication during class)

Exercise 8.1. Given the following lambda expressions

- $m^n := \lambda m \lambda n (n\ m)$
- $0 := \lambda f (\lambda x\ x)$
- $1 = \lambda f (\lambda x (f\ x))$
- $2 = \lambda f (\lambda x (f (f\ x)))$
- $3 = \lambda f (\lambda x (f (f (f\ x))))$

show that

1. $2^0 = 1$
2. $3^1 = 3$
3. $1^3 = 1$

Answer 8.1. TODO

1. $2^0 = 1$
2. $3^1 = 3$
3. $1^3 = 1$

Exercise class 9

Recursion

Exercise class 10

Imperative programs

Part II

Assignments

Organization of repository

To facilitate automated testing of your submitted code, please make sure your repository is organized as follows:

```
repository root/
├── assignment 1/
│   ├── Makefile
│   ├── positives.txt
│   ├── // rest of your code
│   └── // structured as you see fit
├── assignment 2/
│   ├── Makefile
│   ├── positives.txt
│   ├── // rest of your code
│   └── // structured as you see fit
└── assignment 3/
    ├── Makefile
    ├── positives.txt
    ├── // rest of your code
    └── // structured as you see fit
```

- Make sure that at the root of your repository you have three directories: `assignment 1`, `assignment 2`, and `assignment 3`.
- Have a file `positives.txt` in each assignment directory which contains at least one lambda expression which can be read by your program.
- Have a file `Makefile` in each assignment directory. This Makefile should have 2 rules: `build` and `run` (see example Makefiles below).
  - The `build` rule in the Makefile should compile the program. If you are using a non-compiled language (e.g. Python) there should still be a `build` rule, but it can be just be an `echo` statement.
  - The `run` rule in the Makefile should run the program on the content of `positives.txt`.

Example `Makefile` for C++:

```
# Makefile for simple C++ program

CC=gcc
CXX=g++
RM=rm -f
CPPFLAGS=-g $(shell root-config --cflags)
LDFLAGS=-g $(shell root-config --ldflags)
LDLIBS=$(shell root-config --libs)

SRCS=tool.cc support.cc
OBJS=$(subst .cc,.o,$(SRCS))

all: tool

tool: $(OBJS)
    $(CXX) $(LDFLAGS) -o tool $(OBJS) $(LDLIBS)

tool.o: tool.cc support.hh

support.o: support.hh support.cc

clean:
    $(RM) $(OBJS)

distclean: clean
    $(RM) tool

build: tool
        echo finished

run: build
        ./tool positives.txt
        # alternatively:
        # ./tool < positives.txt
        # if your program reads from the standard input (stdin)
```

Example `Makefile` for Python:

```
# Makefile for simple Python program

build:
        echo "nothing to build"

run:
        python tool.py positives.txt
        # alternatively:
        # python tool.py < positives.txt
        # if your program reads from the standard input (stdin)
```

Example `Makefile` for Java:

```
# Makefile for simple Java program

JC = javac
JVM = java
JARCMD = jar

SOURCE = Main.java
MAIN = Main
JAR = program.jar

.PHONY: clean run

default: build

build: $(SOURCE)
        $(JC) $(SOURCE)

jar: build
        $(JARCMD) cvfe $(JAR) $(MAIN) *.class

run: build
        $(JVM) $(MAIN) positives.txt
        # alternatively:
        # $(JVM) $(MAIN) < positives.txt
        # if your program reads from the standard input (stdin)

clean:
        $(RM) *.class $(JAR)
```

Assignment 1

λ-calculus: syntax

1.1 Introduction

In this assignment, you will write a program that lexically analyses and parses a piece of text. The grammar used is that of the lambda calculus, as explained during the lectures, and specified below.

The assignment submission must include a program that:

- reads an expression from standard input into a character string
- lexically analyzes the character string into a string of tokens
- parses the token string (e.g. using recursive descent)
- outputs a character string in a standard format to standard output

The standard format is not specified: this format has to be determined by the designers of the program themselves, and may be explained in the README.

The program must be able to detect syntax errors and then should report an error. The program must not make use of external libraries. The program should use the least amount of standard library code. The assignment submission must include a Makefile that can be used to compile the program if necessary.

The assignment submission must include a README file that documents:

- The student number(s) of the student(s) who worked on the assignment.
- Whether it is known that the program works correctly, or whether the program has known defects.
- Whether there are any deviations from the assignment, and reasons why.

The README may include an explanation of how the program works. Finally, the assignment submission may include the following two files:

- An archive (`positive.zip`) of the positive examples used for testing.
- An archive (`negative.zip`) of the negative examples used for testing.

1.2 Interface

The program must be compilable (if applicable) and work on the command line. The program can then be invoked using the command line. It does not accept any command line arguments.

Input. The program reads a string of characters from the standard input. It is permissible that the program works only for inputs that contain only printable ASCII characters and whitespace. The program may accept multiple expressions, one per line. A line is considered terminated by a newline character (`\n`), or a carriage return and newline character (`\r \n`).

Exit status. The program must exit with exit status 0 whenever all expressions in the input are parsed correctly. The program must exit with exit status 1 whenever there is a syntax error. The program must terminate, and should terminate within a short amount of time (no noticeable delay to a human).

Output. If the program exists with exit status 0 then the program must produce output to the standard output. If the program exists with exit status 1, then it must not print to standard output, and an error message may be printed to standard error. The program may print understandable error messages.

1.3 Grammar

The input is analyzed using the following Backus-Naur grammar:

⟨expr⟩ ::= ⟨var⟩ | `'('` ⟨expr⟩ `')'` | `'\'` ⟨var⟩ ⟨expr⟩ | ⟨expr⟩ ⟨expr⟩

where ⟨var⟩ stands for any variable name. A variable name is alphanumerical: it consists of the letters a-z, A-Z, or the digits 0-9. A variable name must start with a letter from the Latin alphabet, i.e. not with a digit. The grammar should be whitespace insensitive, but whitespace must be used to separate application of two variables. The program may support international variable names (i.e. Unicode), and accept λ instead of \.

The program must support using parentheses in the input to disambiguate expressions. If no parentheses are used, the order of presedence for the operators is as follows: lambda abstraction groups more strongly than application (i.e. abstraction precedes application), and application associates to the left. The program may support a dot after the lambda abstraction variable, where the dot is parsed in the same way as if an opening parenthesis was inserted at the place of the dot with a matching closing parenthesis placed before the next unmatched closing parenthesis or the end of the expression. More generally, the program may support a dot at any place where a closing parenthesis immediately followed by an opening parenthesis could make sense—with the intended meaning that the matching previous open parenthesis is inserted at the place as far as possible to the left, and the matching next closing parenthesis is inserted at the place as far as possible to the right.

If parsing is succesful, the output of the program must be again acceptable by the program to parse: the program then succesfully parses its own output and

should produce the exact same result. The output should be an unambiguous expression, i.e. with sufficiently many parentheses inserted so the parser never applies any of the precedence rules. The output may use the least amount of whitespace and parentheses in its output.

1.3.1 Positive examples

The following examples are acceptable (the program must work if the program is invoked separately and supplied each expression, the program should accept multiple expressions each on its own line):

- `(a b)`
- `abc`
- `a b c`
- `a (b c)`
- `(\ x a b)`
- `(\x((a) (b)))`

1.3.2 Negative examples

The following examples are not acceptable:

- `\` (missing variable after lambda)
- `\x` (missing expression after lambda abstraction)
- `((x` (missing closing parenthesis)
- `()` (missing expression after opening parenthesis)
- `a (b` (missing closing parenthesis)
- `a (b c))` (input string not fully parsed)

1.4 Evaluation criteria

The submission will be evaluated on the following criteria:

- Correctness of the program (hard criterium, 60%): is the program correctly implementing the assignment? Are there cases in which the program is implemented incorrectly?
- Readability of the program (soft criterium, 30%): is the program written to be understandable to humans too?

- Efficiency of the program (soft criterium, 10%): is program executing without noticable delay?

In the above text, the words must, should, and may have a special meaning. The assignment is graded with a passing grade if all features that must be implemented are implemented correctly. Higher grades are for submissions that also correctly implement features that should be implemented. Even higher grades are for submissions that also correctly implement features that may be implemented.

Assignment 2

λ-calculus: interpreter

2.1 Introduction

In this assignment, you will write a program that interprets expressions in the lambda calculus. The grammar used is the same as the previous assignment, and specified below.

The assignment submission must include a program that:

- reads an expression from a file into a character string
- lexically analyzes and parses the expression into an abstract syntax tree
- performs reductions on the abstract syntax tree, until no longer possible
- outputs a character string corresponding to the final abstract syntax tree

The program must be able to detect syntax errors and then should report an error. The program may terminate with an error after a fixed amount of reduction steps (e.g. 1000) if then there are still reductions possible. The program must not make use of external libraries. The program should use the least amount of standard library code. The assignment submission must include a Makefile that can be used to compile the program (if applicable).

The assignment submission must include a README file that documents:

- The class and group number, and the names of the student(s) who worked on the assignment. (Putting the names of the student(s) in each source file is good practice too.)
- The compiler version and operating system used by the student(s) if applicable.
- Whether it is known that the program works correctly, or whether the program has known defects.
- Whether there are any deviations from the assignment, and reasons why.

The README may include an explanation of how the program works, and remarks for improving the assignment. Finally, the assignment submission may include the following two files:

- An archive (`positive.tar.gz`) of the positive examples used for testing.
- An archive (`negative.tar.gz`) of the negative examples used for testing.

2.2 Interface

The program must be compilable (if applicable) and work on the command line. The program can then be invoked using the command line. It accepts one required command line argument, namely the file from which it reads.

Input. The program reads a string of characters from the file named by the first command line argument. The program must work for files which contain only printable ASCII characters and whitespace but may also work for files which contain non-printable ASCII characters. The program should accept only one expression in the input file.

Process. After parsing the input into an abstract syntax tree, the program performs α-conversions and β-reductions. The α-conversions should only be performed if a β-reduction would otherwise lead to a captured variable. If there are multiple places in an expression where a β-reduction can be performed, the program chooses an arbitrary place where reduction is performed. The reduction strategy may be configured, e.g. using command line flags. The README must document what reduction strategies are implemented.

Exit status. The program must exit with exit status 0 whenever the expression cannot be reduced any further by a β-reduction. The program must exist with exit status 1 whenever there is a syntax error, or not enough command line arguments are supplied. The program may exit with exit status 2 whenever a limit on the number of reduction steps has been reached. Alternatively, if the program runs forever on some inputs it must be interruptable by the operating system.

Output. If the program exists with exit status 0 then the program must have outputed the reduced abstract syntax tree, in a standard format, to the standard output. It is permissible that the program outputs intermediary abstract syntax trees on standard error, using a special debugging constant in the program to enable/disable such verbose, diagnostic output. If the program exists with exit status 1 or exit status 2 then an error message may be printed to standard error. The program may print understandable error messages.

2.3 Grammar

The input file is analyzed using the following Backus-Naur grammar:

⟨expr⟩ ::= ⟨var⟩ | `'('` ⟨expr⟩ `')'` | `'\'` ⟨var⟩ ⟨expr⟩ | ⟨expr⟩ ⟨expr⟩

where ⟨var⟩ stands for any variable name. A variable name is alphanumerical: it consists of the letters a-z, A-Z, or the digits 0-9. A variable name must start with a letter from the alphabet, i.e. not with a digit. The grammar should be whitespace insensitive, but whitespace must be recognized to separate application of two variables. The program may support international variable names (i.e. Unicode), and also accept λ instead of \.

The program must support using parentheses in the input to disambiguate expressions. If no parentheses are used, the order of presedence for the operators is as follows: lambda abstraction groups more strongly than application (i.e. abstraction precedes application), and application associates to the left. The program may support a dot after the lambda abstraction variable, where the dot is parsed in the same way as if an opening parenthesis was inserted in the place of the dot with a matching closing parenthesis extending as much as possible to the right.

2.3.1 Positive examples

The following examples show input expressions and their reductions:

- `(x y)` contains no place where β-reduction can be performed, so this expression is the output
- `\x\y(x\z y)` contains no place where β-reduction can be performed
- `(\x x)(\y y)` immediately β-reduces to `(\y y)` which is then the output
- `(\x \y x)(\z y)` first performs α-conversion to `(\x \w x)(\z y)` and then β-reduces to `(\w \z y)` which is then the output
- `(\x y)((\x (x x))(\x (x x)))` performs β-reduction and outputs `y` (N.B. compare with the negative example)
- `(\x x x)(\x x x)` β-reduces to `(x)(\x x x)` or `(\x x x)(x)` and then β-reduces to `(x x)`
- etc.

2.3.2 Negative examples

Invalid input causes the program to terminate with exit status 1:

- `(\x`
- `x x))`
- etc.

The following examples cause the program not to terminate or output with exit status 2 if it reaches a limit:

- `(\x (x x))(\x (x x))` β-reduces to `(\x (x x))(\x (x x))` etc. etc.
- `(\x y)((\x (x x))(\x (x x)))` keeps performing β-reduction in the right-hand side of the outer application (N.B. this is also valid behavior)
- etc.

2.4 Evaluation criteria

The submission will be evaluated on the following criteria:

- Correctness of the program (hard criterium, 60%): is the program correctly implementing the assignment? Are there cases in which the program is implemented incorrectly?
- Readability of the program (soft criterium, 30%): is the program written to be understandable to humans too?
- Efficiency of the program (soft criterium, 10%): is program executing without noticable delay?

In the above text, the words must, should, and may have a special meaning. The assignment is graded with a passing grade if all features that must be implemented are correctly implemented. Higher grades are for submissions that also correctly implement features that should be implemented. Even higher grades are for submissions that also correctly implement features that may be implemented.

Assignment 3

λ-calculus: type checker

3.1 Introduction

In this assignment, you will write a program that type checks expressions in the simply-typed lambda calculus. The grammar used is *almost* the same as in the first and second assignment, and specified below.

The assignment submission must include a program that:

- reads a judgement from a file into a character string
- lexically analyzes and parses the judgement into an abstract syntax tree
- checks whether the judgement is derivable in the type system

The program must be able to detect syntax errors and should report an error if detected. The program must not make use of external libraries. The program should use the least amount of standard library code. The assignment submission must include a Makefile that can be used to compile the program (if applicable).

The assignment submission must not include any binaries, and must include a README file that documents:

- The class and group name, and the names of the student(s) who worked on the assignment. (Putting the names of the student(s) in each source file is good practice too.)
- The compiler version and operating system used by the student(s).
- Whether it is known that the program works correctly, or whether the program has known defects.
- Whether there are any deviations from the assignment, and reasons why.

The README may include an explanation of how the program works, and remarks for improving the assignment. Finally, the assignment submission may include the following two files:

- An archive of the positive examples used for testing
- An archive of the negative examples used for testing

3.2 Interface

The program must be compilable and work on the command line. It accepts one command line argument, namely the file from which it reads.

Input. The program reads a string of characters from the file named by the first command line argument. It is permissible that the program only works for files which contain only printable ASCII characters and whitespace. The program accepts one judgement in the input file, but may process multiple judgements, one per line.

Process. After parsing the input into an abstract syntax tree, the program performs type checking. The type checking process must halt.

Exit status. The program must exit with exit status 0 if the judgement is derivable. The program must exist with exit status 1 whenever there is a syntax error, or the judgement is not derivable.

Output. If the program exists with exit status 0 then the program should output the judgement printed to standard output in an unambiguous and standardized output format (where each complex expression and type is surrounded by parentheses). It is permissible that the program outputs intermediary abstract syntax trees on standard error, using a special debugging constant in the program to enable/disable such verbose, diagnostic output. If the program exists with exit status 1 then an error message may be printed to standard error. The program may print understandable error messages.

3.3 Grammar

The input file is analyzed using the following Backus-Naur grammar:

⟨judgement⟩ ::= ⟨expr⟩ `':'` ⟨type⟩
⟨expr⟩ ::= ⟨lvar⟩ | `'('` ⟨expr⟩ `')'` | `'\'` ⟨lvar⟩ `'^'` ⟨type⟩ ⟨expr⟩ | ⟨expr⟩ ⟨expr⟩
⟨type⟩ ::= ⟨uvar⟩ | `'('` ⟨type⟩ `')'` | ⟨type⟩ `'->'` ⟨type⟩

where ⟨lvar⟩ stands for any variable name that starts with a lowercase letter, and ⟨uvar⟩ stands for any variable name that starts with an uppercase letter. A variable name is alphanumerical: it consists of the letters a-z, A-Z, or the digits 0-9. The grammar should be whitespace insensitive, but whitespace must be recognized to separate application of two variables. The program may support international variable names (i.e. Unicode), and also accept λ instead of \.

The program must support using parentheses in the input to disambiguate expressions and types. If no parentheses are used, the order of presedence for the expression operators is as follows: lambda abstraction groups more strongly than application (i.e. abstraction precedes application), and application associates

to the left. The function type constructor associates to the right. The program may support a dot after the lambda abstraction type, where the dot is parsed in the same way as if an opening parenthesis was inserted with a matching closing parenthesis before the next unmatched closing parenthesis or the end of the expression.

3.3.1 Positive examples

The following examples show input judgements (in a standard output format):

- `(\x^A (\y^(A->B) (y ((\x^A x) x)))):(A -> ((A -> B) -> B))`
- `(\x^A x):(A -> A)`
- `(\x^B (\x^A x)):(B -> (A -> A))`
- `(\y^A (\x^(A -> (C -> A)) (x y))):`
  `(A -> (A -> C -> A) -> C -> A)`
- etc.

3.3.2 Negative examples

Invalid input (according to the grammar) causes the program to terminate with exit status 1:

- `(\x x)` missing types
- `(\x^A (x y))` missing types
- etc.

3.3.3 Incorrect types examples

Inputs which follow the specified grammar but are incorrectly typed should also cause the program to terminate with exit status 1:

- `(x y):A` the variables `x` and `y` have an unknown type
- `(\x^A y):(A->B)` free variable `y` has unknown type
- `(\x^A x):(B -> B)` types do not match
- etc.

3.4 Evaluation criteria

The submission will be evaluated on the following criteria:

- Correctness of the program (hard criterium, 60%): is the program correctly implementing the assignment? Are there cases in which the program is implemented incorrectly?
- Readability of the program (soft criterium, 30%): is the program written to be understandable to humans too?
- Efficiency of the program (soft criterium, 10%): is program executing without noticable delay?

In the above text, the words must, should, and may have a special meaning. The assignment is graded with a passing grade if all features that must be implemented are correctly implemented. Higher grades are for submissions that also correctly implement features that should be implemented. Even higher grades are for submissions that also correctly implement features that may be implemented.

Assignment 4

Individual assignment

4.1 Introduction

In this assignment, you will write a critical literature review using ChatGPT (or any large language model that is similar to ChatGPT). As a scientist it is important to keep track of your sources: whatever goes in your mind influences you. For others to be able to understand what you know, it is necessary to show *where* you learned *what*. In this assignment you will test the knowledge of ChatGPT by asking it to explain concepts of programming languages. However, ChatGPT is not a scientist, and may produce inaccurate or even hallucinatory responses. Your task is to find out what is the truth, and where it comes from.

Hence, in this assignment you will review the answers of ChatGPT. Try to find credible sources to substantiate (or refute) the claims made by the chat bot. It is important to remain critical of the output of the tool: it may be the case that the chat bot gives incorrect, incomplete, or vague answers. However, it may also be the case that the bot gives correct, complete, and clear answers. Or it gives an answer that lies somewhere in between these extremes. Can you find out which of these is the case?

First, make a selection of concepts (some of which are listed below for your convenience) that spans the concepts as they were introduced during this course. For each selected concept you can formulate a question to submit to ChatGPT. For example, ask ChatGPT to explain a concept, or ask it to relate one or more concepts. Sometimes the quality of answers improves by asking it to produce an explanation that is understandable to computer science students, or by asking for a succinct but complete answer. You can also ask for clarifying examples. Document your dialogue with the chat bot (including your own prompts).

Next, investigate where the knowledge presented by the chat bot could come from, and what sources you can use to reflect on the accuracy of the response. Collect your sources in a bibliography. The only acceptable sources in this assignment are, on purpose, limited to published books and scientific (conference or journal) articles. You may gain access to some closed-access articles by using

the university library system.

The assignment results in an individually written paper using LaTeX and its PDF output is submitted in Brightspace. You could use the Overleaf tool for writing the paper, but any other LaTeX editor is fine too. The LaTeX document must use the standard *article* document class: do not change the page or font or margin size. Make sure to include your name and student number on the first page. For inserting references to sources, you could make use of BibTeX and the *plain* referencing format, and include a bibliography section that LaTeX/BibTeX can automatically produce.

4.2 Concepts

Here is an (incomplete) list of concepts: **formal language**, **translation**, **interpretation**, **compilation**, **grammar**, **context-free grammar**, **Backus-Naur Form**, **parse tree**, **abstract syntax tree**, **ambiguity**, **syntactic sugar**, **substitution**, **recursive-descent parsing**, **denotational semantics**, **operational semantics**, **referential transparency**, **compositionality**, **variables**, **variable occurrence**, **variable binding**, **scoping**, **variable capturing**, **shadowing**, **purity**, **side-effects**, **variable storage**, **aliasing**, **memory allocation**, **expressions**, **(short-circuit) evaluation**, **control structures**, **blocks**, **loops**, **breaking**, **returning**, **labeling**, **assignment statement**, **overloading**, **(type) coercions**, **(type) casting**, **(type) conversion**, **overflows**, **underflows**, **simple data types** (**Booleans**, **characters**, **integers**, **floats**), **complex data types** (**strings**, **arrays**, **slices**, **lists**, **maps**, **records**, **tuples**, **free unions**, **discriminated unions**), **pointers**, **references**, **procedures**, **functions**, **methods**, **lambdas**, **closures**, **continuations**, **abstract data types**, **inheritance**, **dynamic dispatch**, **subtyping**, **type systems**, **type checking**, **type inference**, **ad-hoc polymorphism**, **type parameters**, **generics**, **covariance**, **contravariance**, **exceptions**, **exception handling**, **events**, **event handling**, **components**, **composition over inheritance**, **concurrency**, **forking**, **shared memory**, **critical section**, **thread barrier**, **multiprocessing**, **atomicity**, **caching**, **nodes**, **channels**, **message passing**, **actors**, **predicates**, **constraint solving**, **unification**, **resolution**, **theorem proving**, et cetera.

4.3 Evaluation criteria

The paper submitted must be **between 4 and 10 pages long** (excluding bibliography): this requires you to make a selection of concepts to fit within the allowed page limit. One strategy might be to start with concepts you are most familiar with, so you can see whether ChatGPT's output makes any sense. Another strategy is to pick concepts that can be related in some way with one another. Maybe the best strategy is to choose concepts which you are least certain about!

The submission will be evaluated on the following criteria:

- Completeness and correct descriptions of concepts (soft criterium, 30%): are concepts covered in sufficient detail, or are essential details missing? Are concepts clearly and correctly described?
- Critical evaluation and use of sources (soft criterium, 30%): did you critically evaluate the claims and statements made by the chat bot? Are sources clearly indicated in your analysis of the response? Did you use proper sources? Is the bibliography formatted correctly?
- Readability (soft criterium, 40%): are there no spelling mistakes, no grammatical mistakes, and is the paper written to be understandable to your fellow students?