Programming Paradigms

Elective Module 2.4
University of Twente

Martijn Bakker
Martijn Bastiaan
Pim van den Broek
Marco Gerards
Arnd Hartmanns
Marieke Huisman
Sebastiaan Joosten
Jan Kuper
Arend Rensink

2018–2019

Contents

I Introduction **7**
I.1 Overview of the Module . . . 7
I.2 Learning Objectives . . . 8
I.3 Study Material and Software . . . 8
I.4 Lab exercises . . . 9
I.5 Assessment . . . 9
I.6 Fraud . . . 11
I.7 Sub-modules . . . 11
I.8 Teachers and Assistants . . . 12

1 Block 1 **13**
1-OV Overview . . . 13
1-OV.1 Contents of This Block . . . 13
1-OV.2 Mandatory Activities . . . 13
1-OV.3 Expected Self-Study and Project Work . . . 13
1-OV.4 Materials for this Block . . . 13
1-FP Functional Programming . . . 14
1-FP.1 First series: Introduction . . . 14
1-FP.2 Second series: Introduction and Higher-Order Functions . . . 17
1-CC Compiler Construction . . . 19
1-CC.1 EC Chapter 1: Overview of Compilation . . . 19
1-CC.2 EC Chapter 2: Scanners . . . 20
1-CC.3 Scanner implementation . . . 20
1-CC.4 ANTLR . . . 21
1-CC.5 EC Section 3.2: Expressing Syntax . . . 22

2 Block 2 **23**
2-OV Overview . . . 23
2-OV.1 Contents of This Block . . . 23
2-OV.2 Mandatory Presence . . . 23
2-OV.3 Expected Self-Study and Project Work . . . 23
2-OV.4 Materials for this Block . . . 23
2-CP Concurrent Programming . . . 24
2-FP Functional Programming . . . 27
2-FP.1 Third series: Types and Type Classes . . . 27
2-FP.2 Fourth series: Parsing in Haskell . . . 30
2-CC Compiler Construction . . . 32
2-CC.1 EC Section 3.3: Top-Down Parsing . . . 32
2-CC.2 Programming the LL(1) algorithm . . . 32
2-CC.3 ANTLR parser grammars . . . 34

3 Block 3 **37**
3-OV Overview . . . 37
3-OV.1 Contents of This Block . . . 37
3-OV.2 Expected Self-Study and Project Work . . . 37
3-OV.3 Materials for this Block . . . 37
3-CP Concurrent Programming . . . 38
3-FP Functional Programming . . . 43
3-FP.1 Fifth series: ParSec . . . 43
3-FP.2 Sixth series: Advanced Type Classes . . . 45
3-CC Compiler Construction . . . 48
3-CC.1 EC Chapter 4: Type inference and attribute grammars . . . 48
3-CC.2 Listeners and symbol tables . . . 50

4 Block 4 **53**
4-OV Overview . . . 53
4-OV.1 Contents of This Block . . . 53
4-OV.2 Expected Self-Study and Project Work . . . 53
4-OV.3 Materials for this Block . . . 54
4-CP Concurrent Programming . . . 54
4-FP Functional Programming . . . 56
4-FP.1 Seventh series: EDSLs and Code Generation . . . 56
4-CC Compiler Construction . . . 57
4-CC.1 Control flow and dependency graphs . . . 57
4-CC.2 ILOC . . . 59
4-LP Logic Programming . . . 61
4-LP.1 Royal Family: Facts, Rules and Queries . . . 61
4-LP.2 Monkey gets Banana: Functions, Backtracking, Lists . . . 61
4-LP.3 Tree Sort: More on lists, Data structures, Recursion . . . 62
4-LP.4 Ice Cream Tour . . . 64

5 Block 5 **67**
5-OV Overview . . . 67
5-OV.1 Contents of This Block . . . 67
5-OV.2 Expected Self-Study and Project Work . . . 67
5-OV.3 Materials for this Block . . . 67
5-CP Concurrent Programming . . . 68
5-CP.1 Program Dependencies . . . 68
5-CP.2 Compiler Directives for Parallellisation . . . 69
5-CP.3 General Purpose GPU programming (GPGPU) . . . 69
5-FP Functional Programming . . . 72
5-CC Compiler Construction . . . 72
5-CC.1 ANTLR tree visitors . . . 72
5-CC.2 Dealing with procedures . . . 75
5-LP Logic Programming . . . 75

6 Block 6 **77**
6-OV Overview . . . 77
6-OV.1 Contents of This Block . . . 77
6-OV.2 Expected Self-Study and Project Work . . . 77
6-OV.3 Materials for this Block . . . 77
6-CP Concurrent Programming . . . 78
6-CP.1 Rust: Safe Shared Memory Concurrency and Message Passing Concurrency . . . 78
6-CP.2 Software Transactional Memory . . . 78
6-FP Functional Programming . . . 79
6-CC Compiler Construction . . . 79
6-LP Logic Programming . . . 80

7 Block 7 **81**
7-OV Overview . . . 81
7-OV.1 Contents of This Block . . . 81
7-OV.2 Mandatory Presence . . . 81
7-OV.3 Expected Self-Study and Project Work . . . 81
7-OV.4 Materials for this Block . . . 81
7-CP Concurrent Programming . . . 82

A Functional Programming **85**
A-1 GHC installation . . . 85
A-2 QuickCheck . . . 85
A-3 IO in Haskell . . . 90
A-3.1 IO Actions . . . 90
A-4 IO Functor . . . 91
A-5 Sequencing IO . . . 92
A-6 Monoids in GHC 8.4 . . . 93
A-7 Some standard operators and functions . . . 93
A-8 Some important type classes . . . 95
A-9 Literature . . . 97

B ILOC support **99**
B-1 Memory layout and stack . . . 99
B-2 Additional naming conventions: Label references and strings . . . 100
B-3 Additional instructions . . . 100
B-4 Initialisation of constants . . . 101
B-5 Tooling . . . 101

C PP Final Project **103**
C-1 Source language . . . 103
C-1.1 Data types . . . 104
C-1.2 Simple expressions and variables (mandatory) . . . 105
C-1.3 Basic statements: Assignments, if and while (mandatory) . . . 105
C-1.4 Concurrency (mandatory) . . . 105
C-1.5 Division (optional) . . . 106
C-1.6 Procedures/functions (optional) . . . 106
C-1.7 Exception handling (optional) . . . 106
C-1.8 Optimisations (optional) . . . 107
C-2 The compilation process . . . 107
C-2.1 JAVA front-end and code generation . . . 107
C-2.2 JAVA front-end, HASKELL code generation . . . 107
C-2.3 HASKELL front-end and code generation . . . 107
C-2.4 SPROCKELL extensions . . . 108
C-3 Testing . . . 108
C-3.1 Testing for syntax errors . . . 108
C-3.2 Testing for contextual errors . . . 108
C-3.3 Testing for semantic errors . . . 109
C-3.4 Automatic versus manual testing . . . 109
C-4 The final product . . . 109
C-4.1 Software . . . 109
C-4.2 Report . . . 110
C-5 Assessment . . . 111
C-5.1 Assessment of the language features. . . . 111
C-5.2 Use of HASKELL . . . 112
C-5.3 Quality of the final product . . . 112
C-5.4 Example scenarios . . . 113
C-5.5 Feel like a challenge? . . . 113

D FP Project **115**
D-1 μFP . . . 115
D-2 Submission and grading . . . 116
D-2.1 Rules . . . 116
D-2.2 Assessment . . . 117
D-2.3 Suggested and mandatory features . . . 117
D-2.4 Available code and submission . . . 117
D-3 Parser combinator library . . . 119
D-4 Basic Parsers . . . 120
D-5 Embedded Domain Specific Language . . . 121
D-6 Parser . . . 122
D-7 Further features . . . 122

Chapter I

Introduction

This is the reader for elective module "Programming Paradigms" (M2.4) of the BSc curriculum for Technical Computer Science (B-TCS). The reader contains information about the organisation of the module and (especially) the exercises and assignments to be done.

I.1 Overview of the Module

Note: The text in this reader is primarily directed at B-TCS-students who follow the module in the course of their regular curriculum. If you follow (part of) the module in the context of your minor, pre-master, exchange programme or otherwise, some of the text may not or only partially apply to you.

Aims and setup The overall aim of this module is to acquaint you with a broader perspective on the ways in which one can program a computer system. Up until this point, you have mainly learned imperative, object-oriented programming. This module will extend your awareness and ability in the following directions:

- Non-imperative programming: in particular, the functional paradigm (extensively) and the logic paradigm (briefly).
- Programming concurrent systems: concepts, data structures and algorithms for various concurrent computing paradigms
- Programming language technology: parsing, compilation and code generation

These three strands are initially taught separately and come together in the module project, where you develop a compiler for your own language with support for parallellism, generating code that runs on an extensible hardware emulator written in a functional language.

Position in the Curriculum This module is offered in the fourth period (Quarter 2B) of the second year of B-TCS. It is an elective module, which can also be followed by students from other curricula. However, a prerequisite for the module is a thorough knowledge of and experience with the programming language JAVA.

Strands In this reader, will use the following abbreviations for the strands of this module:

- **FP**: Functional Programming;

- **LP**: Logic Programming;
- **CP**: Concurrent Programming;
- **CC**: Compiler Construction.

Block-based structure Because of the large number of bank holidays during Quarter 2B, the module is spread over 11 calendar weeks, but contains only 50 working days — effectively 10 weeks. For this reason, rather than going by calendar weeks, the module is structured in 10 blocks of 5 consecutive working days. Consequently, the material in this reader is also divided over 10 blocks. The Canvas site shows the mapping of weeks to blocks.

I.2 Learning Objectives

After successful completion of this module, you will be able to:

- Describe the programming paradigms covered in the module (FP, LP and CP) and their essential characteristics and differences
- Write basic programs in each of these three programming paradigms
- Solve non-trivial programming problems in FP and CP
- Explain the concepts and importance of typing, in terms of FP and CC
- Explain and use the common types and data structures in FP and CP
- Explain and take advantage of the evaluation and execution mechanisms of FP (lazy evaluation) and CP (hardware-related aspects, concurrency models)
- Explain and use the following concepts:
  - From FP: recursion, list comprehension, higher order functions, parameter accumulation, function composition, lazy evaluation.
  - From CP: interleaving, fairness, deadlock, memory models, synchronisation, locking.
  - From CC: syntactic and semantic analysis, scanning, parsing, run-time organisation, code generation, optimisation.
- Write a compiler for a non-trivial imperative language with concurrency features, generating a given (dedicated) instruction set.

I.3 Study Material and Software

Books For the CC and CP strands, we have selected the following two books:

- **CC**: *Engineering a Compiler, Second Edition*, Keith D. Cooper & Londa Torczon, Morgan Kaufmann (2011) (EC)
- **CP**: *Concurrent Programming and Concurrent and Distributed Programming: Java Concurrency in Practice*, Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes and Doug Lea, Addison Wesley (2006) (JCIP)

For the FP strand no book is required. The following books are advised as reference material:

- *Learn You a Haskell for Great Good!*, Miran Lipovaca, No Starch Press, 2011. `http://learnyouahaskell.com/` (LYH)
- *Programming in Haskell*, Graham Hutton, Cambridge University Press, 2016. (PH)

B

Canvas Additional material for CP, FP and LP will be made available via Canvas.

Software

- **CC**: ANTLR (ANother Tool for Language Recognition), version 4, available at `http://www.antlr.org`. Plugins are available for ECLIPSE, INTELLIJ.
- **FP**: HASKELL, available at `https://haskell.org`
- **LP**: SWI-PROLOG, available at `http://www.swi-prolog.org`

Reader This reader contains all projects and exercises for this module, as well as study material for the FP strand (Appendix A), and for the CP strand (Block 5).

Recorded lectures. Lectures will not be recorded. However, last year's lecture will be made available via Canvas. Though it is always recommended to visit the actual lecture, in this way if you accidentally miss a lecture you can get an impression of what was discussed. However, we do not provide any guarantees that the material discussed during this year's lecture will be exactly the same as the material discussed during last year's lecture.

I.4 Lab exercises

All strands have laboratory exercises. The exercises are contained in this reader, and ordered by block. The exercises should be done in pairs; in special cases, working alone is allowed, but doing the exercises in groups larger than 2 is not. For FP, LP and CC all lab exercises have to be signed off by teaching assistants. For CP the exercises marked with a ☞ symbol have to be signed off by teaching assistants.

Signing off will be done by an assistant during the scheduled lab hours. The assistant may ask further questions about your solution. Both members of the group have to be present during sign-off — otherwise, only the student who is present will be signed off.

Intended sign-off deadline. The basic principle is that you should be able to finish the exercises at the end of the block, within the time scheduled for that block. To achieve this, you have to actively participate from the start of the module, and make sure you do the lab exercises in time.

Presence during the scheduled laboratory sessions is not mandatory, but if you fail to meet the expected deadline while also skipping sessions, less leniency will be shown to you when it comes to continued participation in the module.

Hard sign-off deadline. There may be circumstances that make it impossible for you to meet the above-mentioned intended deadline. For that reason we impose a second, *hard* deadline that offers a bit more leeway. If your lab exercises of block x are not signed off on day 3 of the *next block* $x+1$, you may be denied participation in the rest of the module.

Exceptions can be made on an individual basis. Of course (as usual) if you are forced to be absent for a longer period, due to illness or other circumstances, individual arrangements can be made to sign off lab exercises at a later date. If you feel that this applies to you, please contact the module coordinator.

I.5 Assessment

You will be assessed on the basis of five grades, summarised in Table I.1.

- The column "I/G" indicates if this is an Individual or Group test. A Group, in this module, always consists of two persons collaborating on a project. As is apparent, the final grade consists for 55 % of Individual grades, and for the other 45 % of Group grades.
- The grade for the FP/LP project is the weighted average of the grade for the FP project assignment (75 %) and the LP project assignment (25 %). There is no minimum grade for these individual projects.

Table I.1: Assessment of the Programming Paradigms module

Test	Kind	I/G	Weight	Min.	Block:Day
FP test	written test	I	15%	5.0 [1]	7:5 (resit: 10:1)
LP/FP projects	project	G	15%	5.5 [2]	6:1/7:1
CP test	written test	I	20%	5.0 [1]	8:5 (resit: 10:3)
CC test	take-home tests	I	20%	5.5 [3]	4:6 & 8:1
Integrating project	project	G	30%	5.5	10:5

[1] In addition, the average of the FP and CP tests must be at least 5.5.
[2] Calculated as the weighted average of an LP project (25 %) and an FP project (75 %).
[3] Calculated as the average of two separate take-home tests.

- The grade for the CC test is the average of the grades for the two take-home tests. There is no minimum grade for each individual take-home test.
- The final column lists the block(s) and the day in which the tests are scheduled, or the homework respectively project is due. All written tests have a scheduled resit in the final block. On Canvas there is an overview showing how the block:day numbers translate into concrete dates.

The final grade is the weighted average of the separate grades, with weights as listed in the "Weight" column. On top of this, a sufficient grade can *only* be obtained if both of the following conditions hold:

- the separate grades are larger or equal to the respective minimum grades, listed in the "Min" column, and
- the average of the two written tests (FP and CP) is at least a 5.5.

Diagnostic Test (CP) During the course, there is a Diagnostic test, which will be announced on Canvas. This test is mandatory, but not graded. If you are unable to participate, contact the teacher of CP as soon as possible.

Laboratory exercises (FP, LP, CC, CP) Though the laboratory exercises are *not* part of the assessment for the final grade, they are mandatory and have to be finished in time; see §I.4.

Project and take-home deadlines (FP, LP, CC) If you are not ready with a project or take-home test on the day of the deadline, you can still submit after the deadline. Every day that you submit after the initial deadline (starting one minute after midnight) reduces the grade for the project or take-home test by 1.0 point. This means in particular that if you submit 4 days after the deadline, your maximum grade will be a 6.0. *Note: we count calendar days here,* ***not*** *working days*.

Written tests (FP, CP) Example tests will be made available on Canvas. Both tests are open book, meaning you are allowed to bring:

- a printed version of this reader;
- a printed copy of the lecture slides; and
- a paper version of the book (in case of FP: you may bring one of the recommended books).

You are *not* allowed to bring:

- your own material (print-outs of laboratory exercises, or notes in whatever form).

It is allowed to have text highlighted with a text marker, but not to have any hand-written notes in the reader, slides or book.

I.6 Fraud

The take-home exam for Compiler Construction is to be done individually, despite the fact that it partially builds onto laboratory exercises that you have done in pairs. If we detect that you have developed your solution with, or copied your solution from another student, this is considered fraud, as you are then submitting someone else's work as your own. Yet we do not want to discourage you from helping each other out or discussing solution strategies.

Obviously there is a line between "helping out" and "sharing solutions," but where does it lie? Here are some scenarios that can help you judge what you may and should not do.

- **Scenario.** Ibrahim is quite good at Compiler Construction, as his lab partner Inge knows by now. She is having trouble with one of the homework questions and asks Ibrahim to take a look at her partial solution. He does so, and points out some errors that is making ANTLR misbehave. Inge repairs them based on his suggestion, after which everything is dandy. *Is this allowed or not?*

  Verdict. Yes, this is fine, as long as it is Inge making the repairs, and not Ibrahim. The idea is that this will still allow Inge to learn the material.

- **Scenario.** Josh and Dave discuss the questions and agree that one of them is really very similar to something they did together in the lab. They both take the code they developed for that lab exercise, make essentially the same modifications and submit them separately. *Is this allowed or not?*

  Verdict. Yes, this is also fine. It will in practice never be the case the modifications are so minor that Josh and Dave do them in exactly the same way, so the risk that their answers will be (incorrectly) flagged as being fraudulently shared is negligable.

- **Scenario.** Kim allows Pim to look at her solution to a question, because he does not know where to start. He then goes off to answer it on his own, but he gets stuck. Because the deadline is near, he asks Kim if he can look at her solution once more; she sends her files to him because she's already gone home. Pim now sees what he did wrong and copies chunks of Kim's solution to repair his answer. *Is this allowed or not?*

  Verdict. This became a case of fraud when source files were sent per email. Both Kim and Pim are at fault: Kim for enabling fraud by sending her files (even if it was meant as a friendly gesture) and Pim for partially copying them.

If these scenario's are not sufficient to take away all your doubts, you can always contact the module coordinator about your specific case.

I.7 Sub-modules

Students following a different curriculum than B-TCS may follow parts of the Programming Paradigms module as part of their own programme. The following sub-modules are available:

Functional and Logic Programming (FLP)
Credits: 5
Assessment: Ordinary average of
- FP test (min. grade 5.5 — see remark below)
- FP/LP project (min. weighted avg. grade 5.5)

Concurrent Programming (CP)
Credits: 3
Assessment: CP test (min. grade 5.5 — see remark below)

Compiler construction (CC)
Credits: 5
Assessment: Ordinary average of
- CC homework (min. avg. grade 5.5)
- a reduced integrating project (min. grade 5.5)

Note that, because there is only at most one test, for some of these sub-modules the required grades are *different* than in Table I.1. All other applicable conditions and rules of the module are inherited by the sub-modules.

I.8 Teachers and Assistants

The contact details of the instructors and teaching assistants and their individual responsibilities can be found on the Canvas site of this module.

For questions about the contents, the exercises, or the project, you can always ask one of the teaching assistants at the next lab session; or if that does not result in a satisfactory answer, contact either the lecturer responsible for the strand or the module coordinator, either by email or during the lecture.

For organisational matters, you can approach the lecturers or coordinator directly.

Block 1

Block 1

1-OV Overview

1-OV.1 Contents of This Block

FP This block will introduce some fundamental concepts and demonstrate some elementary examples of functional programs. Some of these concepts are: function, definition, currying, recursion, types, polymorphism, higher order functions, pattern matching, lambda abstraction and list comprehension.

CC This block will introduce the architecture of a compiler, and how to separate a piece of text into words and symbols (scanning). You will also meet ANTLR, the tool used in this course to automatically generate scanners and parsers.

1-OV.2 Mandatory Activities

The sign-off exercises of this block should be completed (and signed off) in the course of the block itself. If you are not finished, make sure that your solutions are ready to be signed off during the first lab session of block 2.

1-OV.3 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 4 hours self-study for the CC material
- 4 hours self-study for the FP material

1-OV.4 Materials for this Block

- **CC**, from EC: Chapters 1 (completely) and 2 (except for 2.4.1–2.4.4 and 2.6).
- **FP** reference material:
  - From LYH: Chapters 1–5 (lecture 1), Chapter 6 (lecture 2), Chapter 8 (lecture 3)
  - From PH: Chapters 1–6 (lecture 1), Chapter 7 (lecture 2), Chapter 8 (lecture 3)

1-FP Functional Programming

For Functional Programming, we use version 8.0.2a of Haskell platform, since more recent versions have incompatibility issues (see Appendix A-1). Please make sure it is installed before the first lab session by precisely following the instructions in the appendix.

Although you may find solutions to some problems online or in another source, you are not allowed to copy them and/or present them as your own (i.e., fraud). If you do this we cannot check if you are able to solve problems by yourself, and most likely you are ill-prepared for the exam. Exercises that are not written by you, or that you cannot fully explain, are not signed off.

1-FP.1 First series: Introduction

Exercise 1-FP.1 Introduction exercise. Type in the next function

$$f\,x \quad = \quad 2x^2 + 3x - 5$$

and evaluate it for a few values of x. □

Remark. From now on every function definition should be preceded by mentioning its type.

Exercise 1-FP.2 This exercise uses and introduces *QuickCheck*. Make sure QuickCheck is installed.

During the lecture the following two alternative function definitions for the sum from 0 to n were discussed:

```
total1 :: Int -> Int
total1 0 = 0
total1 n = total1 (n-1) + n

total2 :: Int -> Int
total2 n = (n * (n+1)) `div` 2
```

First, verify by hand that these functions give the same result for various positive numbers.

Second, import the module `Test.QuickCheck` and add a proposition that tests whether `total1` and `total2` are equivalent, by adding the following to your .hs-file:

```
import Test.QuickCheck
prop_total n =   (n >= 0)  ==> total1 n == total2 n
```

Reload your `.hs`-file in GHCi and run the test with: `quickCheck prop_total`

Finally, modify `total2` such that the QuickCheck test *fails*. □

Exercise 1-FP.3 An operator $*$ is commutative when `x * y == y * x`. Define two QuickCheck propositions of type `Int -> Int -> Bool` to (randomly) test the

- commutivity of the addition operator $(+)$ - this test should succeed,
- commutivity of subtraction operator $(-)$ - this test should fail.

Test the propositions using QuickCheck. Is the result of the test as you expected? □

Remark. Use QuickCheck to verify your solutions to speed up testing. Student assistents may ask you to define tests when *they* see the need (e.g., doubt about validity of your definitions).

Exercise 1-FP.4 Import the module `Data.Char`:

```
import Data.Char
```

Now the following functions are available:

```
ord :: Char -> Int
chr :: Int -> Char
```

These functions translate a character into its code and back.

- Use **ord** and **chr** to define a function `code` that changes a letter (including capital letters) by cyclicly shifting it three positions further in the alphabet, i.e., after `'z'` one should continue with `'a'` again.

  For example:

  ```
  code 'a' = 'd'
  code 'P' = 'S'
  code 'y' = 'b'
  ```

  The function `code` should leave all other characters (digits, spaces, etc.) unchanged. Hint: the relations `<`, `<=`, **`=>`** and `>` also work for characters.

- Evaluate the expressions

  ```
  map code "hello"
  map code "Tomorrow␣evening,␣8␣o'clock␣in␣Amsterdam"
  ```

  (**map** applies the function `code` to all characters in a string).

- Define a new function that generalises your function *code* such that it receives a number n as additional argument to indicates how many positions the character has to be shifted (note that above we have `n=3`). Note that the order in which the arguments are given to *code* is relevant for using the function *map* with the generalised coding function.
- Use QuickCheck to test if *code* and the generalised version for $n = 3$ give the same result.
- Use QuickCheck to verify that first shifting by n positions, and subsequently by shifting $26 - n$ positions results in the original character.

□

Please ask a TA to check Exercise-1.FP.1 to Exercise-1.FP.4

Exercise 1-FP.5 Define *recursively* a function *interest* that calculates how much money you have after n years, if you start with an amount a and receive r percent of interest per year. You have to take into account that you will have "interest over interest", where the interest only has to be computed once per year. □

Exercise 1-FP.6 Define two functions *root1* and *root2* that determine the roots of a quadratic equation of the form

$$ax^2 + bx + c = 0$$

where a, b, c are given, and $a \neq 0$. If the discriminant is negative, your function should give an error, to be defined as follows:

```
error "negative␣discriminant"
```

Write a function `discr` that calculates the discriminant. This function can be used in the functions `root1` and `root2`.

Test your functions for a number of values of a, b, c. □

Exercise 1-FP.7 A second order polynome is an expression of the form

$$ax^2 + bx + c$$

(assume $a \neq 0$).

1. Write a function `extrX` that calculates the value of x where the polynome has its extreme value.
2. Write a function `extrY` that calculates this extreme value.

□

Exercise 1-FP.8 Write *recursive* definitions for the following functions on lists (give the type for every function you define):

1. `mylength` for the length of a list,
2. `mysum` that adds the elements in a list of numbers,
3. `myreverse` that reverses the order of the elements in a list,
4. `mytake` that gives the first `n` elements of a list (when `n` is greater than the length of a list, the whole list should be delivered),
5. `myelem` that determines whether a given element is in a list,
6. `myconcat` that glues together a list of lists into one long list,
7. `mymaximum` that yields the maximum of a list of numbers,
8. `myzip` that transforms two lists into a list of pairs (the shortest of the two lists determines the length of the resulting list).

Hint: you can use QuickCheck to test your functions by comparing them with the built-in functions, e.g.,

```
prop_mytake n xs = n >= 0 ==> mytake n xs == take n xs

prop_mymaximum xs = not (null xs) ==> mymaximum xs == maximum xs
```

□

Please ask a TA to check Exercise-1.FP.5 to Exercise-1.FP.8

Exercise 1-FP.9 A sequence of numbers is *arithmetic* if, starting from some initial number a, every next number in the sequence can be calculated from the previous number by adding a fixed difference d.

1. Write a recursive function `r` that generates the arithmetic sequence starting with `a`, and using difference `d`. Note that the type of `r` is:

   ```
   r :: Num a => a -> a -> [a]
   ```

2. Write a function `r1` that selects the `n`-th number from the sequence above (hint: use the function `r`)
3. Let `i` and `j` be two indices. Write a function `totalr` that calculates the sum of the `i`-th element upto (and included) the `j`-th element. □

Exercise 1-FP.10
1. Write a function `allEqual` which determines whether all elements in a list are equal.
2. Write a function `isAS` which checks whether a sequence is arithmetical (hint: use the function `allEqual`). □

Exercise 1-FP.11 A matrix can be defined as a list of lists of numbers, where the inner lists are the rows of the matrix.

1. Write a function `allRowsEquallyLong` that checks whether all rows in a matrix are equally long
2. Write a function `rowTotals` that yields the list of totals of every row in a matrix,
3. Write a function `mytranspose` that transposes a matrix, i.e., every n-th row is transformed into the n-th column,
4. Write a function `colTotals` that yields the list of totals of every column in a matrix. □

Please ask a TA to check Exercise-1.FP.9 to Exercise-1.FP.11

1-FP.2 Second series: Introduction and Higher-Order Functions

Exercise 1-FP.12 Define the following functions, including their types. They should work the same as the corresponding originals in Haskell.

1. `myfilter`,
2. `myfoldl`,
3. `myfoldr`,
4. `myzipWith`. □

Exercise 1-FP.13 Persons are registered in a database by their: name, age, sex, place of residence. Assume that a person is uniquely determined by his/her name.

1. Give the type `Person` of such a database, in which the data of each person are stored as a four-tuple (use type synonyms). Create an example database of this type to test the functions you have to write below.
2. Define functions to extract the separate data of one person from a four-tuple.
3. Write in *three* ways — with recursion, with list comprehension, and with higher order functions — a function to increase the age of all persons with n years.
4. Write — again in three ways — a function that yields the names of all women between 30 and 40 years.
5. Write a function — one way is sufficient — that yields the age of a person who is given by his/her name. It should be possible that the name contains small letters and/or capital letters (**`import`** `Data.`**`Char`** for the functions **`toLower`** and **`toUpper`**).
6. Sort the database by age. You may not use predefined Haskell functions such as `sortWith`, only **`sort`** is allowed (from the module `Data.`**`List`**). Note that **`sort`** also works for tuples. □

Exercise 1-FP.14 1. The *Sieve of Erathostenes* is a technique to generate the list of all prime numbers. It works as follows: it starts with the list of all natural numbers, starting with 2. Take the first number and remove from the rest of the list all multiples of this first number. Repeat this procedure for the first number from the remaining list, etcetera.
Write a function *sieve* that implements the Sieve of Erathostenes.
Next, write some functions that use the function `sieve`:
   (a) a function that tests whether a given natural number is a prime number,
   (b) a function that delivers the first n prime numbers,
   (c) a function that yields all prime numbers smaller than n.

   Remark. Note that these functions use infinite lists and lazy evaluation.
2. Define a function `dividers` that yields the list of all dividers of a given natural number m. Using this function, define an alternative function to determine whether a given number is prime. □

Exercise 1-FP.15 A three-tuple (a,b,c) is a *Pythagorean Triple* if $a^2+b^2=c^2$. Well known examples of such triples are (3,4,5) and (5,12,13).

1. Write a function *pyth* that generates all Pythagorean Triples (a,b,c) such that all a, b, c are smaller than a given natural number n.

□

Please ask a TA to check Exercise-1.FP.12 to Exercise-1.FP.15

Exercise 1-FP.16 1. A list of numbers is *increasing* if every next number in the list greater is than the previous number. Write a function `increasing` which checks whether a list is increasing.
2. A list is *weakly increasing* if every next number in the list is greater than the mean of all previous numbers. Write a function `weaklyIncreasing` which checks whether a list is weakly increasing. Hint: define a helper function that implements the same functionality, but also receives the list of all previous numbers. □

For the next exercise, we start with some definitions:

- A list `xs` is called a *sublist* of a list `ys` if all elements of `xs` occur *consecutively* in `ys`.
- A list `xs` is called a *partial* sublist if all members of `xs` occur in `ys` in the same order as they occur in `xs`, but not necessarily consecutive.

For example, the list $[2,4,6]$ is a sublist of the list $[0,2,4,6,8]$ and a partial sublist of the list $[1,2,3,4,5,6,7]$.

Note that every sublist is a partial sublist as well, but the opposite need not be the case.

Exercise 1-FP.17 Using the definitions above, solve the following.

1. Write a function `sublist` that checks whether a list `xs` is a sublist of a list `ys`,
2. Ibid for partial sublist. □

Exercise 1-FP.18 In this exercise you have to write five different sorting algorithms:

1. The *bubble sort* algorithm sorts a list by repeatedly going through the list, while on the way comparing and swapping two consectuive elements if they are in the wrong order. After one pass through the list, the largest element will be "bubbled" to the end.
   In the next pass through the list, this last element may be ignored, et cetera. The list is fully sorted if the remaining list to be bubbled does not contain more than one element anymore.
   Write a function `bsort` that implements this process, using a funtion `bubble` which implements a single pass through the list.
   Define a QuickCheck property that checks if `bsort` gives the same result as the function **`sort`** supplied with GHC. Make sure to give the property is of type: `[Int] -> Bool`, and check what may go wrong if no type is given (but derived). This convenient way of verifying your new implementation against a reference implementation can also be used in the exercises below.
   Possible optimization (not compulsory): sorting is ready when no swapping of two elements is necessary anymore.
2. The *min-max* algorithm takes the minimum and the maximum of a list and puts them in front and at the end (respectively). Repeating this process for the list from which the minimum and the maximum are removed, will sort the list.
   Write a function `mmsort` that implements this algorithm.
   Hint: use the function \\ from the module `Data.List`.
3. The algorithm *insertion sort* builds up the sorted list as a separate list next to the list that has to be sorted. Initially this eparate list is empty, and the sorting process is performed by inserting the elements from the original list one by one into this separate list.
   Write a function `isort` that implements this algorithm by first writing a function `ins` which inserts one element into an already sorted list and by exploiting **`foldl`** or **`foldr`**.
   Remark: you may not use the already existing function **`insert`** from `Data.List`).
4. The *merge sort* algorithm splits a list in two halves, sorts these two halves, and merges them together again. Sorting both halves should be done in the same way, i.e., recursively.
   First write a function `merge` that merges two already sorted lists into a sorted list, and then write a function `msort` that implements the merge sort algorithm, using the function `merge`.
5. The *quick sort* algorithm splits a list into two lists: one contains the elements that are smaller than or equal to the head of the list, the other contains the elements that are bigger than the head of the original list. Repeating this process to both resulting lists and concatenating them together again will sort the original list.
   Write the function `qsort`. □

Please ask a TA to check Exercise-1.FP.16 to Exercise-1.FP.18

Exercise 1-FP.19 The function `myflip` receives a function that receives two arguments, and transforms it to a new function that receives its arguments in reversed order. This means it receives a function of type `a -> b -> c` and transforms it into a function of type `b -> a -> c`. For example (the parentheses may be omitted): `(myflip take) "abc" 2 == "ab"`

Use a lambda expression to define the function `myflip`. Explain to the teaching assistant why using a lambda expression is preferred here. □

Exercise 1-FP.20 Use both the *point-free style* and *function composition* to define the function `transform :: String -> String` that:

- Reverses the string,
- Removes all nonalphanumerical character,
- Converts all alphanumerical characters to upper case.

Hint: consider the functions **`toUpper`** and `isLetter`.

Verify: `transform "Hello, world!" == "DLROWOLLEH"` □

Please ask a TA to check 1-FP.19 to 1-FP.20

1-CC Compiler Construction

1-CC.1 EC Chapter 1: Overview of Compilation

Exercise 1-CC.1 Compile for yourself a list of the following terms, with short definitions for each of them. Don't just copy from the book: make sure you understand your own definitions.

back end, front end, grammar, instruction scheduling, instruction selection, optimizer, parsing, register allocation, scanning, type checking

□

Exercise 1-CC.2 Answer the following questions.

1. Using the grammar on Page 12 of EC, explain systematically, i.e., step by step, why the following string is an instance of the syntactic variable *Sentence*:
   "Most students is good programmers."
2. What is actually wrong with the above sentence? To what kind of programming error can you compare this? At what stage would a compiler detect it?
3. Which steps in the two sub-exercises above correspond to the following activities: *parsing*, *type checking*, *scanning*?

□

Exercise 1-CC.3 Extend the grammar on Page 12 of EC so that it can also cope with sentences of the form

"Programming Paradigms is a diverse interesting module."
"Programming Paradigms is a diverse interesting elective module."

without needing a new rule for every case where there are more successive adjectives. (A word like "a" is called a *particle* in English.) □

Exercise 1-CC.4 Consider the assignment

$$d \leftarrow d + 2 \times (a + b)$$

1. Manually carry out the (naive) instruction selection process informally described in §1.3.3 of the book (see Fig. 1.3) on this assignment.
2. Improve the sequence of instructions by minimizing the number of required registers
3. Improve your answer to the previous subquestion by rescheduling the sequence to a minimal execution time (where you may use more registers if that benefits the execution time). Compute the number of clock cycles of the original and the resulting schedule.

Give your answers in the form of tables such as the ones on Pages 17–19. □

1-CC.2 EC Chapter 2: Scanners

Exercise 1-CC.5 Answer Review Question 2 of Section 2.3 (Page 42). Look for a concise RE; you may make use of all the notation introduced in the section. (The quotation mark, `"`, is an element of Σ.) □

Exercise 1-CC.6 Consider a language in with three different token kinds: (i) "`La`", (ii) "`La␣La`" and (iii) "`La␣La␣La␣Li`" (where the symbol ␣ denotes a space), with the proviso that

- Capitalisation must be as given: all `L`'s are upper case and all `a`'s and `i`'s lower case;
- The `a`'s may be repeated arbitrarily often (meaning there can be one or more), but the `i` may not; so `Laaaaaa` is allowed but not `Lii`;
- Every `La` and `Li` (also those that are part of the token kinds (ii) and (iii)) may be followed by zero or more spaces, *which are considered to be part of the token*.

Hence, for instance, "`LaaaaLaLaa␣Laaaa␣␣␣␣LaLiLaa`" is a valid input text; it consists of tokens "`LaaaaLa`", "`Laa␣Laaaa␣␣␣␣LaLi`" and "`Laa`".

1. Give regular expressions for the three token kinds of this language.
2. Give a single DFA that is able to recognise all three token kinds.
3. Answer the following questions, considering that scanning is *greedy*:
   - What is "`Laaaa LaLaa`" broken down into: "`Laaaa␣`" + "`LaLaa`", "`Laaaa␣La`" + "`Laa`" or "`Laaaa␣`" + "`La`" + "`Laa`"?
   - What is "`La␣La␣La␣La␣Li`" broken down into? □

1-CC.3 Scanner implementation

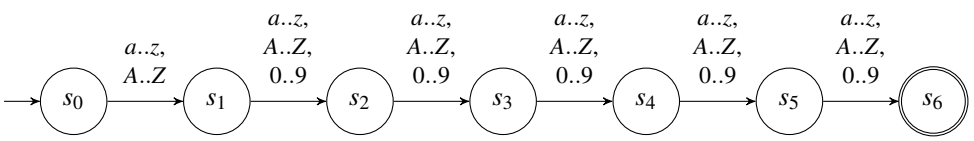
Consider the following files, to be found on BLACKBOARD:

- `pp.block1.cc.dfa.State`. This is a straightforward implementation of a DFA — more precisely, a *state* of a DFA, but since all states are reachable from the initial state, it suffices to represent a DFA by its initial state.
- `pp.block1.cc.dfa.Checker`. This is an interface offering the functionality of testing whether a given DFA accepts a given input text.
- `pp.block1.cc.test.CheckerTest`. This is a JUNIT test for an implementation of `Checker`.
- `pp.block1.cc.dfa.Scanner` This is an interface offering the functionality of applying a given DFA as a scanner to an input text.
- `pp.block1.cc.test.ScannerTest`. This is a JUNIT test for an implementation of `Scanner`.

Exercise 1-CC.7 Program an *efficient* implementation of `Checker`, and show it correct using `CheckerTest`. *Efficient* means that the execution time of your algorithm is *linear* in the length of the input string. You should be ready to argue why your solution is efficient. □

Exercise 1-CC.8 Program an *efficient* implementation of `Scanner`, and show it correct using `ScannerTest`. Efficiency here means the same as in Exercise 1-CC.7. Make sure that your solution correctly implements the notion of greediness, and that you are ready to argue why it is efficient. (*Hint:* reusing the `Checker` implementation of Exercise 1-CC.7 will *not* give rise to an efficient solution!) □

Exercise 1-CC.9 The following finite automaton accepts *identifiers* consisting of an alphabetic character followed by exactly five alphanumeric characters. (See also Review Question 1 of Section 2.2 (Page 33).)



The pre-defined class `State` includes a constant `DFA_ID6` that encodes this automaton, and `CheckerTest` and `ScannerTest` contain test methods for this automaton.

Add a constant `DFA_LALA` to `State`, analogous to `DFA_ID6`, that implements the DFA you developed in Exercise 1-CC.6. Add tests for this DFA to `CheckerTest` and `ScannerTest` that show the correctness of your DFA. □

1-CC.4 ANTLR

A grammar in ANTLR is defined in a file with extension `.g4`. In this course we will combine grammar files with JAVA source files. The ANTLR tool functionality consists of generating JAVA files from grammars that, when properly invoked, do the job of scanning, and later on, parsing and code generation.

This text assumes that you use ANTLR through its ECLIPSE plugin.

The lab files include the following example grammar:

```
1 lexer grammar Example;
2
3 @header{package pp.block1.cc.antlr;}
4
5 WHILE : 'while';              // Keyword
6 DO    : 'do';                 // Keyword
7 WS    : [ \t\r\n]+ -> skip ; // At least one whitespace char; don't make token
```

Here is an explanation of the most prominent features:

- Line 1: This declares the grammar. The name has to equal the file name (minus extension and not including any namespace information). The optional keyword **`lexer`** (which is used in the Compiler Construction world as a synonym for "scanner") means that this grammar only supports scanner rules; the default (obtained by leaving out the keyword) combines scanner and parser rules.

- Line 3: This specifies a line that should be inserted at the top of the Java files generated from the grammar. Since we want the generated files to be within the Java package `pp.block1.cc.antlr`, we need a **`package`** declaration in the Java file; that is what this line achieves.

- Lines 5 and 6: This specifies that `while` and `do` are tokens of our `Example` language. `WHILE` and `DO` are the identifiers chosen for these tokens by the grammar designers; the fact that they equal the keywords is coincidental. Right now the names are themselves no used anywhere in the grammar.

- Line 7: This specifies that any non-empty sequence of characters from the set ' ' (space), '`\t`' (the TAB character), '`\r`' (the CR or Carriage Return character) and '`\n`' (the NL or Newline character) is considered a token; hoewever, the directive `->` **`skip`** then specifies that this token may actually be discarded. This construction is typically used for any information in the input file that does not need to be passed on for further processing: whitespace between other tokens (as here) or comments in the input file (in whatever comment syntax the input language supports).

Exercise 1-CC.10 To get acquainted with the ANTLR tool, carry out the following steps:

1. Generate the actual scanner `pp.block1.cc.antlr.Example` (a JAVA-file), and observe that, after this, all pre-defined lab files compile correctly. In the ECLIPSE plugin, this can be done by right-clicking the `g4`-file and selecting "Run As... → Generate Antlr Recognizer".
2. Study the syntax diagram of your language using the ECLIPSE plugin[1] or the ANTLRWORKS tool.[2] This results in a so-called *railroad diagram* of your grammar rules; essentially a variation on a finite automaton where the labels are displayed as nodes and the nodes with incoming/outgoing edges as wiring between the nodes. Explain the difference between the railroad diagrams for `WHILE` and `WS`.
3. Run the JUNIT test `pp.block1.cc.test.ExampleTest`, and make sure you understand what's happening.

[1] In ECLIPSE this purpose, go to "Window → Show View... → Other..." and select "ANTLR 4 → Syntax Diagram".
[2] See BLACKBOARD on how to install ANTLRWORKS

4. Run the JAVA class `pp.block1.cc.antlr.ExampleUsage`, and make sure you understand what's happening. □

Exercise 1-CC.11 Consider the automaton for 6-character identifiers given in Exercise 1-CC.9.

1. Give an ANTLR lexer grammar that will recognize identifiers of this kind. Use so-called ANTLR *fragments* for alphabetic characters (i.e., letters) and for alphanumeric characters, to make your rule(s) more readable (look up what fragments are in the online ANTLR documentation).
2. Test your grammar using the `ID6Test` in the pre-defined lab files. Explain why the last two tests in `successiveTest` go wrong.
3. What is the effect if you omit the keyword **`fragment`** from your grammar? Run the test again, and observe and explain the difference. □

Exercise 1-CC.12 Consider again the grammar for PL/1 strings from Exercise 1-CC.5.

1. Give an ANTLR lexer grammar that will recognize strings of this kind, using the proper ANTLR syntax for exclusion (look it up!).
2. Test your grammar by providing a JUNIT test similar to `ExampleTest` (using the `LexerTester` class). □

Exercise 1-CC.13 Consider the musical scanner of Exercise 1-CC.6.

1. Give an ANTLR lexer grammar that will recognize tokens of this kind.
2. Test your grammar by providing a JUNIT test. □

1-CC.5 EC Section 3.2: Expressing Syntax

Exercise 1-CC.14 Compile for yourself a list of the following terms, with short definitions for each of them. Don't just copy from the book: make sure you understand your own definitions.

Sentential form, parse tree, ambiguity, left/right recursion, recursive-descent parsing, LL(x), bottom-up parsing, LR(x).

□

Exercise 1-CC.15 Regard the following variation on the grammar on Page 12 of the book:

1	*Sentence*	→	*Subject* `verb` *Object* `endmark`
2	*Subject*	→	`noun`
3	*Subject*	→	*Modifier Subject*
4	*Object*	→	`noun`
5	*Object*	→	*Modifier Object*
6	*Modifier*	→	`adjective`
7	*Modifier*	→	*Modifier Modifier*

After scanning, the sentence "all smart undergraduate students love compilers." gives rise to the token sequence

```
adjective adjective adjective noun verb noun endmark
```

Answer the following questions, where you may use the abbreviations:

S	*Sentence*	`n`	`noun`
U	*Subject*	`v`	`verb`
O	*Object*	`a`	`adjective`
M	*Modifier*	`e`	`endmark`

1. Give all leftmost derivations and all rightmost derivations of the above sentence, as well as the parse trees they generate. (There are five of each.)
2. Which parse tree best reflects the grammatical structure of the sentence?
3. How can you change the grammar so that it becomes unambiguous, and the only parse tree of the above sentence is the one you consider to be the best one?

□

Block 2

Block 2

2-OV Overview

2-OV.1 Contents of This Block

CP This block will quickly recapitulate the basic constructs to write concurrent Java programs, as discussed in Module 1.2 (Software systems). Further, we will discuss the meaning of thread safety, different ways to share data between threads, and how to test concurrent programs.

FP In this block the main topics are 1) types and type classes and 2) parsing.

CC In this block we concentrate on the parsing phase of compilation. From EC we will intensively study LL(1) parsing. We'll skip the details of the equally interesting, more powerful but conceptually harder LR(1) parsing algorithm. We then see what ANTLR has to offer in the way of parser rules: quite a bit, as it turns out.

2-OV.2 Mandatory Presence

During the following activities, your presence is mandatory.

2-OV.3 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 4 hours self-study for the CP exercises;
- 4 hours self-study for the FP material.

2-OV.4 Materials for this Block

- **CP**, from JCIP: Chapters 1- 3 and 12.
- **FP** reference material:
  - From LYH: Chapter 8 (lecture 3)
  - From PH: Chapter 8 (lecture 3)
- **CC**, from EC: Sections 3.1–3.3 (intensively), Sections 3.5–3.6 (superficially).

2-CP Concurrent Programming

During this week, you are asked to solve a collection of basic concurrent programming exercises. You are asked to develop complete Java test programs, run these programs, and analyse the results of your tests.

The objective of this first set of exercises is twofold. Firstly, we want you to brush up your Java concurrency knowledge with some simple multi-threaded Java applications. Secondly, you will discover that concurrency bugs are not just theory, but easily happen in practice.

Note that several exercises ask you to show that something is not thread safe. On Blackboard we provide a simple test framework, which you can reuse, such that you do not have to write the boilerplate code, e.g., for spawning and joining threads, every time.

Unsafe Sequence. The goal of the first exercise is to get more insight in the possible behaviours a concurrent program can have. You are asked to implement a class that is not thread-safe, and to demonstrate that it is indeed not thread-safe by analysing the different behaviours.

☞ **Exercise 2-CP.1** *Learning goal:* testing a concurrent application, identifying non-thread-safe behaviour.

1. Take the non-thread-safe sequence generator from Listing 1.1 (JCIP, page 6). Add a test that allows you to demonstrate that it is indeed *not* thread-safe. Due to the randomness of the thread interleaving, different runs will probably yield different results.
2. Take the output of one of your test runs and show where thread safety is violated. Give an interleaving that could have led to the behaviour you observed.
3. Develop at least 3 different alternatives to make the class thread safe and show that these implementations are indeed working properly.

Hint: Even though the sequence generator is not thread-safe, your *test class* should be thread-safe in order to get valid results. □

Producer-Consumer Pipelines. Queues in their various different forms are essential building blocks for multithreaded processing pipelines. In order to achieve maximal throughput it is important that the queue allows for multiple producers to add messages to the queue, and for multiple consumers to process the messages, each running possibly in its own thread. Standard examples of queues are:

- stack (FILO, First In Last Out)
- buffer (FIFO, First In First Out)
- priority queue (the message with the highest priority gets consumed first)

☞ **Exercise 2-CP.2** *Learning goal:* ensuring thread-safety.

In this exercise you are asked to investigate how such a producer-consumer pipeline can be used safely in a concurrent environment. Create your queue (or buffer) adhering to the following simple interface.

```
/**
 * Simple Queue as proposed by the exercise in the Reader.
 * @param <T> The type of objects in the queue.
 */
public interface Queue<T> {
    /** Pushes an element at the head of the queue. */
    void push(T x);

    /** Obtains and removes the tail of the queue. */
    T pull() throws QueueEmptyException;

    /** Returns the number of elements in the queue. */
    int getLength();
}
```

1. Implement a queue, based on a linked list (i.e., a node contains a reference to the next node) without worrying about concurrency considerations.

2. Implement a test class and document the events that cause your data structure to fail for multiple consumers and producers.
   Hint: Again, make sure your test class is thread safe in order to obtain valid results.
3. Change your implementation to make it thread safe and show that your queue passes the tests from assignment 2.
4. Are there ways to further improve your implementation, using finer-grained concurrency, to improve concurrency? If possible, adapt your implementation accordingly.
5. Due to the randomness of thread interleaving it is *impossible* to prove the correctness of the synchronisation by merely running a test. Provide an informal proof why your synchronisation method achieves correctness.

□

Thread Safety. It is often hard to see from documentation whether a class is thread safe and under what conditions.

Exercise 2-CP.3 *Learning goal:* identifying thread-safety-related problems.

For this exercise we will analyse the class `java.util.Timer`. Before proceeding, carefully read the javadoc documentation of `java.util.Timer`.

1. Create a program which has a *single* `Timer` object with n `TimerTask`s associated. Each of these `TimerTask` objects should trigger at the exact same moment t_0. Use `Timer`'s `schedule` method to start the tasks at the same moment t_0 (or after a certain delay). In the `run` method of the `TimerTask` objects you should increment a variable which is shared by all `TimerTasks` objects. After all tasks have finished, check the number of updates to the variable.
2. Create a program which has a n `Timer` objects which trigger at the exact same moment t_0. Each of theses n `Timer` objects gets a single `TimerTask` object which is triggered at t_0. In the `run` method of the `TimerTask` objects you should increment a variable which is shared by all `TimerTask` objects. Again, after all tasks have finished, check the number of updates to the variable.
3. What do you observe? Under what conditions does everything work out and under what conditions does it go wrong? Based on your observations, what do you conclude about the implementation of `java.util.Timer`?

□

Exercise 2-CP.4 *Learning goal:* identifying thread-safety-related problems.

Paragraph 4.5 of JCIP (page 75) mentions that `java.text.SimpleDateFormat` is not thread safe (and so does the javadoc of `SimpleDateFormat`). Show that this is indeed the case.
Hint: Breaking things is usually easier than guaranteeing their correctness. Create some threads which share a `java.text.SimpleDateFormat` object. Let each thread update the shared object to random (but correct!) dates using the method `parse`. □

Assertions and Concurrency. Concurrent execution also has an impact on the claims that you can make about the behaviour of a program. As discussed in Section 3.5.2 of JCIP, concurrency can make assertions invalid, even though from a sequential point of view they would be considered obviously valid.

JML specifications also can be considered as assertions: every precondition translates to an assert at the beginning of the method; every postcondition translates to an assert at the end of the method.

☞ **Exercise 2-CP.5** *Learning goal:* understanding impact of concurrency on reasoning about the behaviour of an application.

Consider the JML-annotated class `Point`.

```java
package pp.block2.cp.annotation;

import net.jcip.annotations.NotThreadSafe;

/**
 * Simple point class as given for the exercise.
 */
@NotThreadSafe
public class Point {
    /*@ spec_public */
    private int x;

    /*@ spec_public */
    private int y;

    //@ ensures \result >= 0;
    /*@ pure */
    public int getX () {
        return this.x;
    }

    //@ ensures \result >= 0;
    /*@ pure */
    public int getY () {
        return this.y;
    }

    /*@
    requires n >= 0;
    ensures getX() == \old(getX()) + n;
    */
    public void moveX(int n) {
        this.x = this.x + n;
    }

    /*@
    requires n >= 0;
    ensures getY() == \old(getY()) + n;
    */
    public void moveY(int n) {
        this.y = this.y + n;
    }
}
```

This class `Point` is used in a class `RandomDrift`, which moves the point up and to the right with random steps.

```java
/**
 * Class which implements the RandomDrift as given in the exercises,
 * but uses a {@link Runnable} rather than a {@link Thread} implementation
 * to make it testable using the JUnit concurrent test runner.
 */
@NotThreadSafe
public class RandomDrift implements Runnable {

    /**
     * The point used by this RandomDrift object.
     */
    private final Point point;

    /**
```

```
     * Amount of drifts to perform.
     */
    private final int amount;

    public RandomDrift(Point point, int amount) {
        this.point = point;
        this.amount = amount;
    }

    @Override
    public void run() {
        for (int i = 0; i < this.amount; i++) {
            int n = (int) (Math.random () * 10);
            this.point.moveX(n);
            int m = (int) (Math.random () * 10);
            this.point.moveY(m);
        }
    }
}
```

1. Class `RandomPoint` starts two `RandomDrift` threads, using the same instance of `Point`. Describe the state space of this program.
2. Discuss where JML specifications might become violated because of concurrency aspects. Show an example execution that violates the specifications.
3. Suppose the methods `moveX` and `moveY` in `Point` become synchronized. Would this solve the problem? Argue why, or why not.

□

2-FP Functional Programming

2-FP.1 Third series: Types and Type Classes

Preparation. This series of exercises is about *trees*.

- In order to visualize trees in a webbrowser, you have to install the module *twentefp-eventloop-trees*, by the (shell) commands (see also the explanation on Canvas, under `Graphics for trees in Haskell`):

  ```
  cabal update
  cabal install twentefp-eventloop-trees
  ```

- You can use the functionalities of this package by including the following line in your program:

  ```
  import FPPrac.Trees
  ```

  Now you have the type `RoseTree`, defined as

  ```
  data RoseTree = RoseNode String [RoseTree]
  ```

  at your disposal, as well as the pre-defined tree `roseExampleTree` and the functions `showRoseTree` and `showRoseTreeList`.

- To show the trees in a browser window, you have to download from Blackboard the file (under `Tooling` → `Graphics for trees in Haskell`):

  ```
  standard_webpage_18-3-17.zip
  ```

Then unpack the `.zip` file, and open the file `standard_webpage.html` in a browser.[1]

- Evaluate the following expressions (check your browser for the output):

```
roseExampleTree
showRoseTree roseExampleTree
```

Remark. In all exercises below you should demonstrate your functions graphically, using your own example trees.

Exercise 2-FP.1 To graphically show trees of a type different from `RoseTree`, they have to be translated into `RoseTree`s first.

1. Given is the following type of binary trees with **`Int`**s at the internal nodes and at the leaves:

```
data Tree1a = Leaf1a Int
            | Node1a Int Tree1a Tree1a
            deriving (Show, Eq)
```

   Define a function `pp1a` (for "pre-processor") that translates a tree of type `Tree1a` into a tree of type `RoseTree`.
2. Define a type `Tree1b` for binary trees that contain 2-tuples of **`Int`**s at the internal nodes and at the leaves. Define a function `pp1b` that transforms trees of type `Tree1b` into trees of type `RoseTree`.
3. Define a type `Tree1c` for binary trees that contains **`Int`**s at the leaves and no information at the internal nodes (*hint*: use empty strings in your rose trees). Define a function `pp1c` that transforms trees of type `Tree1b` into trees of type `RoseTree`.
4. Finally, define the type `Tree1d` that has 2-tuples of **`Int`**s at the leaves, and no information at the internal nodes. It should be possible that a tree of this type has any number of subtrees in its internal nodes. Define a function `pp1d` that transforms trees of type `Tree1d` into trees of type `RoseTree`.
5. Add the following type class definition to your Haskell file:

```
class PP a where
  pp :: a -> RoseTree
```

   Add instances for this type class for: `Tree1a`, `Tree1b`, `Tree1c` and `Tree1d` □

Exercise 2-FP.2
1. Define a function `treeAdd` that adds a number `x` to every number in a tree of type `Tree1a`.
2. Define a function `treeSquare` that squares every number in a tree of type `Tree1a`.
3. Define a function
   `mapTree :: (Int -> Int) -> Tree1a -> Tree1a`
   that applies a function `f` of type **`Int`** `->` **`Int`** to every number in a tree of type `Tree1a`. Define the functions in 1 and 2 in terms of `mapTree`.
4. Define a function `addNode` that replaces every 2-tuple in a tree of type `Tree1b` by the sum of the numbers in each 2-tuple.
5. Define a function `mapTree1b` such that a function `f :: (Int,Int) -> Int` can be applied to every 2-tuple in a tree of type `Tree1b`. Demonstrate your function with several binary operations (e.g., addition and multiplication) - hint: use lambda abstraction. □

Exercise 2-FP.3
1. Define a function `binMirror1a` that mirrors a tree of type `Tree1a`. Check (manually) if mirroring twice results in the original tree again.
2. Define a type class `BinMirror` that offers an interface to a function `binMirror`. Create an instance of this type class for `Tree1a`.
3. Write a variant of the function called `binMirror1d` that works for trees of type `Tree1d` such that also all tuples at the leaves will be swapped.

□

[1] NOTE: not all browsers work equally well because of different choices browsers make concerning all sorts of technical details. However, it works well in *Chrome*.

One further suggestion is to open only *one* tab in your browser, in order to let the communication between Haskell and the browser go smoothly. For more information on some problems with specific browsers, see the abovementioned manual.

Definition. A binary tree with numbers is called *sorted* if for every node in the tree it holds that every number in the *left* subtree is smaller than or equal to the number in that node, and every number in the *right* subtree is larger.

Exercise 2-FP.4 In this exercise we use trees with **`Int`**s at the internal nodes and nothing at the leaves. Define a type `TreeInt` for such trees.

1. Write a function `insertTree` that inserts a number in a `sorted` tree of type `TreeInt`. *Hint*: it is practical to insert a number at a leaf.
2. Write a function `makeTree` that produces a sorted tree from an unsorted list of numbers, by using the function `insertTree`.
   Write your function in two ways: by recursion, and by **`foldl`** or **`foldr`**.
3. Write a function `makeList` that delivers the list of all numbers in a tree. If that tree is sorted, the list should maintain the sorting.
4. Combine the functions above to sort a list. Use QuickCheck to verify this function by using the **`sort`** function from `Data.`**`List`** as a reference.
5. The converse of 4: combine the functions to sort a tree of type `TreeInt`. □

Please ask a TA to check 2-FP.1 to 2-FP.4

Exercise 2-FP.5 Add **`import`** `Data.`**`Maybe`** to your .hs-file.

Write a function `subtreeAt :: TreeInt ->` **`Int`** `->` **`Maybe`** `TreeInt` that searches in a *sorted* tree of type `TreeInt` the subtree at a node with a given number *n*. If the number *n* does not occur in the tree, your function should result in **`Nothing`**. □

Exercise 2-FP.6 Define a function `cutOffAt` that cuts off all branches in a tree of type `Tree1a` at a given depth, leaving shorter branches unchanged. As a result, an internal node may change into a leaf. □

Exercise 2-FP.7 Define a parameterized tree:

```
data BinTree a = Leaf
               | Node a (BinTree a) (BinTree a)
               deriving (Show, Eq) --Eq for the property below
```

In this tree the nodes have values of type `a`.

1. Make `BinTree` an instance of `PP`.
2. Make `BinTree` an instance of `BinMirror`.
3. Make `BinTree` an instance of **`Functor`**.

□

Exercise 2-FP.8 Define `MyList`:

```
data MyList a = Nil | Cons a (MyList a)
              deriving (Show, Eq)

mylst = Cons 1 $ Cons 2 $ Cons 3 $ Nil
```

1. Make `MyList` an instance of **`Functor`**.
2. Define a function `fromList :: [a] -> MyList a` that converts a list to a `MyList`.
3. Use QuickCheck to test if for your functor the first functor law holds (note: you may want to restrict this test to `MyList` **`Int`**). Hint: let QuickCheck generate a list and use `fromList` to convert it.
4. (optional) Use QuickCheck to test if also the second functor law holds (note: again you may want to restrict this test to `MyList` **`Int`**).
5. (suggestion for self-study) Prove both functor laws for `MyList` (use pen and paper). □

Exercise 2-FP.9 In the second exercise set, you defined a database using tuples. The same database is defined in this exercise, but now you need to use *records*.

1. Persons are registered in a database by their: name, age, sex, place of residence. Give the type `Person` of such a database, in which the data of each person are stored as a *record* such that the fields can be easily extracted using their names.

   Create an example database of this type to test the functions you have to write below.
2. Use the record update syntax to define a function `plus :: Int -> [Person] -> [Person]`, which increases the age of all persons by n years.
3. Give a function that gives all names in a gives list of `Person`s. □

Exercise 2-FP.10 The function **`getLine :: IO String`** defines an IO action that reads a **`String`** from the standard input (i.e., usually the keyboard). Use **`getLine`** combined with the IO functor to define an IO action

```
getInt :: IO Integer
```

that reads an **`Integer`**. □

Please ask a TA to check 2-FP.5 to 2-FP.10

2-FP.2 Fourth series: Parsing in Haskell

Exercise 2-FP.11 The `BinTree` type from series 3 has a single type constructor that provides the type of the data stored in the internal nodes. Extend `BinTree` to accept one more type constructor `b`, and use this to extend leaves to contain data of type `b`. □

Parsing The following exercises are about parsing. In the first exercises no tokenizer is needed, and strings are parsed directly. In all exercises below, no error handling is needed.

Exercise 2-FP.12 Consider the following grammar:

$$\begin{array}{rcl} \langle expr \rangle & ::= & \langle term \rangle \ '+' \ \langle expr \rangle \\ & | & \langle term \rangle \\ \langle term \rangle & ::= & \langle factor \rangle \ '*' \ \langle term \rangle \\ & | & \langle factor \rangle \\ \langle factor \rangle & ::= & \texttt{num} \end{array}$$

Here, the terminal symbol `num` is an integer number (single digit).

Examples of valid expressions are:

- $2+3*9$
- 4

1. In this exercise we assume that expressions do not contain spaces, there are no identifiers, and a number consists of only a single digit. Thus, each *token* consists of exactly *one* character.

   Define a function `parseExpr` to transform an expression (given as a string) to an abstract syntax tree (AST) of type `BinTree a b` with appropriate choices for the types used for `a` and `b`. Numbers within the string have to be transformed to **`Int`**s using the function **`read`** (study the type of **`read`**; the polymorphic character of **`read`** means that the result type is determined by the context).

   Note that ASTs are not the same as parse trees, see Figure 2.1 for an example.

   Hint: first write a parser `parseFactor`, and then both `parseTerm` and `parseExpr`.
2. Now you will extend the parser with identifiers and expressions between parentheses. This requires two kinds of leaves in the parse tree: numbers and identifiers. To achieve this, a type is needed that expresses the choice for the type for leaves in `BinTree a b`. Add the following definition to your file:

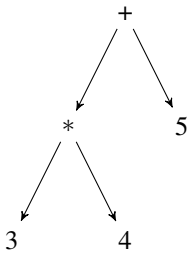


Figure 2.1: AST for the expression $3*4+5$

```
data Value = Const Int
           | Id String
             deriving Show
```

Extend your parser such that it works with the following (extended) productions for ⟨*factor*⟩:

$$\begin{array}{lcl} \langle \textit{factor} \rangle & ::= & \texttt{num} \\ & | & \texttt{identifier} \\ & | & '(' \ \langle \textit{expr} \rangle \ ')' \end{array}$$

Here, the terminal symbol called `identifier` is an identifier that consists of a single letter. □

Please ask a TA to check Exercises 2-FP.11 and 2-FP.12

Exercise 2-FP.13 The following exercises extend the parser above to use a separate type for tokens instead of characters. For this you will define a "proper" tokenizer that satisfies the following requirements:

- *numbers* are integers that consist of one or more digits.
- *identifiers* consist of multiple letters/digits, and start with a letter.
- Separate tokens are defined for opening/closing parentheses, addition and multiplication.
- Spaces are ignored (discarded).

First, define a type `Token` for all the tokens in the expressions grammar. Then define a function `tokenizer :: String -> [Token]` that splits a string into tokens.

Hint: you may use the tokenizer from the lecture slides as basis. □

Exercise 2-FP.14 Define a function *parseExpr'* and the required helper functions that can transform an expression (given as a list of tokens) to an AST. Now variables and identifiers in the tree consist of multiple characters. □

Please ask a TA to check Exercises 2-FP.13 and 2-FP.14.

Exercise 2-FP.15 Define a function `eval` that calculates the value of an expression by first tokenizing and parsing it, and subsequently evaluates the resulting tree. You may write a helper function `evalTree` to evaluate a tree.

Since an expression may contain variables, you need to assign a value to each variable. This can be done, e.g., by defining a *function* that gives a value to each variable, or by defining a *lookup table* (as a list of variable-value pairs). The function `eval` should have this function or lookup table as an extra argument. □

Please ask a TA to check Exercise 2-FP.15

2-CC Compiler Construction

2-CC.1 EC Section 3.3: Top-Down Parsing

Exercise 2-CC.1 Consider the following grammar (which is a variant of the grammar shown on p. 91 of EC):

1	*Stat*	→	`assign`
2		|	`if expr then` *Stat ElsePart*
3	*ElsePart*	→	`else` *Stat*
4		|	ε

Show, through a calculation of the FIRST-, FOLLOW- and FIRST$^+$-sets, that this grammar is not LL(1). In the calculation of the FIRST- and FOLLOW-sets, show the outcome after each iteration of the **`while`**-loops in Figs. 3.7 and 3.8, respectively. □

Exercise 2-CC.2 Make Exercise 3.4 from EC. *L* is the start symbol of the grammar. As in Exercise 2-CC.1, show the iterations in the calculation of FIRST and FOLLOW. *Note:* the terminals are the single `a`, `b` and `c`, *not* sequences such as `aba`. *Also note:* In EC (§3.3.1) it is explained how to turn left-recursive rules into right-recursive ones. □

2-CC.2 Programming the LL(1) algorithm

For the following questions, you have to do some programming. The lab files provided for this block contain the following general classes (in package `pp.block2.cc` and subpackages thereof):

- `Symbol`, `Term` and `NonTerm`: implementations of grammar symbols, further divided into terminals and nonterminals. `Symbol` also contains definitions of the special (terminal) symbols `EMPTY` amd `EOF`.
- `Rule`: a pair of a nonterminal (the rule's left hand side or LHS) and a list of symbols (the rule's right hand side or RHS). Note that if the RHS of a rule is empty, `Rule.getRHS()` will return an empty list, *not* a list containing `Symbol.EMPTY`.
- `Grammar`: a combination of a set of rules and a start symbol (a nonterminal). Note that the special terminals `EMPTY` and `EOF` will normally not occur in a `Grammar`-instance.
- `LLCalc` and `LLCalcTest`: an interface for the calculation of FIRST, FOLLOW and FIRST$^+$, and a JUNIT-test for your implementation of the same.
- `SymbolFactory`: a helper class that can extract token names from ANTLR grammars and build `Term`-instances from those. `SymbolFactory` is used in `LLCalcTest`.

The Javadoc provides further information. Moreover, there are two grammars included in the lab files:

- `Sentence.g4` and `If.g4`: two ANTLR lexer grammars providing the vocabulary of Exercise 1-CC.15 and Exercise 2-CC.1.

Exercise 2-CC.3 Write your own implementation of `LLCalc`, with a constructor that takes a `Grammar` as argument. Make sure you deal correctly with the special symbols `Symbol.EMPTY` and `Symbol.EOF`: those are *not* present in the constructed `Grammar`, you have to add and manipulate them explicitly in your `LLCalc`-implementation. Test your implementation using `LLCalcTest`.

Hint: In the calculation of the FIRST$^+$-set you have to compute FIRST(β) where β is the right hand side of a rule. Since this is also what happens in the **`while`**-loop of the FIRST-algorithm, it makes sense to define an auxiliary method for this. □

Exercise 2-CC.4 Extend `LLCalcTest` with tests for the grammars of Exercise 2-CC.1 and Exercise 2-CC.2.

1. Extend `Grammars` with analogous instances for the *If*-grammar of Exercise 2-CC.1 as well as the grammar of (your answer to) Exercise 2-CC.2. Use the *Sentence* grammar definition as a template. *Note:* in order to use the `SymbolFactory` class for the grammar of Exercise 2-CC.2, you have to create a corresponding ANTLR lexer grammar first.
2. Extend `LLCalcTest` with tests for these grammars, along the lines of:

```
Grammar ifG = Grammars.makeIf(); // to be defined (Ex. 2-CC.4.1)
// Define the non-terminals
NonTerm stat = ifG.getNonterminal("Stat");
NonTerm elsePart = ifG.getNonterminal("ElsePart");
// Define the terminals (take from the right lexer grammar!)
Term ifT = ifG.getTerminal(If.IF);
... // (other terminals you need in the tests)
Term eof = Symbol.EOF;
Term empty = Symbol.EMPTY;
LLCalc ifLL = createCalc(ifG);

@Test
public void testIfFirst() {
    Map<Symbol, Set<Term>> first = ifLL.getFirst();
    assertEquals(/* see 2-CC.1 */, first.get(stat));
    // (insert other tests)
}

@Test
public void testIfFollow() {
    Map<NonTerm, Set<Term>> follow = ifLL.getFollow();
    assertEquals(/* see 2-CC.1 */, follow.get(stat));
    // (insert other tests)
}

@Test
public void testIfFirstPlus() {
    Map<Rule, Set<Term>> firstp = ifLL.getFirstp();
    List<Rule> elseRules = ifG.getRules(elsePart);
    assertEquals(/* see 2-CC.1 */, firstp.get(elseRules.get(0)));
    // (insert other tests)
}

@Test
public void testIfLL1() {
    assertFalse(ifLL.isLL1());
}
```

   (and similar for the grammar of Exercise 2-CC.2)
3. Confirm your own answers to Exercise 2-CC.1 and Exercise 2-CC.2 by running the extended test.

□

In Section 3.3.2 of EC, you can read about the principle of (hand-coded) recursive-descent parsers based on the LL(1) principle. In particular, if you have calculated the FIRST$^+$-set of the rules of your grammar, then the corresponding recursive-descent is really easy to write. (Actually, it becomes a bit harder if you also want to deal with errors in a usable fashion, in particular to make sure that your parsing doesn't just give up on the first error; however, we'll simply ignore this here. See Section 3.5.1 for a discussion.)

Exercise 2-CC.5 Among the lab files, you will also find `cc.block2.ll.SentenceParser`, which is a hand-written recursive-descent parser for the *Sentence*-grammar of Exercise 1-CC.15 — minus Rule 7, which (as you have seen) makes the grammar ambiguous. The class has a `main`-method that lets you try it out on texts you pass in as arguments.

1. Try out `SentenceParser` on a few sample inputs, both correct and wrong. Note that the lexer grammar `Sentence.g4` determines which words are actually recognised. in particular, confirm that the example sentence of Exercise 1-CC.15 gives rise to the parse tree corresponding to your answer to 1-CC.15.3.
2. Write a recursive-descent parser for the grammar of Exercise 2-CC.2, analogous to `SentenceParser`. (You should already have created the necessary lexer grammar in your solution to Exercise 2-CC.4.)
3. Test your solution to the previous subquestion on the input texts `abaa`, `cababcbca`, `bbcca` and `bbcba`.

□

Exercise 2-CC.6 You should now have grasped the principles of top-town, LL(1) parsing well enough to be able to write a table-driven parser. In Section 3.3.3 of EC (Figure 3.11) you can find a *non-recursive* algorithm for writing a table-driven parser. However, for this exercise you should build a *recursive* table-driven parser so that it can simultaneously produce a parse tree. The lab-file `pp.block2.cc.ll.GenericLLParser` gives a skeleton for you to fill in.

1. Program `GenericLLParser`.
2. Make sure that `pp.block2.cc.test.GenericLLParserTest` shows no errors.
3. Extend `GenericLLParserTest` with a method to test `GenericLLParser` by comparing the grammar of Exercise 2-CC.2 against the hand-written parser of Exercise 2-CC.5, analogous to the test for the *Sentence*-grammar in `testSentence()`.

Note that the implementation of the LL(1)-parse table (Figs. 3.11(b) and 3.12 of EC, see p. 112–113) foreseen in this file is of type `Map<NonTerm, List<Rule>>`, which maps every non-terminal to a list of rules indexed by token type. The token types are the numbers corresponding to the tokens in the ANTLR-generated lexer. □

2-CC.3 ANTLR parser grammars

In `pp.block2.cc.antlr.Sentence.g4` you will find a full grammar specification (not just a lexer) for the *Sentence*-grammar of Exercise 1-CC.15. (It is actually more elegant to *import* a lexer grammar into a full grammar, rather than just copy/pasting the whole thing as done here; but in the context of this course, the chosen package structure gets in the way.) The first few lines of the grammar are:

```
1  grammar Sentence;
2
3  @header{package pp.block2.cc.antlr;}
4
5  /** Full sentence: the start symbol of the grammar. */
6  sentence: subject VERB object ENDMARK;
7  /** Grammatical subject in a sentence. */
8  subject: modifier subject | NOUN;
9  /** Grammatical object in a sentence. */
10 object: modifier object | NOUN;
11 /** Modifier in an object or subject. */
12 modifier: ADJECTIVE;
```

The following should be noted:

- The first line now reads **`grammar`** rather than **`lexer grammar`**, reflecting the fact that this contains both parser and scanner (i.e., lexer) rules.
- The rules `sentence` etc. have essentially the same syntax as the lexer rules; in fact, the most prominent difference is that the names of the nonterminals start with a lowercase letter.

If you now generate the corresponding recogniser, you will get four JAVA files instead of just one:

- `SentenceListener`: an interface for listeners to the parser. We'll see below how to use listeners.
- `SentenceBaseListener`: a skeleton implementation of the above, with empty listener methods.

- `SentenceLexer`: this equals the lexer class generated for the lexer grammar.
- `SentenceParser`: this class inherits from `org.antlr.v4.runtime.Parser`, which in turn offers functionality to add and remove `SentenceListener`s.

Exercise 2-CC.7 Generate the above files from `pp.block2.cc.antlr.Sentence.g4`, and run `SentenceUsage`. Study the class and make sure you understand what goes on. What happens if you parse an incorrect text? □

The best way to use ANTLR parse trees such as returned by `SentenceParser.sentence()` is to *walk* the tree using a tree listener; specifically, a `SentenceListener`. See `https://github.com/antlr/antlr4/blob/master/doc/listeners.md` and elsewhere on the web to find more information on how to program and invoke ANTLR tree listeners. Furthermore, `pp.block2.cc.antlr.SentenceCounter` contains an example.

Exercise 2-CC.8 Program `SentenceConverter` to transform ANTLR parse trees of the `Sentence`-language to `pp.block2.cc.AST` instances.

1. Make sure you can run `pp.block2.cc.antlr.SentenceCounter` and understand how the code works.
2. Extend `pp.block2.cc.antlr.SentenceConverter` given in the lab files.
3. Test your solution using the provided `SentenceConverterTest` class. □

ANTLR grammars are in fact quite a bit more powerful than LL(1)-grammars. Among other things, direct left-recursion is automatically factored out. This means that it is often possible to write rules that directly express the intended tree structure, as for instance in the case of operator precedence (also treated extensively in Chapter 3 of EC).

For the following question, there are some special features of ANTLR that come in quite handily.

- The alternatives of a rule are processed in the order given in the grammar. Thus, the first alternative is tried out first. This makes it convenient to program operator precedence.
- You can *name* the top-level alternatives of a rule by inserting a tag of the form `# myName` into the grammar. This will result in modified listener methods that are called precisely when that alternative is visited, rather than every time the whole rule is visited. For instance, in the `Sentence` grammar above, you could write the rule for `subject` as

```
subject : modifier subject # modSubject
        | noun             # simpleSubject
        ;
```

  resulting in listener methods `[enter/exit]ModSubject` and `[enter/exit]SimpleSubject`, *replacing* the previous `[enter/exit]Subject`.

- You can explicitly set the associativity of a top-level alternative by inserting the directive `<assoc=right>` directly in front of the alternative. For instance, in `Sentence.g4` above, you could change the `modifier` rule to

```
modifier : <assoc=right> modifier ',' modifier
         | adjective
         ;
```

  resulting in a non-ambiguous grammar in which adjectives are interpreted in a right-associative manner.

Exercise 2-CC.9 Try out the features explained above (on a *copy* of the `Sentence` grammar, so you don't ruin your solutions to the exercises above!)

1. Change the rule for `subject` in the way described above, and study the resulting `SentenceListener` interface. What parameters do `enterModSubject` and `enterSimpleSubject` etc. get, and what functionality do those parameters offer?

2. Change the rule for `modifier` in the way described above, and try parsing the sentence of Exercise 1-CC.15 (with commas inserted between the adjectives). If you look again at your answer to that exercise, which parse tree does the one now generated by ANTLR correspond to? If you leave out the **`assoc`**-directive (but leave in the left-recursive alternative itself), which parse tree do you then get? Use the "Parse Tree View" of your ANTLR plugin as a fast way to check this. (see BLACKBOARD for a brief explanation about how to get parse tree views to work under ECLIPSE and INTELLIJ.) □

Exercise 2-CC.10 Design a language for arithmetic expression that includes addition (e.g., `1+2`), subtraction (e.g., `2-3`), multiplication (e.g., `2*3`), negation (e.g., `--2` should stand for double negation, yielding 2) and exponentiation (e.g., `2^3`), in increasing order of precedence. Addition, subtraction and multiplication are left-associative (for instance, `1-2-3` stands for `(1-2)-3` and not `1-(2-3)`) but exponentiation is right-associative (for instance, `2^3^4` stands for `2^(3^4)` and not `(2^3)^4`). Of course, the language should also provide parentheses and (natural) numbers.

1. Write an ANTLR grammar for this arithmetic expression language.
2. Program a `Calculator` as a tree listener that can take an expression in this language and compute the outcome, based on `BigInteger` values.
3. Write a test for your class where you demonstrate all features of the expression language. □

Block 3

Block 3

3-OV Overview

3-OV.1 Contents of This Block

CP This block will discuss synchronisation using locks (both intrinsic and explicit), condition variables, and other synchronizers. A warning is in place: the concurrent programming exercises for this block are interesting, but also challenging. Make sure to block enough time to work on them.

FP This block first continues with parsing: the ParSec tool is used to get familiar with parser combinators. Also several advanced type classes are introduced, namely `Monoid`, `Foldable` and `Applicative`.

CC This block will address the "elaboration" phase of the first stage of compilation. This involves especially type and scope checking/inference. As solutions, you will learn about attribute grammars, syntax-directed translation and symbol tables.

3-OV.2 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 4 hours self-study to prepare the CP exercises;
- 4 hours self-study to complete the FP lab exercises;
- 4 hours self-study to complete the CC lab exercises.

3-OV.3 Materials for this Block

- **CP**, from JCIP: Chapters 4, 5, 13, and 14.1-14.4.
- **FP** reference material:
  - From LYH: Chapters 8 and 11 (lecture 6)
  - From PH: Chapters 13 (lecture 5), Chapters 12.1-12.2 and 14.1-14.2 (lecture 6)
  - For parser combinators and ParSec: lecture slides and example code
  - ParSec API documentation:
    - Text.ParserCombinators.Parsec
    - Text.ParserCombinators.Parsec.Token

  - Text.ParserCombinators.Parsec.Combinator
  - Text.ParserCombinators.Parsec.Language
- **CC**, from EC: Chapter 4 and Section 5.5.

3-CP Concurrent Programming

Exercise 3-CP.1 *Learning goal:* identifying concurrency problems.

Note: a variation of this exercise has been used in the CP Test of 29 June 2016.
Consider the class `Hotel` below, which implements some simple hotel functionality, maintaining an array of rooms and a list of people waiting to check in. A clumsy attempt has been made to make it suitable for concurrent use. For simplicity, it does not model that guests can check out; you can assume that a separate thread will take care of this.

```
package pp.block3.cp.hotel;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * Simple Hotel class as given in exercise 1 of block 3.
 */
public class Hotel implements Runnable {

    private final static int NR_ROOMS = 10;
    private final Person[] rooms = new Person[NR_ROOMS];
    private final List<Person> queue = new ArrayList<>();
    private final Lock queueLock = new ReentrantLock();

    private boolean occupied(int i) {
        return (rooms[i] != null);
    }

    private int checkIn(Person p) {
        int i = 0;
        while (occupied(i)) {
            i = (i + 1) % NR_ROOMS;
        }
        this.rooms[i] = p;
        return i;
    }

    private void enter(Person p) {
        this.queueLock.lock();
        this.queue.add(p);
        this.queueLock.unlock();
    }

    // every desk employee should run as a separate thread
    @Override
    public void run() {
        while (true) {
            if (!this.queue.isEmpty()) {
                this.queueLock.lock();
                Person guest = this.queue.remove(0);
                this.queueLock.unlock();
```

```
                checkIn(guest);
            }
        }
    }

}
```

1. Class `Hotel` implements the `Runnable` interface, and therefore contains a `run` method. Discuss why this is a bad choice, and what would be a way to improve this.
2. The implementation also several other concurrency errors. Discuss *at least three* different errors, explain why they cause a problem, and what should be done to fix them.

□

☞ **Exercise 3-CP.2** *Learning goal:* understanding that the preservation of a relation between variables influences locking policies.

Consider the class `Point` and its JML annotations below. It is similar in spirit to the class `Point` that we discussed last week, except that:

- it uses `synchronized` blocks to make sure that the access to the shared variables `x` and `y` is properly protected, and
- it tries to ensure that `x` and `y` never get the same value.

```
package pp.block3.cp.jml;

/**
 * The point class given with exercise 2 of block 3.
 */
public class Point {

    /*@ spec_public */private int x = 0;
    /*@ spec_public */private int y = 1;
    //@ public invariant x != y;

    private final Object lockX = new Object();
    private final Object lockY = new Object();

    //@ ensures \result >= 0;
    /*@ pure */
    public int getX() {
        synchronized (this.lockX) {
            return this.x;
        }
    }

    //@ ensures \result >= 0;
    /*@ pure */
    public int getY() {
        synchronized (this.lockY) {
            return this.y;
        }
    }

    /*@ requires n >= 0;
        ensures getX() == \old(getX()) || getX() == \old(getX()) + n; */
    public void moveX(int n) {
        boolean b;
        synchronized (this.lockX) {
            synchronized (this.lockY) {
```

```
                b = (x + n != y);
            }
        }
        synchronized (this.lockX) {
            if (b) {
                this.x = this.x + n;
            }
        }
    }
    /*@ requires n >= 0;
        ensures getY() == \old(getY()) || getY() == \old(getY() + n); */
    public void moveY(int n) {
        boolean b;
        synchronized (this.lockX) {
            synchronized (this.lockY) {
                b = (this.x != this.y + n);
            }
        }
        synchronized (lockY) {
            if (b) {
                this.y = this.y + n;
            }
        }
    }

}
```

For all JML specifications of `Point` (i.e., for the methods and the class invariant) discuss the following:

1. Are they valid when the class `Point` is used in a concurrent context, where multiple threads can simultaneously call the methods in `Point`? If so, explain why. If not, give a counter example.
2. If the specifications are not valid, how can the specifications and/or the program be adapted to make the specifications valid?
3. What happens if we change the order of the locks in `moveY`: first acquire `lockY` and subsequently acquire `lockX`?

□

Fine-grained locking. When implementing a concurrent data structure, there are several strategies you can use with regard to locking. The most simple and straightforward is to simply protect all accesses and changes of the data structure by a single lock. This does result in a correct program, but it can make the performance of your program less optimal than expected for a multithreaded program, as you essentially make all accesses to the data structure sequential.

This assignment deals with a different strategy: *fine-grained locking*. Simply put, if you want to reduce the congestion of a lock (which will be very high when you use a single lock) you could use multiple locks to spread the congestion to a more manageable level, improving the performance. In the case of a linked list, the most fine-grained solution is to protect every element with its own lock. When you only want to set, get, prepend or append values this approach is fairly straightfoward to implement. However, when the operations on the linked list also include e.g., insertions and deletions, only holding a single lock does not guarantee correctness any more. To solve this issue, the concept of *hand-over-hand* locking or *lock-coupling* was invented. A basic explanation of lock-coupling is given in the book (JCIP, 13.1.3, page 281-282).

☞ **Exercise 3-CP.3** *Learning goal:* complex locking policies.

1. *Implement* a linked list using lock-coupling, capable of (at least) insertions and deletions. Obviously, the list should behave correctly in a multithreaded environment, i.e., there should not be any deadlocks or invalid memory accesses.

   Your linked list class should (at least) implement the following interface:

```
package pp.block3.cp.lockcoupling;

/**
 * Interface as given for implementing an concurrent linked list
 * in exercise 3 of block 3.
 */
public interface List<T> {
    /**
     * Insert an element at the specified position in the list.
     * @param position The position to insert the element at.
     * @param value The value of the element to insert.
     */
    void insert(int position, T value);

    /**
     * Add an element to the end of the list.
     * @param value The value of the element to add to the list.
     */
    void add(T value);

    /**
     * Remove the specified element from the list.
     * @param item The element to remove from the list.
     */
    void remove(T item);

    /**
     * Delete the element at the specified position.
     * @param position The position of the element that should be deleted.
     */
    void delete(int position);

    /**
     * Get the amount of elements currently in this list.
     * @return The size of the list.
     */
    int size();

    /**
     * A toString function for the list.
     * @return The string representing the list.
     */
    String toString();
}
```

2. Report on the performance gains when compared with guarding the entire list by a single lock.
3. What are the issues that you encountered whilst developing your list implementation?

□

Custom synchronizers. The Java concurrency library contains quite a few different and useful *synchronizers*: a wide variety of locks and semaphores. However, in order to really understand the inner workings of a synchronizer it is instructive to implement one yourself. The first binary mutual-exclusion (*mutex*) algorithm which was proven to be correct was Dekker's algorithm.

☞ **Exercise 3-CP.4** *Learning objective: Understanding different ways to implement a synchronizer*Consider the following `BasicLock` interface which can be used to protect a critical section. The argument `thread_nr` makes it easier to implement the algorithms of this exercise.

```
package pp.block3.cp.synchronizers;
```

```
/**
 * BasicLock interface which can lock with 2 threads based on thread number.
 */
public interface BasicLock {
    /**
     * Acquires the lock.
     * @param threadNumber is the number of the requesting thread,
     * threadNumber == 0|1
     */
    void lock(int threadNumber);

    /**
     * Releases the lock.
     * @param threadNumber is the number of the releasing thread,
     * threadNumber == 0|1
     */
    void unlock(int threadNumber);
}
```

1. Look up a pseudocode version of Dekker's algorithm and use it to make an implementation of `BasicLock`. Write a simple test for your mutex class and use it to verify your implementation. For example, you could use the `UnsafeSequence` class of block 1, Exercise 1, and show that your mutex implementation protects an unsafe variable in a thread-safe way.

   Hint: Instead of using two `volatile` boolean variables to express the desire to enter the critical section you could consider to use a `AtomicIntegerArray`.
2. Create a new mutex class, based on the *compare-and-set* instruction from the `AtomicInteger` class. Start out with a non-reentrant version, suitable for only two threads. Re-use the test harness of Dekker's implementation and validate that your *compare-and-set* mutex works correctly.
3. Expand the *compare-and-set* mutex implementation to accomodate reentering. If a thread holds the lock and requests it again this will be allowed; a non-reentrant lock would cause a deadlock in this scenario.
4. Use the `ReentrantLock` from the Java library to implement your `BasicLock`.

□

Barriers and map-reduce. Barriers (or fences), as the name implies, are used to separate your program into sections; execution is only allowed to continue whenever *all* threads in the section have finished that section. A simple example in which this is useful is adding all numbers in a certain array. A single-threaded solution would use a single thread to loop over all elements. However, when using a barrier, you can instruct thread `1` to sum the first half of the array and thread `2` to sum the second half of the array, almost halving the execution time, though one still needs to add the two sub-sums to obtain the final answer.

This simple example is a course-grained concurrent version of the approach known as *map-reduce*. The first step (*map*) is to apply a function to all elements of the collection; in this example, the two sub-arrays. The executions of these functions are independent. The second step (*reduce*) is to apply a function which merges the results of the map-operation into the final answer.

A more realistic example of map-reduce is a possible implementation of a web analyzer, e.g., Google's PAGERANK. The input of this algorithm is a very large set of HTML files, found by a webcrawler. These are processed individually in order to extract keywords, find hyperlinks and anything else which might be useful in later analysis. This is the map-stage. Next, this mapped data is fed to the reducer where it gets combined to obtain useful information.

Note: you might recognize the term `map` from its origin in functional programming.

☞ **Exercise 3-CP.5** *Learning goal:* Understand how to use alternative synchronization mechanisms.

1. Create a minimal *map-reduce* framework in Java using barriers. The idea is to develop an abstract `MapReduceBase` class, which has two abstract methods `map` and `reduce`, which will be implemented by subclasses of `MapReduceBase`. The class `MapReduceBase` is responsible for spawning the threads that perform the `map` and combining the results with `reduce`.
2. As always: devise a test, and apply it to make sure that your framework works as expected.

□

3-FP Functional Programming

3-FP.1 Fifth series: ParSec

Exercise 3-FP.1 First install ParSec using Cabal: `cabal install parsec`

Use `Set5.hs` from the zip-file for this block as basis. In the exercises below you should extend this file with your own definitions. The parser defined in the following exercises should generate an embedded domain specific language (EDSL). This is still very similar to an AST. Try if ParSec works using a few of the examples from the lecture. □

Exercise 3-FP.2 Provide a Parsec language definition `languageDef` for the following grammar:

$$
\begin{array}{rcl}
\langle expr \rangle & ::= & \langle term \rangle \ (\texttt{'+'}\ \langle term \rangle)* \\
\langle term \rangle & ::= & \langle factor \rangle \ (\texttt{'*'}\ \langle factor \rangle)* \\
\langle factor \rangle & ::= & \texttt{num} \\
& | & \texttt{identifier} \\
& | & \texttt{'('}\ \langle expr \rangle\ \texttt{')'}
\end{array}
$$

The unquoted $*$ means "zero or more times", as usual.

Use `languageDef` to define the following functions for *tokenization*: `lexer`, `identifier`, `integer`, `parens`, `symbol` and `reserved`. You may want to use the lecture slides to see how this can could be achieved. □

Exercise 3-FP.3 In the following you define a parser for the grammar given above.

1. Use Parsec to define a parser function `parseFactor :: Parser Expr`, in which the production `'('` ⟨*expr*⟩ `')'` is still omitted (since ⟨*expr*⟩ is not yet defined). Note that the type `Expr` is already provided and should later be extended. Use **`Integer`** for the numeric type, and **`read`** to convert a **`String`** to an **`Integer`**.

   Verify that your parser can obtain the following results:

   ```
   *Set5> parser parseFactor "123"
   Const 123
   *Set5> parser parseFactor "test2"
   Var "test2"
   ```

2. Define a parser `parseTerm :: Parser Expr`
3. Define a parser `parseExpr :: Parser Expr`

   Now also include the ⟨*factor*⟩ production for parenthesized expressions `'('` ⟨*expr*⟩`')'`. Verify that your parser can produce the following result:

   ```
   *Set5> parser parseExpr "3*(1+a)"
   Mult (Const 3) (Add (Const 1) (Var "a"))
   ```

□

Please ask a TA to check Exercises 3-FP.1 through 3-FP.3

Exercise 3-FP.4 Consider the following extension to the grammar:

$$\begin{array}{rcl} \langle condition \rangle & ::= & \langle expr \rangle \texttt{'==' } \langle expr \rangle \\ \langle if \rangle & ::= & \texttt{'if' } \langle condition \rangle \texttt{ 'then' } \langle expr \rangle \texttt{ 'else' } \langle expr \rangle \\ \langle dec \rangle & ::= & \texttt{'dec' } \langle expr \rangle \end{array}$$

1. Define a data type that descibes the result of parsing a $\langle condition \rangle$ nonterminal. Also extend the `Expr` data type to support the functionality described by the $\langle if \rangle$ and $\langle dec \rangle$ nonterminals.
2. Define the parsers `parseCondition`, `parseIf` and `parseDec`.

   Test `parseIf` at least with the following input: `if 1 == (0+1) then a else 1+1`
3. Extend `parseExpr` such that it supports the following grammar:

$$\langle expr \rangle \quad ::= \quad \langle dec \rangle \quad | \quad \langle term \rangle \, (\texttt{'+' } \langle term \rangle) * \quad | \quad \langle if \rangle$$

□

Exercise 3-FP.5 In this exercise the grammar is extended as follows:

$$\begin{array}{rcl} \langle function \rangle & ::= & \texttt{'function' identifier identifier '=' } \langle expr \rangle \\ \langle factor \rangle & ::= & \texttt{num} \\ & | & \texttt{identifier '(' } \langle expr \rangle \texttt{ ')'} \\ & | & \texttt{identifier} \\ & | & \texttt{'('} \langle expr \rangle \texttt{')'} \end{array}$$

This adds the possibility of defining functions that receive a single argument and to use these functions in expressions.

1. Define a data type `FunDef` that describes a function definition, and extend the data type `Expr` with a data constructor to support function calls.
2. Define parser functions for the extensions.
3. Add the following definitions to your file and check if the function can be parsed:

```
parserFun :: String -> FunDef
parserFun = parser parseFunc -- replace parseFunc by the name of your parser

fib :: FunDef
fib = parserFun
 "function␣fib␣x␣=␣if␣x␣==␣0␣then␣1␣else␣(if␣x␣==␣1␣then␣1␣else␣fib(dec␣x)+fib(dec␣dec␣x))"
```

□

Please ask a TA to check Exercises 3-FP.4 and 3-FP.5

Exercise 3-FP.6 The parser from the previous exercises builds an EDSL that descibes some computations. For this exercise you define a function that evaluates a function in our EDSL for a given **`Integer`**.

1. Define a function

   ```
   evalfun :: FunDef -> Integer -> Integer
   ```

   which receives a `FunDef` function and an integer to which it should be applied, and produces an **`Integer`** that is the result of the calculation. Define all separate functions to evaluate expressions, conditions, etc.

   Some remarks about how to evaluate the EDSL:

- Only one function can be defined; there is no syntax to define multiple functions.
- Feel free to ignore the function name and argument name (i.e., you always bind them to the variable/function that were provided), since there is only one variable and function in the scope. For example:

  ```
  correctfun = parser parseFunc
               "function␣f␣x␣=␣if␣x␣==␣0␣then␣1␣else␣y+g(0)"
  ```

  This function evaluated for 0 gives 1, and when evaluated for 10 it gives 11.
- *dec* decreases a value by one.
- *if* is an *expression*, just like in Haskell.

2. Add the following test code to your file, and use QuickCheck to test your code.

   ```
   -- Checks if in the evaluation (x^2 + 2*x + 1) == (x+1)^2
   fn  = (evalfun . parserFun) "function␣f␣x␣=␣x*x␣+␣2*x␣+␣1"
   fn' = (evalfun . parserFun) "function␣f␣x␣=␣(x+1)␣*␣(x+1)"
   prop_fn n = n >= 0 =⇒ fn n == fn' n
   ```

3. Add the following function that calculates factorials, and test it using the given QuickCheck test.

   ```
   factorial :: Integer -> Integer
   factorial = (evalfun . parserFun)
     "function␣factorial␣x␣=␣if␣x␣==␣0␣then␣1␣else␣factorial(dec␣x)␣*␣x"
   prop_factorial n = n >= 0 =⇒ factorial n == product [1..n]
   ```

□

Please ask a TA to check Exercise 3-FP.6

3-FP.2 Sixth series: Advanced Type Classes

Name your Haskell file `Set6.hs` and include the following code (you may use `Set6.hs` from the zip-file of this block):

```
module Set6 where

import Data.Foldable
import Data.Monoid
import Text.Read
import Data.Char
import Control.Applicative
```

Exercise 3-FP.7 Define a Monoid instance for `Int` that uses addition as its binary operator. Use this monoid and `mconcat` to define a function `sumint :: [Int] -> Int` that sums a list of integers. □

Exercise 3-FP.8 Add the following to your file:

```
data A a = A { fromA :: Maybe a }
         deriving Show
```

1. Define a Monoid for `A` of which the binary operator provides the *first* value with the constructor `Just`.
2. Use this Monoid together with `mconcat` to define a function

   ```
   firstInt :: [String] -> Maybe Int
   ```

   that receives a list of strings and looks for the first string that can be read as an `Int`. You may use the function

   ```
   readMaybe :: Read a => String -> Maybe a
   ```

from `Text.Read` (Google can be used to lookup the documentation).

Example usage:

```
*Set6> firstInt ["a", "3.2", "1", "2"]
Just 1
*Set6> firstInt ["a", "1a", "2a"]
Nothing
```

□

Exercise 3-FP.9 Use the applicative style to define a function

```
addstr :: String -> String -> Maybe String
```

that receives two strings `xs` and `ys`, and if both contain integer they are added. Otherwise the function results in `Nothing`. You may use the function

```
readMaybe :: Read a => String -> Maybe a
```

from `Text.Read`.

Example usage:

```
*Set6> addstr "123" "9"
Just "132"
*Set6> addstr "123a" "9"
Nothing
```

□

Please ask a TA to check Exercises 3-FP.7, 3-FP.8 and 3-FP.9

Exercise 3-FP.10 Briefly study the Haskell functions `zipWith` and `zipWith3` and make sure you understand what they can be used for. Add the following code to your file:

```
data MyList a = Nil | Cons a (MyList a)
              deriving (Show, Eq)

mylst  = Cons 1   $ Cons 2   $ Cons 3   $ Nil
mylst2 = Cons 10  $ Cons 20  $ Cons 30  $ Nil
mylst3 = Cons 100 $ Cons 200 $ Cons 300 $ Nil
```

Also add the `MyList` functor instance you defined in the previous block.

1. Define the function

   ```
   myzipWith3' :: (a->b->c->d)->MyList a->MyList b->MyList c->MyList d
   ```

2. Make `MyList` a member of `Applicative` in such a way that it "zips" lists. Verify that it has the following behavior:

   ```
   *Set6> (+) <{$}> mylst <*> mylst2
   Cons 11 (Cons 22 (Cons 33 Nil))
   ```

3. Use the applicative style to define the functions `myzipWith` and `myzipWith3`, which should be the `MyList` versions of `zipWith` and `zipWith3`. □

Exercise 3-FP.11 Use the *applicative style* to define an IO action `f :: IO Integer` that reads two integers from the standard input and adds them.

Recall: `IO` has an (applicative) functor instance. You may use/copy the function `getInt` that you defined in block 2. □

Exercise 3-FP.12 The function `justs :: [Maybe a] -> Maybe [a]` receives a list of **Maybe** values, and results in **Nothing** when one or more of these values is **Nothing**, otherwise it produces the list of values.

Define the function `justs` by combining *recursion* and the *applicative style*. □

Please ask a TA to check Exercises 3-FP.10, 3-FP.11 and 3-FP.12

Exercise 3-FP.13 Endo functions are functions from `a` to `a`, i.e., they functions for which the argument and result have the same type. The following data type describes such functions in general:

```
data MyEndo a = MyEndo {
  apply :: a -> a
  }
```

1. Define a Monoid for `MyEndo` for which the operation behaves as function composition.
2. Use `mconcat` to define the function `listtoendo :: [a -> a] -> MyEndo a`, which makes for a list of functions `fs` the endo function that is the sequential composition of all functions `fs`.

   Test this function using:

   ```
   transform :: String -> String
   transform = apply $ listtoendo [map toUpper, reverse, filter isLetter]
   ```

   This function behave similarly as `transform` from Set 2. □

Exercise 3-FP.14 Consider the following parser data type:

```
data Parser r = P {
  runParser :: String -> [(r, String)]
}
```

And the (parameterised) character parser:

```
char :: Char -> Parser Char
char c  = P p
 where
   p []                 = []
   p (x:xs) | c == x    = [(x, xs)]
            | otherwise = []
```

1. Define a parser `parseOne` that parses the single character '1'. Define functions that tests your parser: one that is succesful and one that fails.
2. Define a **Functor** instance for `Parser` that maps over the parsing result
3. Define a parser `parseOneInt :: Parser Int` that parses the single character '1' and result in an **Int**. Define functions that tests your parser: one that is succesful and one that fails.
4. Define an `Applicative` instance for `Parser` that combines two parsers by parsing sequentially. Test this parser using:

   ```
   parseAB :: Parser (Char, Char)
   parseAB = (,) <$> char 'a' <*> char 'b'
   ```

5. This is a new exercise, which helps the students to prepare for the FP project.
   Define the function `parseString :: String -> Parser String` that generates a parser for a given **String**.
   Hint: Use recursion, where `pure` is used to define the base case for the empty string.
6. **(optional)** Define an `Alterative` instance for `Parser` that combines two parsers by trying both. □

Exercise 3-FP.15 This question assumes that your ParSec solutions are stored in a file `Set5.hs` and has the module name `Set5`. Add the following to the imports at the top of your `Set6.hs` file:

```
import Set5
```

Extract `fib.txt` from the zip-file for this block and put it in the same directory as your Haskell files. You can read the file using **`readFile :: FilePath -> IO String`**. For example, try **`readFile`** `"fib.txt"` in GHCi, and use `:t` to determine the type of this expression.

1. Use the applicative style to define a function

   ```
   fibonacci :: IO (Integer -> Integer)
   ```

   that reads the file, uses the function parser from the previous set, and evaluates it.
2. Add the following definitions to your file.

   ```
   fib5 :: IO Integer
   fib5 = fibonacci <*> pure 5

   fibs :: IO [Integer]
   fibs = (map <$> fibonacci) <*> pure [0..]
   ```

   Verify that the functions produce the correct results.

> **Important**: the exercises below are optional.

3. Use the applicative style to define

   ```
   prompt :: IO Integer
   ```

   which reads an expression (i.e., a **`String`**) and an **`Integer`** from the standard input, parses it and evaluates it for the give **`Integer`**. Since a `FunDef` argument is required, you may use `fact :: FunDef` defined as

   ```
   fact = parserFun
      "function␣fact␣x␣=␣if␣x␣==␣0␣then␣1␣else␣fact(dec␣x)␣*␣x"
   ```

   Hint: If the arguments of the evaluation expression are not in a convenient order, use either the function **`flip`** or a lambda abstraction.
4. Define a function

   ```
   calculations :: Integer -> [String] -> [Integer]
   ```

   that receives a list of expressions (i.e. `[String]`) and an **`Integer`**. It parses all expressions, and evaluates them using the provided **`Integer`** and the function `fib`. Note, that no **`IO`** is used in this exercise.
5. Use the applicative style to define a function

   ```
   calcfile :: FilePath -> Integer -> IO [Integer]
   ```

   which receives the name of a file that contains some expressions and an integer to bind to the variable within the expressions. This function uses `calculations` to parse and evaluate all expressions. □

> **Please ask a TA to check Exercises 3-FP.13 through 3-FP.15**

3-CC Compiler Construction

3-CC.1 EC Chapter 4: Type inference and attribute grammars

Exercise 3-CC.1 Compile for yourself a list of the following terms, with short definitions for each of them. Don't just copy from the book: make sure you understand your own definitions.

Operator overloading, type inference, synthesized resp. inherited attribute, syntax-directed translation.

□

Exercise 3-CC.2 Answer Review Question 1 of Section 4.2 (p. 181 of EC) for JAVA. Explicitly compare the types in JAVA with each of the categories discussed in Section 4.2.2 (base types, compound types etc.). □

Consider an expression language with types `Num` (numbers), `Bool` (booleans) and `Str` (strings), and the following overloaded operators, in decreasing order of precedence:

- Hat (`^`): either exponentiation of numbers (e.g., `3^2`, which equals 9) or duplication of strings (e.g., `"ab"^3`, which equals `"abababab"`)
- Plus (`+`): either addition of numbers (e.g., `2+3`, equalling `5`) or concatenation of strings (e.g., `"ab"+"ac"`, equalling `"abac"`) or disjunction ("or") of booleans (e.g., **`true`**`+`**`false`**, equalling **`true`**)
- Equals (`=`): equality between any pair of values of the same type, always yielding a boolean.

This gives rise to the following grammar (never mind that it is not LL(1), that is not important right now):

1	$T \rightarrow T$ `^` T	6	$T \rightarrow$ `num`
2	| T `+` T	7	| `bool`
4	| T `=` T	8	| `str`
5	| `(` T `)`		

Exercise 3-CC.3 Answer the following questions for the language sketched above.

1. For each of the operators, sketch tables like the one in Fig. 4.1 of EC showing the result type of an expression given the type of its operands.
2. Give attribution rules for the type inference of the grammar, in the style of Figs. 4.5 and especially 4.7. (Note: the rules should express type inference, not computation of the result!)
3. Are your attributes synthesized or inherited?

□

To operationalize the attribute rules, we turn again to ANTLR. As an example, consider the following grammar, which defines yet another type of simple expression, with attribute rules that calculate the *value* of the expression that has just been parsed.

1	$E_0 \rightarrow E_2$ `*` E_1	E_0`.val` $\leftarrow E_1$`.val` $*$ E_2`.val`
2	| E_1 `+` E_2	E_0`.val` $\leftarrow E_1$`.val` $+$ E_2`.val`
3	| `(` E_1 `)`	E_0`.val` $\leftarrow E_1$`.val`
4	| `number`	E_0`.val` $\leftarrow$ `number.val`

`pp/block3/cc/antlr/Calc.g4` contains an ANTLR grammar for this language, and `CalcAttr.g4` is a variant of this grammar where the attribute rules have been implemented using explicit, built-in *actions*. (In reality this is an imperative solution, more like the ad hoc syntax-directed rules of Section 4.4.) Note the use of the method `getValue` in the rule for `NUMBER`: this is a user-defined method whose declaration is given in the **`@members`** block of the grammar. This piece of user-defined code is included in the `CalcAttrParser` class generated by ANTLR from the grammar definition.

On the other hand, the class `pp.block3.cc.antlr.Calculator` in combination with the grammar `Calc.g4` in the same package shows an alternative solution where the attribute rules are implemented through a *tree listener*, as already encountered in Block 2. In particular, the `vals`-field, declared to be of type `ParseTreeProperty<Integer>`, implements the required attribute, *not* by including an **`int`**-field in every node of the parse tree but through a *map* from parse tree nodes to `Integer`s.

The class `pp.block3.cc.test.CalcTest.java` is a JUNIT test for both solutions.

Exercise 3-CC.4 Study the action-based and listener-based solutions for the attribute rules.

1. Add a unary minus operator (i.e., negation, as in $1 + -2$) to both `Calc.g4` and `CalcAttr.g4` and extend the attribute rules and the `Calculator` class to cover this new case. Also extend the test (and make sure it runs).
2. What are advantages of the action-based solution, and what are advantages of the listener-based solution?
3. The subrule for `NUMBER` in `CalcAttr.g4` not only has an action *after* the evaluation of the right hand side, but also *before* evaluation (the `println`-statement). What happens if you also add such `println`-statements in front of the other subrules? Can you explain this effect?
4. What is the relation of the tree listener methods to the concept of synthesized versus inherited attributes? □

Exercise 3-CC.5 Give two different ANTLR grammars for the language of Exercise 3-CC.3 and implement your attribute rules for type inference in the following ways:

1. Define a grammar analogous to `CalcAttr.g4`, including direct actions for type inference. Use the provided **`enum`** type `pp.block3.cc.antlr.Type` to represent the inferred types.
2. Define a grammar analogous `Calc.g4` with a corresponding listener.
3. Program a JUNIT test for your grammars, analogous to `CalcTest` above, in which you show that type inference works correctly for all operators (including correctly flagging type errors).

□

3-CC.2 Listeners and symbol tables

Symbol tables The grammar `pp/block3/cc/symbol/DeclUse.g4` (provided in the lab files) contains the following parser rules:

```
program : '(' series ')' ;
series  : unit* ;
unit    : decl | use | '(' series ')' ;
decl    : 'D:' ID ;
use     : 'U:' ID ;
```

This defines a simple language with nested scopes, in which variables are *declared* (`D:var`) and *used* (`U:var`). An example program is

```
(D:aap (U:aap D:noot D:aap (U:noot) (D:noot U:noot)) U:aap)
```

The policies for declaration and use are:

- An identifier may only be used if it has been declared before, in the same scope or an enclosing one.
- An identifier may not be redeclared in the same scope.
- An identifier *may* be redeclared in an inner scope. This inner declaration temporally "overwrites" the previous (outer) one.

To keep track of declarations and uses, we need a data structure called a *symbol table*, which keeps track of nested scopes. An interface for this data structure is provided in `pp.block3.cc.symbol.SymbolTable`, and a test for it in `pp.block3.cc.test.SymbolTableTest`.

Exercise 3-CC.6 Program your own implementation of the `SymbolTable` interface, and make sure it passes the test. Use the information in the slides to make the best choice about the data structure to be used. □

To solve the next exercise, you have to read your input from a file rather than a string. This is actually straightforward: instead of `CharStream.fromString`, use the static method `CharStream.fromFileName` to create the character stream to pass into the lexer.

> *Take care!* If you import an ANTLR type into your JAVA code, there is often a choice between equally named classes from ANTLR v3 and ANTLR v4. In that case, you *always* have to choose the v4-variant. You can distinguish it by the fact that there is a `v4` somewhere in the package name.

Exercise 3-CC.7 Program a tree listener that checks the declare/use policy outlined above, for a given "program" in the `DeclUse` language, using your `SymbolTable`. Your solution should:

- Collect all errors it finds in a list of understandable error messages *including line and column number of the token that caused the error*. This information is available through methods `Token.getLine()` and `Token.getCharPositionInLine()`. (The class `ParserRuleContext`, which all `...Context` classes extend, has a number of methods to retrieve the tokens from which a given parse (sub)tree was constructed.)
- Be able to return this error list after having walked a parse tree.

Provide some sample (correct and incorrect) programs and a JUNIT test that shows that your solution gives the right answers. □

From LaTeX to HTML. LaTeX is a powerful typesetting language, which is used throughout the scientific world. It is especially useful for texts containing a lot of mathematics and formulas. Typesetting a document in LaTeX has many similarities to programming.

An example LaTeX feature is the `tabular`-environment. The `tabular`-environment can be used to specify and format tables. An example table specification and formatting is the following:

```
% An example to test the Tabular application.
\begin{tabular}{lcr}
  Aap  & Noot & Mies \\
  Wim  & Zus  & Jet  \\
  1    & 2    & 3    \\
  Teun & Vuur & Gijs \\
\end{tabular}
```

Amongst others, this example table can also be found in the lab files in `tabular-1.tex`. A basic description of the `tabular`-environment is as follows.

- The following characters are reserved keywords in LaTeX: `\ { } $ & # ^ _ ~ %`
- Whitespace is *not* ignored in LaTeX; see below for the treatment of whitespace.
- The symbol `%` starts a comment line; the comment runs to (and includes) the end of the line (similar to JAVA comments starting with `//`). Comments are ignored.
- The start and end of a `tabular`-environment are given by the strings `\begin{tabular}` and `\end{tabular}`. The string `\begin{tabular}` expects one argument between curly braces, which is a non-empty string of characters `l` (left), `c` (centered) and `r` (right) specifying column alignments. In this exercise, we will disregard the argument.
- The entries of a `tabular` are specified row by row. The end of a row is specified with a double backslash: '`\\`'.
- Each row consists of entries separated with an ampersand: '`&`'.
- All tokens discussed above (table *start*, table *end*, and the *row* and *column* separators) may be preceded and followed by whitespace characters; these are considered to be part of the token.
- Entries consist of a possibly empty sequence of non-special characters, optionally separated by (but not starting or ending with) spaces.

Exercise 3-CC.8 The first part of writing a `tabular` compiler is to create a parser:

1. Given the listed description above, what are suitable tokens for scanning LaTeX tabular environments?
2. Write an ANTLR parser for `tabular` environments. Test your grammar by making sure that the provided auxiliary test files `tabular-[1...5].tex` parse correctly. □

You will now write a tree listener that transforms a `tabular` parse tree into an HTML table. The grammar of HTML table documents is specified in Figure 3.1. For instance, the specification of `tabular-1.tex` above should be translated to

```
<html>
<body>
<table border="1">
```

Doc	→	<html> <body> *Table* </body> </html>
Table	→	<table border = "1"> *Row** </table>
Row	→	<tr> *Entry** </tr>
Entry	→	<td> *Text* </td>
Text	→	(any character except < and &)

Figure 3.1: Simplified grammar of HTML table document.

```
<tr>
  <td>Aap</td>
  <td>Noot</td>
  <td>Mies</td>
</tr>
<tr>
  <td>Wim</td>
  <td>Zus</td>
  <td>Jet</td>
</tr>
<tr>
  <td>1</td>
  <td>2</td>
  <td>3</td>
</tr>
<tr>
  <td>Teun</td>
  <td>Vuur</td>
  <td>Gijs</td>
</tr>
</table>
</body>
</html>
```

Before starting the output generation phase, we have to make sure the parsing phase encountered no errors.

ANTLR organises error reporting with a listener of type `org.antlr.v4.runtime.BaseErrorListener`. By default, every lexer and parser that ANTLR generates has an implementation of such a listener that only prints errors to `stderr`. To get more control over the error reporting process, you may remove any previous error listeners, and add your own implementation of `BaseErrorListener`.

```
Recognizer.removeErrorListeners()       // remove default reporting to stderr
Recognizer.addErrorListener(myListener) // add your own error listener
```

Finally, it's important to realise that ANTLR's scanning phase is interwoven with the parsing phase. Thus in general, you can not report all lexer errors before all parser errors.

Exercise 3-CC.9 The second part of writing a `tabular` compiler is to create an HTML output generator:

1. Investigate the method `syntaxError` of `BaseErrorListener`. What parameters does it have?
2. Write your own error listener that formats the error message in the same way as the default reporter (including line and column number and an indication of the offending symbol), but saves the errors in a list rather than sending them directly to `stderr`. Add the error listener to the lexer and parser as described above.
3. Build a tree listener for tabular parse trees that writes the corresponding HTML table to a file.
4. Add to your tree listener a `main`-method that accepts a file name, reads and parses it, and only calls the HTML-conversion if the parse tree is error free.
5. Test your result by trying it on the provided lab files `tabular-[1...5].tex`, and visually inspecting the resulting HTML in a browser. □

Block 4

Block 4

4-OV Overview

4-OV.1 Contents of This Block

CP This block will discuss liveness, performance and fairness of multithreaded applications.

FP In this block, we will deal with code generation in a functional way.

LP In this block, the Logic Programming (LP) strand will start. You will learn basic logic programming with Prolog (facts, rules, queries, backtracking, lists, data structures, recursion). This block LP contains four exercises:

- The Royal Family (§4-LP.1),
- The Monkey and Banana puzzle (§4-LP.2),
- The Treesort Algorithm (§4-LP.3).
- The Ice Cream Tour puzzle (§4-LP.4).

Note that there is the LP project in the next block.

CC This block will discuss several forms of intermediate representation: Control Flow Graphs, Dependence Graphs, ILOC with Static Single Assignment, and symbol tables.

4-OV.2 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 4 hours self-study to prepare the CP exercises;
- 4 hours self-study to complete the FP lab exercises;
- 4 hours self-study to complete the LP lab exercises;
- 2 hours self-study to complete the CC lab exercises.

Moreover, note that the CC homework exercises are due on the first day after this block.

4-OV.3 Materials for this Block

- **CP**, from JCIP: Chapters 10 and 11.
- **FP**, the lecture on parser combinators discussed code generation. Example code was provided in block 3.
- **CC**, from EC: Chapter 5.
- **LP**, slides from Blackboard, and documentation of SWI Prolog where necessary.

4-CP Concurrent Programming

Thread dumps. Section 10.1 and 10.2 of JCIP discuss several examples that can cause deadlocks under certain conditions. As explained in Section 10.2.2 of JCIP, JVM's thread dump mechanism can be used to investigate the source of deadlocks.

Exercise 4-CP.1 *Learning goal:* identifying causes of deadlocks.

Listing 10.1 contains a straightforward lock-ordering deadlock. Develop a multi threaded Java program which makes use of the class `LeftRightDeadlock` of Listing 10.1 and observe that it indeed may lead to a deadlock. Use Java's thread dump mechanism to analyse the source for the deadlock. Explain the thread dump information, and how you can derive the source of the deadlock from it. □

Performance optimization of a multi-threaded queue. Buffers are an important concept in concurrent applications, used to distribute work evenly. As they are so ubiquitously spread, they deserve some attention in order to make them as fast as possible and to minimize the chance that the buffer acts as a bottleneck under a very high amount of concurrent accesses.

☞ **Exercise 4-CP.2** *Learning goal:* understanding how implementation choice impact data structure performance.

Recall the `Queue` interface from Block 2.

```
public interface Queue<T> {
    /** Push an element at the head of the queue. */
    public void push(T x);

    /** Obtain and remove the tail of the queue. */
    public T pull() throws QueueEmptyException;

    /** Returns the number of elements in the queue. */
    public int getLength();
}
```

1. Start out with your *linked list* implementation of the `Queue` of block 2. You probably used Java's *monitor* pattern to make the implementation thread safe, i.e., declared the methods of the class as `synchronized` methods.
2. Update your test class of Block 2 to add timing measurements to your test framework. Also ensure yourself that your *linked list* implementation is still thread safe.
3. Implement the `Queue` interface with an `java.util.concurrent.ConcurrentLinkedQueue`. Minimize the use of (intrinsic or explicit) locks as much as possible, and ensure that your implementation is thread safe.
4. Implement the `Queue` interface with a `MultiQueue` class. In this `MultiQueue` class each producer has its own `linked list` queue. Consumers will try to obtain an element from any of the producer queues. Again, minimize the use of (intrinsic or explicit) locks as much as possible, and ensure that your implementation is thread safe.
5. Compare the three implementations on performance. How does your implementations of `Queue` scale with the amount of consumers and producers, and the number of insertions per producer. What causes the behavior that you observe? What are the bottlenecks of your implementations? Is there room for further improvement?

□

Copy-on-write. There are many scenarios in which the ratio between reads and writes of a data structure skews heavily in favour of reading. In these cases it might be advantageous to use a *copy-on-write* data structure. The principle of operation is that every *read* of information can be done directly. However, when you want to *write* to the structure the process is a bit more complex:

- Acquire the write lock.
- Make a *deep copy* of the main data structure into thread-local memory. This means that you also need to copy all other objects referenced by the data structure.
- Perform the write operation on the local object. For some structures (e.g., arrays) this might only modify a single object. For other, more complicated objects, more work might be involved. For example, a single write in a balanced tree might trigger a rebalance, thereby possibly affecting the entire tree.
- Replace the main structure by the updated version (i.e., with the write operation applied).
- Release the write lock.

Exercise 4-CP.3 *Learning goal:* understanding how choice of underlying data structures can impact performance of an application.

1. Implement a *copy-on-write* list. Implement at least the methods `add`, `set` and `get`.
2. Compare the performance of your implementation with a `Collections.synchronizedList` on the basis of an `ArrayList`. Compare the performance of your implementation with the class `CopyOnWriteList` from the Java concurrency library. Can you explain your results?
3. Why is there no `CopyOnWriteMap` present in the Java library?

□

Fairness in locks. Sometimes, an important requirement of a concurrent system controlled by locks can be that it is *fair*. The most common meaning of fairness is that every thread gets the chance to do its work eventually. This is a very weak notion of fairness and it should hold for every correct program (otherwise some threads would never be able to make progress). Another notion of fairness is that the thread that requests access first, will also be allowed access first (like a FIFO queue). It is a form of fairness but you could wonder whether it is really fair. What if thread `1` only spends a very short amount of time in the critical section per access but thread `2` spends a lot of time in the critical section per access. In the end thread `2` will have spent a lot more time there than thread `1`, mostly blocking thread `1` from access. Is this fair?

Other possible definitions of fairness are that every thread gets the same amount of time to execute code in the critical section, or that attempted entries with a special high priority flag get their access earlier than lower priority computations.

☞ **Exercise 4-CP.4** *Learning goal:* understanding different notions of fairness.

In this exercise, we will implement a fair lock, for your own notion of fairness.

1. Pick one of the previously mentioned (or come up with your own) definitions of a relatively strong form of fairness, i.e., not just that all threads eventually get to run their code.
2. Why is the fairness definition you picked fair? In what scenarios would it not be fair? Can you come up with an example scenario in which this fairness definition would be useful?
3. Modify one of the synchronizers you created in Exercise 3-CP.4 in order to make it behave according to your definition of fairness.

□

4-FP Functional Programming

4-FP.1 Seventh series: EDSLs and Code Generation

In this series of exercises we will work on instruction sets and code generation.

Exercise 4-FP.1 Code Generation.
Extract the file `FP_Core.hs` from the the .zip-file for block 4. This file contains a very elementary processor (`core`) that simulates the execution of a program written as a list of instructions. The instructions, as well as the "opcodes" for the arithmetical unit (`alu`), are defined as *algebraic types* (i.e., as "embedded domain specific languages").

Furthermore, the file `FP_Core.hs` contains:

- the function `alu` that executes the arithmetical operations,
- the function `core` that executes at every clock cycle one of the instructions from the list `instrs`, and updates the following elements:
  - the stack (`stack`),
  - the program counter (`pc`),
  - the stack pointer (`sp`),

  Note that `pc` refers to the instruction that is executed next, and `sp` points to the first free position "above" the top of the stack (even though on that position there still may be an "old" number).

In addition, the file `FP_Core.hs` contains the definition of the (recursive) algebraic type `Expr`, which expresses the tree structure of arithmetical expressions. Note that `Expr` also may be seen as an EDSL, for arithmetical expressions in this case.

To start, evaluate the expression `test` from the file `FP_Core.hs` to see its effect (`PutStr` is a standard **`IO`**-function in Haskell). Use `toRoseTree expr` to show the expression `Expr` as a tree. □

Exercise 4-FP.2 Write a function `codeGen :: Expr -> [Instr]` that generates the list of instructions of type `Instr`. This list of instructions, when it is processed by the function `core`, yields the result of the expression as the top element of the stack, i.e., as that element of the stack that is indicated by `sp - 1`. □

Exercise 4-FP.3 The stack represents a piece of memory that (by means of the instructions) only is accessible at the top. Add a second memory `heap :: [Int]` that is accessible at any address directly. Thus, the second argument of the function `core` becomes: `(pc, sp, heap, stack)`.

Rename the existing `Push`-instruction to `PushConst`, and extend the instruction set `Instr` and the functions `core` with the following instructions (i.e. data constructors):

- `PushAddr Int`: for pushing the value at the indicated address in the `heap` on the `stack`
- `Store Int`: to store the top value from the `stack` in the specified address in the `heap`. Furthermore, the value is removed from the `stack`.

Adapt and extend the definition of `core` accordingly. □

Please ask a TA to check Exercises 4-FP.2 and 4-FP.3

Exercise 4-FP.4 Add an embedded language for *statements* with initially only *one* statement (for *assignment*):

```
data Stmnt = Assign Variable Expr
           deriving (Show, Generic, ToRoseTree)
```

As always (in an imperative language), a *variable* refers to a location in memory (in the *heap*). You may choose for yourself whether you define the type `Variable` as an **`Int`** for the address in the heap to which the variable refers, or as a **`String`** for the name of the variable. In the latter case you will need to use a lookup table `lut` (say `x`, `y` are variables, and `a`, `b` are addresses in the heap) `[(x,a), (y,b), ...]` to bind a variable to the address to which it refers. Clearly, then you will also need a function to determine the address of a variable from this lookup table.

1. Extend the type `Expr` with a clause for variables.
2. Write a function `codeGen' ::Stmnt -> [Instr]` that generates code to execute the `Assign`-statement, such that the value of the expression is stored at the correct address in the heap. □

Exercise 4-FP.5 Define a typeclass `CodeGen` that contains a single function `codeGen`. Make the types `Expr` and `Stmnt` members (instances) of this class.

□

Please ask a TA to check Exercises 4-FP.4 and 4-FP.5

Exercise 4-FP.6 In this exercise you have to add a statement for *repetitions* to the type `Stmnt`:

- `Repeat Expr [Stmnt]`: the list of statements is executed a number of times as indicated by the expression.

To make implementing such statement possible, extend the instruction set (i.e., the type `Instr`) with the following instructions:

- `PushPC`: pushes the current program counter on the stack.
- `EndRep`: a limited jump-instruction at the end of the body of the Repeat-statement: it decreases the value of the expression by 1 (on the correct position in the stack), and changes the value of the program counter to the pc-value that was pushed onto the stack.

Extend the definition of `core` for these instructions, and extend the function `codeGen` for the `Repeat` statement. □

Exercise 4-FP.7 Show that your functions work correctly by writing a program (of type `[Stmnt]`, using the `Repeat` statement) that adds the numbers $1, 2, \ldots, 10$. □

Please ask a TA to check Exercises 4-FP.6 and 4-FP.7

4-CC Compiler Construction

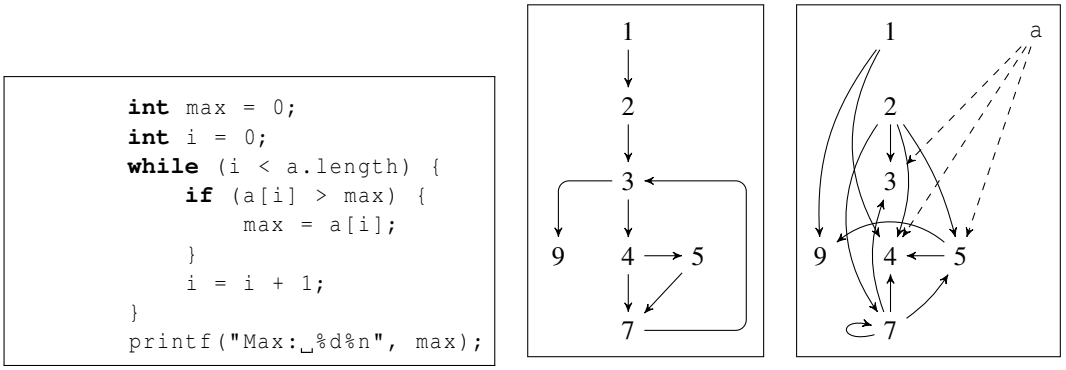
4-CC.1 Control flow and dependency graphs

Exercise 4-CC.1 Compile for yourself a list of the following terms, with short definitions for each of them. Don't just copy from the book: make sure you understand your own definitions.

AST (esp. difference with parse tree), DAG, basic block, CFG (two meanings!), dependence graph, call graph, SSA, symbol table

□

Consider the program fragment in JAVA in Figure 4.1(a), which finds the maximum element in an array `a` of non-negative integers. The control flow graph and dependency graph of this fragment are depicted as (b) and (c). The numbers in the figure correspond to the line numbers of the fragment. From the control flow graph, it can be seen that only 1 and 2 may be combined into a basic block, as all other nodes either have more than one incoming edge or more than one outgoing edge. From the data flow graph, it can be seen that within that block, statements 1 and 2 may be reordered, as there is no dependency between them.



(a) (b) (c)

Figure 4.1: Control flow (b) and dependency graph (c) of a JAVA fragment (a)

Exercise 4-CC.2 Consider the following code fragment:

```
1  int x = in();
2  int i = 0;
3  boolean found = false;
4  while (!found && i < a.length) {
5      found = (a[i] == x);
6      if (!found) {
7          i = i + 1;
8      }
9  }
10 printf("Index: %d%n", i);
```

1. Draw the control flow graph and the data dependency graph.
2. According to the control flow graph, what are the basic blocks of this fragment?
3. According to the data dependency graph, which statements can be reordered? □

Exercise 4-CC.3 Consider the following code fragment:

```
1  int up = in();
2  int sum = 0;
3  for (int i = 0;
4           i < up;
5           i = i + 1) {
6      if ((i & 1) == 0) {
7          continue;
8      }
9      sum = sum + i;
10 }
11 printf("Sum: %d%n", sum);
```

If you have never seen the **`continue`**-statement, look it up!

1. Draw the control flow graph and the data dependency graph.
2. According to the control flow graph, what are the basic blocks of this fragment?
3. According to the data dependency graph, can any statements be reordered? □

In the next exercise you will write a tree visitor to build control flow graphs. Some code has been provided in the package `pp.block4.cc.cfg`:

- `Graph.java`: a simple control flow graph
- `Node.java`: node of a control flow graph, with fields for ID and number.
- `Fragment.g4`: Grammar for a fragment of JAVA that will compile the code snippets above.
- `BottomUpCFGBuilder.java`: Unfinished template for Exercise 4-CC.4

- `TopDownCFGBuilder.java`: Unfinished template for Exercise 4-CC.4

Control flow graphs can be constructed through top-down, inheritance-based rules or bottom-up, synthesis-based rules. In either case, the idea is that every parse subtree adds control flow nodes and edges to a global graph (given as an instance variable), but there is a `ParseTreeProperty` associating the correct entry and exit nodes.

Bottom-up: This is the most straightforward way. Every `exit`-method of the listener can build a control flow graph, based on the CFGs of its sub-statements. The construction is very much like that of an NFA from regular expressions. For simplicity, make sure that you stick to the convention given in the book that every CFG has a single entry point and a single exit point; then you only have to store the entry and exit points as attributes in your listener. The choice of having a single exit point will sometimes necessitate the creation of "fake" control flow nodes that do not correspond to any statement but just serve as exit nodes of some sub-CFG. Thus, the graphs constructed in this way will not be identical to the hand-drawn ones of the previous exercises.

Top-down: This is more tricky. Every statement tells all its children the entry and exit nodes they should use. For this purpose, the `enter`-methods of the listener should prepare those nodes and set them as attributes for their children. The `exit`-methods are unused.

Exercise 4-CC.4 Now program both solutions. *You may omit the* ***`break`*** *and* ***`continue`*** *statements; they are addressed in a challenge exercise below.*

1. Program the bottom-up CFG builder (template provided in `BottomUpCFGBuilder`)
2. Program the top-down CFG builder (template provided in `TopDownCFGBuilder`)
3. For both builders, show that they return correct flow graphs for the code snippets of Figure 4.1 above and Exercise 4-CC.2 (note that you do not have to declare array `a`). It is expected that the graphs are not *identical* to your answers to those exercises. What are the differences? □

Challenge JAVA knows the concept of *abrupt termination*: this is what happens if you **`break`** out of a loop or **`continue`** within that loop, and also if you **`return`** anywhere except at the end of a method body or when an exception gets **`throw`**n. All such phenomena disrupt the normal flow of control (and most are actually frowned upon by code purists for that reason); to get the CFG correct for such cases takes special care.

Exercise 4-CC.5 *This exercise is meant as a challenge and does not have to be signed off.*

Extend your bottom-up or top-town CFG builder (or both) from Exercise 4-CC.4 with methods for **`break`** and **`continue`** statements (both of which are already in the provided grammar `Fragment.g4`). For this purpose, you should realise that in fact every type of abrupt termination essentially requires its own exit node; those exit nodes can be synthesized or inherited in essentially the same way as the "regular" exits. Test out your program on the the JAVA snippets of Exercise 4-CC.3 and on the following (which computes the same value as the snippet of Exercise 4-CC.2):

```
int x = in();
int i;
for (i = 0; i < a.length; i++) {
    if (a[i] == x) {
        break;
    }
}
printf("Index: %d%n", i);
```

□

4-CC.2 ILOC

Appendix A of EC contains a specification of the linear intermediate representation format ILOC used in the book. Examples of the use of ILOC are scattered throughout the book. For instance, Fig. 5.8(b) shows how an array element is assigned in ILOC, and Fig. 5.14 shows more complicated ILOC code fragment. The

JAVA snippet in Figure 4.1 is translated to the following ILOC code[1], in which `@a` denotes the offset of the beginning of array `a` from the address pointed to by `r_arp`, `@alength` denotes the length of the array `a`, and all other variables are kept in registers.

```
1  start: loadI 0 => r_max           // Line 1: max = 0;
2         loadI 0 => r_i             // Line 2: int i = 0;
3         loadI @alength => r_len
4  while: cmp_LT r_i,r_len => r_cmp  // Line 3: while (i < a.length)
5         cbr r_cmp -> body, end
6  body:  i2i r_i => r_a             // compute address of a[i]
7         multI r_a,4 => r_a         // multiply by size of int
8         addI r_a,@a => r_a         // add a's base offset
9         loadAO r_arp,r_a => r_ai   // r_ai <- a[i]
10        cmp_GT r_ai,r_max => r_cmp // Line 4: if (a[i] > max)
11        cbr r_cmp -> then,endif
12 then:  i2i r_ai => r_max          // Line 5: max = a[i];
13 endif: addI r_i,1 => r_i          // Line 7: i = i + 1;
14        jumpI -> while
15 end:   out "Max: ", r_max         // Line 9: out; not "official ILOC"
```

In the lab files you will find an assembler, a disassembler and a virtual machine for ILOC:

- `pp.iloc.Assembler` is an assembler. It has a singleton instance that; the method `parse` reads in ILOC programs in textual form (given as a string or in a file), such as the one shown above, and returns an object of type `pp.iloc.model.Program`.
- The method `prettyprint` of the class `Program` returns a string representation of the program, in the original syntax of ILOC.
- `pp.iloc.Simulator` simulates a `Program` by running it on a virtual machine (`pp.iloc.eval.Machine`) consisting of a simulated block of memory, a register map, and values for the symbolic constants `@a` etc.
- `pp.iloc.test` contains examples of the use of `Assembler` and `Simulator`.

Exercise 4-CC.6 Try out the assembler, disassembler and simulator on `max.iloc` by writing a JUNIT test with the following test methods:

1. A test method that runs the assembler on `max.iloc` and calls `prettyprint` on the resulting `Program` object. Test that, if you parse the prettyprinted program again, the resulting `Program` equals the one you got by parsing the original `max.iloc`.
2. A test method that run the resulting `Program` in the `Simulator`. The method `Simulator#run` starts off the simulation; however, before calling it you should initialize the VM (which you can access through `Simulator.getVM`) by initializing the array `a`, i.e., giving values to the constants `@a` and `@alength` using `Machine#init` and `Machine#setNum`, respectively. Test that you get the expected output by programmatically inspecting the value of the register `r_max` after the simulation has finished, using `Machine#getReg`.
3. Explain the multiplication by 4 in Line 7 of the program. □

In the ILOC program above, the *basic blocks* correspond to the fragments starting with a label.

Exercise 4-CC.7 Draw the CFG of the ILOC program above, using the basic blocks as nodes, and compare it to the one in Figure 4.1. □

Exercise 4-CC.8 Answer Review Question 1 of Section 5.4 of EC. Ignore the fact that `fib` is a function: just implement the function body.

1. Write an ILOC program implementing `fib` based on a register-to-register model.
2. Write an ILOC program implementing `fib` based on a memory-to-memory model.
3. Write a test that shows that both programs return the same value for a range of input values of your choice. (For this purpose, you have to appropriately initialize the VM and inspect the result afterwards.)

[1] Available in the lab files of Block 4 as `pp/block4/cc/iloc/max.iloc`

4. What is the largest value for `n` for which the programs do not produce an integer overflow? □

Exercise 4-CC.9 Write ILOC code for the JAVA fragment in Exercise 4-CC.5. (It does not matter if you did not do that exercise.) To mimic the `in()`-call in the JAVA program, use the pseudo-ILOC-instruction **`in`**; see `pp.iloc.OpCode.in`. (Like the **`out`**-instruction demonstrated in the ILOC code fragment, this instruction is not part of the "official" ILOC: it is only provided for debugging purposes.)

Make sure your solution passes the provided `pp.block4.cc.test.FindTest` (which assumes that your code can be found in `pp/block4/cc/iloc/find.iloc`; change that to the correct filename). □

Finally, in the following exercise you will do some actual code generation (of ILOC code). This will be the first real compiler you build! Use the attribute rules in Fig. 4.15 (Page 207) of EC as inspiration.

Exercise 4-CC.10 Extend `pp.block4.iloc.CalcCompiler`, a tree listener for parse trees produced by the grammar `pp/block4/cc/iloc/Calc.g4` (which is actually identical to `pp/block3/cc/antlr/Calc.g4` of the previous Block). `CalcCompiler#compile` should generate ILOC code (in the form of a `Program` object) for an arbitrary `Calc`-expression, using a register-to-register memory model, ending with an `out` operation to print the calculated value. For instance, for the expression $1 + -3 * 4$, the generated code should be something like

```
loadI   1               => r_0
loadI   3               => r_1
loadI   4               => r_2
mult    r_1,r_2         => r_3
rsubI   r_3,0           => r_4
add     r_0,r_4         => r_5
out     "Outcome: ",r_5
```

Make sure your solution passes the `pp.block4.cc.test.CalcCompilerTest`. □

4-LP Logic Programming

4-LP.1 Royal Family: Facts, Rules and Queries

If you have not done so: Install SWI Prolog, following the instructions under Tools/Installing Prolog, and find the program `royal_family_facts.pl` on Blackboard (lab files Block 4). You can load this program from the Prolog prompt by: `?- consult(royal_family\facts).`

Exercise 4-LP.1 Start a new program `royal_family_defs.pl` and enter the rules from the lecture for `child/2`, `grandparent/2`. Test if this works. □

Exercise 4-LP.2 Provide precise definitions of several family relations, like: `brother/2`, `aunt/2`, `cousin/2`, `nephew/2`. □

Exercise 4-LP.3 Test your definitions with a number of test instances. Explain the answers, as well as the number of repetitions that you see. □

Exercise 4-LP.4 Provide your own definition of ancestor/2. Show that this definition might loop, by modifying the facts database `royal_family_facts.pl`.

4-LP.2 Monkey gets Banana: Functions, Backtracking, Lists

The "Monkey gets Banana" problem is modeled as `state(Monkey,Pos,Box,Has)`, where `Monkey` and `Box` indicate the location of the monkey and the box in the room, respectively (at the door, at the window, or in the middle); `Pos` indicates the position of the monkey (on the box or on the floor) and `Has` indicates if the Monkey has already got the banana.

Exercise 4-LP.5 Define a predicate `goal/1` that is true for all states where the Monkey has the Banana. □

Exercise 4-LP.6 Define a predicate `init/1` that is true for the initial state, where the Monkey is on the floor at the door, without having the banana, and the box is at the window. □

Exercise 4-LP.7 Define the move-predicate `move/3`, where `move(State1,Move,State2)` indicates that action `Move` brings the monkey from `State1` to `State2`. The possible moves are:

1. *climb*: If the monkey and the box are in the same location, and the monkey is on the floor, it can climb on top of the box.
2. *grasp*: If the monkey and the box are in the middle and the monkey is on the box without a banana, it can grasp for the banana and get it.
3. *walk(L1,L2)* If the monkey is in location L1 on the floor, it can walk to position L2
4. *push(L1,L2)* If the monkey and the box are in location L1 and the monkey is on the floor, it can move with the box to location L2.

□

Exercise 4-LP.8 Write a general problem solver `solve/1`, which can use `move/3` and `goal/1`.

`:- init(S), solve(S)` shall check if the goal can be reached from the initial state by a number of moves. □

Exercise 4-LP.9 Experiment with the order of the `move/3` clauses in the program. How does the order affect termination of the program? □

Exercise 4-LP.10 Extend the solver `solve/2` to return a solution: `init(S), solve(S,L)` should return a list of moves L, e.g.:

```
yes. S=... and L=[walk(atdoor,atwindow),climb,...].
```

Exercise 4-LP.11 *Optional:* Extend the solver to print intermediate states, so you can get some insight in Prolog's search and backtracking. You can use the built-in predicate `print/1` to print a term, and `nl/0` to print a new-line. □

4-LP.3 Tree Sort: More on lists, Data structures, Recursion

A binary tree is a datastructure in which each node has a value and two children. In this assignment, we define a binary tree as either `nil` or `t(L,N,R)`, where `nil` denotes the empty tree and `L` and `R` are trees again, called the left subtree and the right subtree respectively; and `N` is a term. You may suppose that `N` always is a number, without checking this explicitly.

A binary tree is sorted if for each node (subtree) `t(L,N,R)` in the tree:

- The left subtree `L`, if not nil, has a maximum smaller than or equal to the value `N`; and
- The right subtree `R`, if not nil, has a minimum greater than or equal to its own value `N`.

The idea is to make exercises 4-LP.12 – 4-LP.20, and signoff only exercise 4-LP.20 with a practicum assistant. The last two exercises 4-LP.21 – 4-LP.23 are optional, for those who like an extra challenge.

Exercise 4-LP.12 Write a predicate `istree(T)` that evaluates to true iff T is a binary tree. Example:

```
?- istree(
   t( t(nil,2,nil),
      4,
      t(  nil,
          5,
          t(nil,18,nil)
 )  )  ).
Evaluates to True.
```

```
?- istree(t(t(2,2),4,nil)).
Evaluates to False.
```

□

Exercise 4-LP.13 Write the predicate `max(T,N)` and `min(T,N)` that find respectively the maximum value and minimum value of a sorted binary tree. Example:

```
?- min(t(t(t(nil, 1, nil), 2, nil), 3, nil), R).
R=1
?- max(t(t(t(nil, 1, nil), 2, t(nil, 8, nil)),
                        18, t(t(nil, 21, nil), 81, t(nil, 218, nil))),R).
R = 218
```

□

Exercise 4-LP.14 Write a predicate `issorted(T)` that evaluates if the argument of the predicate is a sorted binary tree (see the explanation above for a definition). Example:

```
?- issorted(t(t(nil,4,nil),5,t(nil,8,nil))).
true
?- issorted(t(t(nil,2,nil),1,nil)).
false.
```

□

Exercise 4-LP.15 Write a predicate `find(T, N, S)` that finds a node with a specific value N in a sorted binary tree T and returns the corresponding subtree in S. It fails if such a node does not exist. Example:

```
?- find(t(t(nil,1,t(nil,3,nil)),5,t(t(nil,6,nil),8,nil)),1,S).
S = t(nil, 1, t(nil, 3, nil)).
```

□

Exercise 4-LP.16 Write a predicate `insert(T, N, S)`, which inserts a value N in the sorted binary tree T, giving sorted binary tree S. Example:

```
?- insert(t(t(nil,1,t(nil,3,nil)),5,t(t(nil,6,nil),8,nil)),7,S).
S = t(t(nil, 1, t(nil, 3, nil)), 5, t(t(nil, 6, t(nil, 7, nil)), 8, nil)).
```

□

Exercise 4-LP.17 Write a predicate `deleteAll(T, N, S)` that deletes one value N from a sorted binary tree T, giving sorted binary tree S. Hint: Use the result of assignment 2. It might be easier to use an auxiliary function that deletes only one element. Example:

```
?- deleteAll(t(t(nil, 1, t(nil, 3, t(nil,3,nil))), 5, t(t(nil, 6, nil), 8, nil)),3,S).
S = t(t(nil, 1, nil), 5, t(t(nil, 6, nil), 8, nil)).
```

□

Exercise 4-LP.18 Write a predicate `listtree(L,T)` where input L is a list of integers, and output T is a sorted binary tree with the same values. Hint: it might be helpful to use `insert`. Example:

```
?- listtree([8,3,10,2,11,7,1],T).
T = t(t(t(t(nil, 1, nil), 2, nil), 3, t(nil, 7, nil)), 8, t(nil, 10, t(nil, 11, nil))).
```

□

Exercise 4-LP.19 Write a predicate `treelist(T,L)` where input T is a sorted binary tree and output L is a sorted list of unique integers with the values from T. Hint: it might be helpful to use `deleteAll` Example:

```
treelist(t(t(nil,1,nil),2,t(nil,8,nil)),L).
L = [1, 2, 8]
?- treelist(t(t(nil,1,t(nil,3,nil)),3,t(t(nil,6,nil),8,nil)),L).
L = [1, 3, 6, 8].
```

□

Exercise 4-LP.20 Write a predicate `treesort(L1,L2)` which succeeds if output L2 is the sorted permutation of input L1. The sorting algorithm should use a binary sorted tree. Test the algorithm with a number of test instances. What happens if L1 contains duplicate entries? Explain why this happens.

Optional Extension

A binary tree is balanced if, for each node, the number of nodes of the left subtree and the number of nodes of the right subtree differ at most 1. We only use this concept in the optional excercises.

Exercise 4-LP.21 *(optional).* Write the predicate `listtree_balanced` such that the resulting tree is balanced. Use this to write `balanced_treesort`. Compare the complexity of the two sorting algorithms by measuring the runtime on a number of larger examples. □

Exercise 4-LP.22 *(optional).* Write a predicate to write a tree graphically to your screen.

Example (but you can decide on another tree-layout yourself):

```
listtree_balanced([8,3,10,2,11,7,1],T), draw(T).
          7
         / \
        /   \
       /     \
      /       \
     2         10
    / \        / \
   /   \      /   \
  1     3    8    11
T = t(t(t(nil,1,nil),2,t(nil,3,nil)),7,t(t(nil,8,nil),10,t(nil,11,nil))).
```

□

Exercise 4-LP.23 *(optional).* Try to run `insert` in reverse. Is it possible to get `delete` for free? Similar, can the computation of `treelist` be obtained as the inverse of `listtree`? Maybe after modifications to the program? □

4-LP.4 Ice Cream Tour

The Ice Cream tour is an instance of the (in)famous logic puzzles. The goal is to model this puzzle, and solve it by using the strength of Prolog's unification and backtracking. Note that backtracking over all possible solutions might be too slow, so the preference is to solve parts of the puzzle by just unification. We start with an "empty" solution and gradually fill it in by "matching" the partial solution with new facts. Sometimes, backtracking might be unavoidable.

Description

On his bike tours around in Twente, Sherlock realized that he could hit four ice cream stands. Each stand is near a different castle along his tours and the name of each stand was a different person's first name. Sherlock loved ice cream and so felt compelled to try a different one on his way home each night for the rest of the week. Determine the name of each ice cream stand, what castle each stand was close by, what day of the week he stopped at each, and what kind of ice cream cone he ordered at each one.

Here are the concrete clues:

1. Arend's Ice Cream wasn't near Twickel. Sherlock didn't get peppermint stick ice cream on Thursday night.
2. He got coffee bean ice cream on Wednesday, but not at Marieke's Ice Cream.
3. At Jaco's Ice Cream he got peanut butter ice cream but not on Tuesday.
4. Sherlock stopped at the ice cream stand near Westerflier on the day before he got the chocolate chip ice cream and the day after he stopped at Marco's Ice Cream.
5. He stopped at the stand near Weldam the day before he stopped at Arend's Ice Cream.

Actually, I can add one more piece of information. I called Sherlock last night, and he told me that:

6. The ice he got near Espelo was gorgeous as well...

Exercise 4-LP.24 Let Prolog reconstruct Sherlock's ice cream tour, based on the clues above. Use the power of unification, resolution and backtracking. Make sure your program has a predicate `go/1`, such that `:- go(X)` generates the solution(s) to this puzzle.

Additional question: Does this puzzle actually have a unique solution? □

In principle, you are completely free how to model and solve this problem in Prolog. However, since the time for the Prolog project should be limited, some hints follow that might push you in some direction (hopefully saving time, but, unfortunately, also pruning some creativity). Feel free to diverge from this path!

1. You can model the problem as finding the ice-cream tour as an (ordered) list of the form:

```
[ stand(tuesday,_,...,_),
  stand(wednesday,_,...,_),
  stand(thursday,_,...,_),
  stand(friday,_,...,_) ]
```

   The list order is relevant to model the clues with "before" and "after".

2. Proceed as follows:
   (a) Figure out what fields are needed in the stand-structures in that list.
   (b) Write a predicate `icecream(Tour)`, constraining Tour by all clues.
3. Some last hints:
   (a) Write auxiliary functions on lists to express the before-after constraints conveniently;
   (b) Be careful with negative information: the negative statements contain positive information as well;
   (c) Remember that at the moment Prolog evaluates **`not`**`(P)`, the variables in `P` are supposed to be instantiated.
4. Don't forget to wrap everything in a single `go(X)` predicate.

Wish you ice-cream weather and a lot of fun!

Block 5

Block 5

5-OV Overview

5-OV.1 Contents of This Block

CP This block will discuss a concurrency model for homogeneous threading, where each thread executes the same instructions. We look at two programming techniques to support this: via compiler directives about possible parallellisations of a sequential program; and using dedicated hardware that natively supports this model, namely graphics processing units (GPUs).

FP There is no new material in this block.

CC This block will discuss issues related to procedure calls and memory layout: how should local variables, global variables and parameters be stored and accessed so as to achieve the well-known call mechanism?

LP This block has a last lecture on advanced features of Prolog, like meta-programming and constraint solving. There are no additional lab exercises.

Note that the LP project starts and ends this block!

5-OV.2 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 4 hours self-study to prepare the CP exercises;
- 2 hours of unsupervised work on the FP project.

5-OV.3 Materials for this Block

- **CP**, The material for this week consists of several papers and book sections:
  - The text on dependencies in this manual (5-CP.1).
  - Barbara Chapman, Gabriele Jost and Ruud van der Pas. Using OpenMP – The Book. In any case, read Sections 4.3 (except 4.3.4), 4.4, 4.5 (except 4.5.3 and 4.5.4), 4.6.1, 4.6.3, 4.8.2, and 4.8.4. You can ignore all the bits about Fortran syntax. If you want to know more, Chapters 1 - 4 should give you the full basis.

  - The text on GPU programming in this manual (Section 5-CP.3).
  - Matthew Scarpino, *A Gentle Introduction to OpenCL*. Dr.Dobb's, August 2011.
  - For more information on OpenCL, see `http://streamcomputing.eu/knowledge/for-developers/tutorials/`.

  Links to the papers are available on Canvas.
- **CC**, from EC: Chapter 6.
- **LP**: slides from Canvas.

5-CP Concurrent Programming

During this course you have seen a variety of methods for employing parallelism and concurrency in your programs. Up till now, everything has been based on heterogeneous threading, i.e., each thread executes its own code, and does its own updates on the shared data. However, there are large classes of problems where a homogeneous threading model can be used, i.e., each thread executes the same set of instructions on its own set of data. This execution model is usually called SIMD: *Single Instruction Multiple Data*. During this block, we look at two concurrent programming techniques that support this execution model.

- OpenMP is a library which adds compiler directives to a sequential program. The compiler directives indicate places where the program may be parallellised. This kind of programming is often called *structured parallel programming*.
- OpenCL is a programming language for graphic processing units, where the hardware model supports the homogeneous threading model.

5-CP.1 Program Dependencies

To understand why certain parallellisations are not possible, we introduce the notion of program dependency. If we have two statements

```
x = 3 * y;
z = 2 * x;
```

then we say that there is a *dependency* from `x` on `y`, and a dependency from `z` on `x`, because the values of `x` and `z` depend on the values of `y` and `x` (and thus indirectly, `z` also depends on `y`).

This notion of dependency is in particular important for the parallellisation of loops. Consider the example programs in Figure 5.1. In the first loop (Figure 5.1 (a)) all iterations are independent of each other. If they are executed in a different order, it will not change the outcome of the loop. In the second loop (Figure 5.1 (b)) there is a *forward dependency* between the iterations. The behaviour of the $i+1^{th}$ iterations depends on the result of the i^{th} iteration. Finally, Figure 5.1 (c) shows a *backward dependency* between the iterations of the loop: the behaviour of the i^{th} iterations depends on the result of the $i-1^{th}$ iteration.

When we try to parallellise a program, we have to make sure that we do not break the dependencies between instructions, because this would change the behaviour of the program. Clearly, loops with independent iterations can be parallellised without problem. Loops with forward dependencies can be

```
for (int i=0;i<N;i++) {
  a[i] = 3;
}
```

(a)

```
for (int i=0;i<N;i++) {
  a[i] = a[i] + 4;
  if (i > 0) {
    c[i] = a[i-1] + 2;
  }
}
```

(b)

```
for (int i=0;i<N;i++) {
  a[i] = a[i] + 4;
  if (i < N-1) {
    c[i] = a[i+1] + 2;
  }
}
```

(c)

Figure 5.1: Loop dependencies

parallellised if we make sure the instructions are executed in lock step, or we insert some appropriate synchronisation between the two steps. Loops with backward dependencies cannot be parallellised, and have to be executed sequentially.

5-CP.2 Compiler Directives for Parallellisation

The following questions use the VM provided on Canvas. Be sure to download and run it. For more information regarding the VM, please refer to the PDF about the VM on Canvas.

In the following exercises you will be asked to implement a parallel version of a sequential kernel. It is recommended to first copy-paste the sequential version and start from there. The implementations are in the files `kernels_*.c`:

1. `kernels_sequential.c`: provided sequential implementations
2. `kernels_openmp.c`: skeleton for code to add compiler directives to
3. `kernels_opencl.c`: skeleton for some OpenCL kernels

With all exercises, have a look at the impact your changes have on performance. Try to find out out why the parallel version is faster or why it is slower; there are many factors involved.

When you run the program, all implementations are checked for correctness and benchmarked. When a kernel exhibits differing behaviour, the difference is printed. For large inputs, the output is minimal. You may comment out some tests in `main.c` when you want to run only a particular set of tests.

☞ **Exercise 5-CP.1** *Learning goal:* experiment with different parallellisation directives.

Consider the implementation of a naive sequential matrix multiplication provided on Canvas (`impl_matrix_multiply_sequential`). The performance of this sequential program is cubic (n^3, where n is the size of the matrix). However, the program is very suitable for parallellisation.

Use OpenMP compiler directives to try out various different ways to parallellise this program (i.e., parallellising a single loop, two loops etc.) and compare the performance of these different versions on various examples. Implement this in the function `impl_matrix_multiply_openmp`.

Which conclusions can you draw from these experiments? □

☞ **Exercise 5-CP.2** *Learning goal* Understanding reductions

Consider `impl_count_zero_sequential`, a function to compute the number of zero elements of a matrix.

Use OpenMP directives to parallellise this function, without changing their input-output behaviour in the function `impl_count_zero_openmp`. □

☞ **Exercise 5-CP.3** *Learning goal:* Obtaining more experience with program paralllellisations.

This exercise is based on Challenge 3 of the VerifyThis 2017 program verification competition. The function `impl_vector_sort_sequential` is a sequential version. Use OpenMP compiler directives to parallellise this program in the function `impl_vector_sort_openmp`.

□

5-CP.3 General Purpose GPU programming (GPGPU)

As mentioned above, OpenMP supports homogeneous threading by allowing the developer to add compiler directives about possible parallellisations of the program. An alternative approach is to use hardware that is especially dedicated to this programming style, and to write programs that are explicitly targeting such hardware. This is the approach that we will study now.

The specific hardware that we are targeting are Graphical Processing Units (GPUs). Originally, GPUs were developed to have fast built-in support for graphics. For this purpose, GPUs have a large number of threads that can be executed in parallel. However, this model can also be efficiently used for general purpose programming, and that is what we will consider here. (Many of the considerations for general purpose GPU programming also apply to specific graphics programming.)

A GPU architecture consists of multiple *work groups*, where each work group consists of multiple threads. All threads within a workgroup execute the same instruction simultaneously. However, by using

variables such as the thread identifier in the instruction, we can make sure that every thread executes its instruction on its own part of the (shared) data. For example, suppose we have an array of length 1024, and we have a workgroup with 1024 threads. If each thread executes one instruction

```
a[tid] = 0;
```

where `tid` denotes the thread identifier, then afterwards all elements in the array are set to 0.

This programming model is already very old, and is sometimes referred to as *vector programming*. Some hardware has built-in support for vector instructions, which can execute this kind of program natively. With the advent of GPUs, we have obtained hardware that is fully dedicated to vector programming.

Programming languages for GPUs thus support this kind of programming. In this course, we look at OpenCL. OpenCL is an *extended subset of C*, i.e., it uses C syntax, provides a few extra GPU-specific constructs, and disallows features of C that make programming in this model complicated (in particular, disallowing things such as pointer arithmetic).

The code that is executed on a GPU is called a kernel. In essence, the code in a GPU kernel is the same as the code snippet above, setting all elements in the array to 0 (i.e. `a[tid] = 0;`. The same kernel as an OpenCL program would look as follows (where `get_global_thread_id()` is the OpenCL version of the `tid` keyword.

```
__kernel void (__global float* input, __global float* output){
{
    int tid = get_global_id(0);
    a[tid] = 0;
}
```

In addition to kernel code, an OpenCL program also consists of hostcode. This is executed on the CPU, and initialises and invokes the kernel. This is standard C code, and we do not discuss that further here (there is a lot of boilerplate code necessary in the host code, see the DrDobbs tutorial, and the template provided on Canvas).

Memory As the OpenCL kernels are so close to hardware, we have to know a bit more about the hardware model to make the code run efficiently. Note that this is in fact independent of the concurrent behaviour; it is just necessary to understand how OpenCL kernels are set up. As mentioned above, a GPU contains work groups, and each work group consists of a (large) number of threads. A work group executes all its threads in parallel. Different workgroups might be executing in parallel.

Each thread has a small private memory, to store local variables etc. All threads in a workgroup share a common memory (which is called *local memory*), to which they have a very fast access. Finally, there is a *global memory* which is accessible to all work groups, and also to the CPU. Access to the global memory is *slower*, thus a typical optimisation pattern that one often sees in a GPU kernel is the following:

1. The CPU copies the data that is required for the kernel to global GPU memory.
2. The kernel is invoked.
3. The kernel first copies the data to local GPU memory.
4. Then the computations are done, reading and writing from local GPU memory (and possibly using the thread-only memory).
5. When all threads are done, the kernel copies the result back to global GPU memory.
6. The kernel terminates.
7. The host retrieves the result from global GPU memory.

However, notice that it depends on the actual program whether this is indeed an optimisation. If a problem is solved by calling multiple kernels sequentially, or even iterating kernel invocations, it might be faster to leave all data in global memory.

Notice that because of this layout of the hardware, thread identifiers also come in two flavours. The function `get_local_thread_id()` returns the thread identifier within the work group; the function `get_global_thread_id()` returns the global, overall thread identifier of a thread. If data is accessed by threads within multiple work groups, this distinction is important (but for this course, the example kernels will be small, and this difference will be irrelevant).

Synchronisation When writing kernels, we cannot make any assumptions about when instructions are executed. In particular, suppose we want each thread to execute the following two instructions:

```
a[tid] = 3;
b[tid] = a[(tid + 1) % nr_of_threads];
```

The second instruction requires that the first instruction for the neighbouring thread has finished. If this is not guaranteed, a data race might happen, because thread `tid + 1` can write to `a[tid + 1]`, while thread `tid` is reading it. The execution model of the GPU hardware does not guarantee this (typically, a number of threads is executed in lockstep, but usually this is a subset of the threads in the work group, and the size of this set is hardware-specific).

Fortunately, OpenCL provides a way to synchronise threads, by means of a `barrier` instruction. If we put a barrier between the two instructions

```
a[tid] = 3;
barrier(CLK_LOCAL_MEM_FENCE)
b[tid] = a[(tid + 1) % nr_of_threads];
```

then all threads will execute their code upto the barrier, and only continue once all other threads have also reached the barrier. Moreover, when threads reach a barrier, they make sure to flush all their updates to the shared memory, thus making them visible to other threads. As there are two kinds of memories for GPUs, barriers can be set with flags to determine whether data will be flushed to local and/or global memory. In the example above, data will be flushed to local memory. Flushing to local memory only will be faster than also flushing to global memory.

Notice that in a typical kernel application, before the results are written to global memory, all threads will synchronise with a barrier (on local memory) and then continue copying the data to global memory.

However, barriers can only synchronise threads within a single work group, and there is no way to synchronise all threads within a kernel.

If communication and synchronisation between work groups is necessary, the only way to encode this is by means of *atomic operations*, which provide asynchronous updates on shared memory (either local or global). We do not go into full details here, but explain this by means of an example. The following kernel adds up all elements in an array. First each workgroup computes the sum of a subsequence of the array, and when all threads in the workgroup are done, one thread in the workgroup (with local thread id 0) adds this contribution to the shared result `r` in global memory.

```
atomic_add(x, values[gtid]);
barrier(CLK_LOCAL_MEM_FENCE);
if (ltid ==0) {
  atomic_add(r, x);
}
```

Currently, two mainstream ways of programming GPUs exist, namely CUDA and OpenCL. NVIDIA has its proprietary CUDA (Compute Unified Device Architecture) programming toolbox. However, as it is proprietary, it only runs on NVIDIAs own CUDA-enabled GPUs. In contrast, OpenCL is not associated to any specific vendor and therefore – as long as the manufacturer of the device supplies the proper library to link to – you will be able to compile your OpenCL program to run on every OpenCL-capable *compute device*. Notice that *compute devices* are not only limited to GPUs, but also include the CPU. Therefore, if OpenCL is not able to find a suitable OpenCL-capable GPU, your program can also run on the CPU. Hence, the main advantage of OpenCL is cross-'compute device' compatibility. Therefore, this course uses OpenCL instead of CUDA. Nevertheless, the main programming principles for CUDA are identical.

☞ **Exercise 5-CP.4** *Learning goal:* writing a basic OpenCL kernel.

Implement an OpenCL kernel with the same behaviour as the function `impl_count_zero_sequential`, considered in Exercise 5-CP.2. □

☞ **Exercise 5-CP.5** *Learning goal:* understanding how to use massive parallellism to improve performance, and understand synchronisation in OpenCL.

Sum. Write an OpenCL kernel which takes an array of numbers and computes the sum of all numbers in logarithmic time complexity. □

Exercise 5-CP.6 *Learning goal:* writing a more advanced OpenCL kernel.

Making Change. Many different puzzles that look rather complex can be easily solved using dynamic programming techniques. The problem is broken into smaller sub-problems and the solutions of the sub-problems are combined together. Moreover, when the computations of the sub-problems are independent, they are suitable to be executed in parallel. These programs can take advantage of GPU programming: if the work is properly divided between the GPU threads, this can result in significant performance optimization.

The *Making Change Problem* is an example of such a dynamic programming problem. We are given n coins with values $v_1, v_2, v_3, \ldots v_n$, such that: $1 \leqslant v_1 \leqslant v_2 \leqslant v_3 \ldots \leqslant v_n$. For a given amount of money S, the goal is to compute how to make this amount of change using as few coins as possible.

Input parameters:

`coins[1..n]` an array of `integer` values, each of them greater then `0`

`S` an `integer` representing an amount of money

Return value: a number representing the minimum number of coins required to make change for the amount of money `S`. If it is not possible to make this amount with the given set of coins, then the returned result is `-1`.

Examples

- Given `coins == [1, 1, 2, 2, 3, 5]` and `S == 9`, the result will be `3`.
- Given `coins == [2, 2, 4, 6]` and `S == 15`, the result will be `-1`.

Implement a OpenCL kernel to solve this problem. □

5-FP Functional Programming

There is no new material for Block 5.

5-CC Compiler Construction

5-CC.1 ANTLR tree visitors

So far, we have used ANTLR to define *tree listeners*. You have seen that a tree listener combines a pre-order and a post-order traversal of the parse tree. In some cases, however, this is not powerful enough: it can happen that one needs an in-order traversal (where the parent node is visited after the first child or in between every pair of children) or even another, more dedicated traversal strategy. For this purpose, ANTLR also supports *tree visitors*. Tree visitors require a bit more work from the programmer but offer more control: essentially, they allow you to completely define your own traversal strategy.

The lab files for this block include a grammar `Building.g4` to illustrate the principle of tree visitors. To generate a tree visitor for this grammar, invoke ANTLR on `Building.g4` with the command-line arguments `-no-listener -visitor`. (See BLACKBOARD for directions on how to do this.) This will generate classes `BuildingVisitor` and `BuildingBaseVisitor` rather than the `BuildingListener` and `BuildingBaseListener` we have been working with so far.

In contrast to `BuildingListener`, the tree visitor interface `BuildingVisitor` provides not two but a *single* abstract method for every nonterminal: the dedicated method for a nonterminal `x` is called `visitX`. As parameter, each such method gets the same sort of context object as the `enter-` and `exit`-methods of a listener. Moreover, `BuildingVisitor` has a generic type; this is the return type of every `visitX`-method. To implement a visitor, extend `BuildingBaseVisitor` and override its `visitX`-methods, taking care of the following:

- Each `visitX` should call the global method `visit` on all children that need to be visited, in the order they need to be visited.
- In between the calls to `visit`, each `visitX` can do whatever it likes.

- Each `visitX` method should return a value of the instantiated generic type.

Exercise 5-CC.1 *This exercise does not need to be signed off; it serves as a preparation for the next ones.* Study the provided `Building.g4` and `Elevator.java`

1. Generate the `Building` visitor classes using ANTLR.
2. Invoke `Elevator.java` on input of your choice. Study the `visitBuilding` and `visitFloor` methods and make sure you understand what they do. Note that `visitRoom` is never called (and hence it is not overridden in this class). □

Next, you'll program your own tree visitor. Consider the situation where an input string consists of an alternating sequence of numbers and words, such as for instance the text

```
4 strands 10 blocks 11 weeks 15 credits
```

and we want to print this on the standard output in the more readable form

```
4 strands, 10 blocks, 11 weeks and 15 credits
```

while simultaneously calculating the sum of the numbers occurring in the sentence (in this case, the sum would be 40). In the lab files, you will find a grammar `NumWord.g4` that can parse the input: the parser rules are

```
sentence: (number word)+ EOF;
number: NUMBER;
word: WORD;
```

(where `WORD` and `NUMBER` are token types with the expected definition).

Neither a preorder traversal nor a postorder traversal provides a natural way to print the required separators ", " and " and " between the groups of `number word`-pairs, so using a tree listener on this grammar it is not straightforward to program the desired behaviour. (It is not *impossible*: the exit method for `word` may inspect its parent and depending on whether that is a `sentence` and on the position of itself within that sentence print the correct separator.)

Exercise 5-CC.2 Implement the tree visitor `NumWordProcessor.java` (a skeleton of which is provided in the lab files) by overriding the `visitSentence`, `visitNumber` and `visitWord` methods so that the class has the required behaviour, described above (print the converted sentence, and return the sum of the numbers). You can test your `NumWordProcessor` by invoking the (provided) `main` method. □

Optional exercise: Faking the tree visitor. Alternatively, the functionality of a visitor can be faked through a listener by adding auxiliary intermediate nonterminals to the grammar, which are traversed by the listener at the right moments. For instance, the following rules, provided in the lab files in the form of the grammar `NumWordGroup`, generate the same language as `NumWord` above:

```
sentence
    : (group* penultimateGroup)? lastGroup EOF;

group
    : number word;

penultimateGroup
    : number word;

lastGroup
    : number word;

number: NUMBER;
word: WORD;
```

Here, the nonterminals `group`, `penultimateGroup` and `lastGroup` (which actually have exactly the same right hand sides) are defined with the sole purpose of allowing an "ordinary" tree listener to distinguish the right places in the sentence for the different separators.

Exercise 5-CC.3 *(This is an optional exercise and does not have to be signed off.)* Implement the tree listener `NumWordGroupProcessor.java` (a skeleton of which is provided in the lab files) by overriding the appropriate `enter-` or `exit-` methods so that the class has the required behaviour. Again, you can test your `NumWordGroupProcessor` by invoking the (provided) `main` method. □

Exercise 5-CC.4 *This is a follow-up to the optional Exercise 5-CC.3. It is meant for you to form your own opinion about the different solutions in Exercises 5-CC.2 and 5-CC.3; it does not have to be signed off.*

1. What advantages can you see of the visitor-based solution over the listener-based solution?
2. What advantages can you see of the listener-based solution over the visitor-based solution? □

Generating ILOC for control structures. We'll now combine a lot of the ingredients from Block 4 to create an ILOC code generator for a language that starts to look like you could actually do something with it. The provided grammar `SimplePascal.g4` defines a non-trivial fragment of the programming language `Pascal`, with the following features:

- The language supports integer and boolean types. On the ILOC level, booleans may be encoded as integers.
- All variables are defined up front, in a single, global scope. Upon declaration, variables are initialised to `0`, respectively `false` for booleans. Variables should be stored in memory; for this purpose, you have to calculate the offset of every variable with respect to the base of the scope. Remember that an integer uses four bytes.
- Besides assignments and blocks, the language supports **`If`**- and **`While`**-statements. This means that you have to generate conditional and immediate jumps (`cbr` and `jumpI`, respectively. For this purpose, you will need the tree visitor functionaltity introduced in Exercises 5-CC.1 and 5-CC.2.

Two sample PASCAL programs are provided together with the lab files:

- `gcd.pascal`, containing Euclid's algorithm for the computation of the greatest common divisor of two numbers;
- `prime.pascal`, containing a naive primehood detection algorithm.

Exercise 5-CC.5 Write a Simple Pascal program `gauss.pascal` that calculates the sum of all integers up to some upper bound set through an **`In`**-statement. *This question does not have to be signed off separately; you'll be asked to add this program to the tests of one of the next questions.* □

Because this language is no longer trivial, it becomes necessary to use a *two-pass compiler*. The first pass should achieve the following:

- It takes care of type checking;
- It calculates the offsets of the declared variables;
- It determines the entry points of the flow graphs corresponding to the statement and expression nodes of your parse tree.

The first pass (Exercise 5-CC.6) can be implemented using a tree listener. The results of this pass are collected into a single object that is used in the second pass (Exercise 5-CC.7 below) to generate ILOC code.

Exercise 5-CC.6 Complete the tree listener `pp.block5.cc.simple.Checker` so that it returns an instance of the class `Result` containing the outcome of the checker phase. Note that all expression-related listener methods have already been implemented; you only have to add the statement- and declaration-based methods. (Look into the grammar `SimplePascal.g4` to see which methods those are: as usual, everything that (transitively) occurs in the rules for `decl` and `stat` can potentially play a role in the calculation of the attributes.)

Test your solution using the provided `SimpleCheckerTest`. □

The second pass is more conventiently programmed as a tree visitor rather than a tree listener.

Exercise 5-CC.7 Complete the tree visitor `pp.block5.cc.simple.Generator` so that it generates (correct) ILOC code. Test your solution by running `SimpleCompiler` manually on the provided PASCAL files (`basic`, `fib`, `gcs`, `prime`) as well as your own solution to Exercise 5-CC.5, and also by running `SimpleGeneratorTest`.

To use the provided `Generator` optimally, consider the following:

- The generic type of the `TreeVisitor` has been instantiated to `Op`, meaning that every `visit` method returns a value of this type. You can take advantage of this to return the *first instruction* of the code emitted for a given parse tree node, which in turn can help you to find the correct (conditional and immediate) jump targets.
- To generate code for the conditional statements (**`If`** and **`While`**), you can apply the techniques you learned for the (bottom-up or top-down) control flow graph construction in Exercise 4-CC.4 of the previous Block.

□

5-CC.2 Dealing with procedures

Exercise 5-CC.8 Solve Exercise 6.3 from EC. □

Exercise 5-CC.9 In the PASCAL program of the previous exercise, consider the situation where

- `Main` has called `P1` (Line 33);
- `P1` has called `P4` (line 16);
- `P4` has called `P5` (line 30);
- `P5` has called `P1` (line 25).

Draw a graph showing the activation records at this point (including the one of `Main`), along the lines of Fig. 6.4 and 6.8, for each activation record including:

- The local data area (containing slots for each of the local variables — you do not have to invent values of the variables);
- The pointer to the caller's ARP;
- The access link (see Fig. 6.8). □

Exercise 5-CC.10 Solve Exercise 6.6 from EC, taking the following points into account:

- Figure 6.7 from EC gives an example of the kind of answer that is expected here.
- Show a single instance of `Elephant` and a single instance of `Dumbo`; choose your own values for the instance variables.
- The situation in JAVA differs from the one depicted in Fig. 6.7 in that the class of a class object is `Class`, whereas the superclass of every class that does not explicitly declare its own superclass is `Object`. Make sure you depict this situation correctly. □

Exercise 5-CC.11 Solve Exercise 6.8 from EC for the cases of call-by-value, call-by-reference and call-by-name (in other words, omitting call-by-value-result). For each of the simulations, show the content of the variables `a` and `i` after lines 6, 7, 8 and 9 of the program. You do not have to show the result of the print statements. □

Exercise 5-CC.12 Solve Exercise 6.10 from EC. Take into account that the ARs are stack-allocated here, and hence you have to actually combine them into a single stack. □

Exercise 5-CC.13 Solve Exercise 6.11 from EC. □

5-LP Logic Programming

Note that this block there are no new LP tutorials, but there is the **LP project**.

Block 6

Block 6

6-OV Overview

6-OV.1 Contents of This Block

CP This block discusses safe concurrent programming models that try to avoid concurrency errors by providing sufficient guarantees. We discuss Rust, a new programming language, where many concurrency errors are excluded by typechecking, and we discuss Software Transactional Memory, where operations on shared memory are executed as transactions, which can be retried if they interfere with other concurrent transactions.

FP This block has supervised lab time to work on the FP project.

CC This block continues and finishes the discussion of the procedure abstraction and memory layout started in Block 5.

LP This block finalises the LP strand with the project.

6-OV.2 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 1 hour self-study to prepare the CP exercises;
- 8 hours of unsupervised work on the FP project;
- 2 hours self-study to complete the CC lab exercises.

6-OV.3 Materials for this Block

- **CP** For Rust, the material is in the programming language guide:
  - The Rust Programming Language, second edition (draft). Read in particular the chapters on Understanding Ownership and Fearless Concurrency.

  Links are available on Canvas. For Software Transactional Memory, the material will be covered by the slides and the exercises.
- **CC**, from EC: Chapter 6.

6-CP Concurrent Programming

6-CP.1 Rust: Safe Shared Memory Concurrency and Message Passing Concurrency

Rust imposes strict rules on ownership of variables. This ensures memory safety, but also thread safety (as the ownership rules of Rust ensure that a program must be free of data races). Therefore, we first make a few exercises to become acquainted with Rust's *move semantics*.

☞ **Exercise 6-CP.1** *Learning objective:* Become familiar with Rust's ownership mechanism.

On this webpage: `https://github.com/carols10cents/rustlings` there are several small exercises on Rust's move semantics. Make the exercises `move_semantics1.rs`, `move_semantics2.rs` and `move_semantics3.rs` □

Next, we will make some exercises dealing with the safe shared memory concurrency as supported by Rust.

☞ **Exercise 6-CP.2** *Learning objective:* Becoming acquainted with basic shared memory concurrency in Rust

Make exercise `threads1.rs` on `https://github.com/carols10cents/rustlings`. □

☞ **Exercise 6-CP.3** *Learning objective:* Understand how Rust ensures safe shared memory concurrency.

Consider the program `array_sum.rs` provided on Blackboard. This is an unfinished Rust program. The idea is that there are two threads that sum up the elements in an array. The first thread sums up the elements in the first half of the array. The second thread sums up the elements in the second half of the array.

1. The provided program does not compile. Make it compile, and make sure that the results of the two parallel computations are added and printed on screen.
2. Now we want to update the elements in array, and set them all to 0. The file `array_reset` provides a skeleton for this (it compiles, but is not complete). Try to finish this, such that the contents of the resulting array are printed after the two threads have finished. Test your program. Explain why the contents of the array not changed.
3. Change your program to use a vector instead of an array, and make your program compile and behave correctly (using 2 threads). Hint: to pass mutexes to different threads, one should clone them before moving them to a new thread.
4. Finally, change the `array_sum` program so that it uses a single shared variable `sum`.

□

6-CP.2 Software Transactional Memory

Software Transactional Memory (STM) algorithms provide programmers with a high-level synchronization technique for concurrent programming. STMs guarantee "seemingly atomic" access to shared state via transactions. Transactions access locations in shared state. The following operations need to be supplied by STM implementations:

- a start operation (sometimes also called begin operation),
- a write operation writing a particular value on a particular location,
- a read operation reading from a particular location,
- a cancel and a commit operation.

Transactions need to be well-formed, i.e., always start with the start operation, then to be followed by a number of reads and writes and finally end with a call to the commit operation. The implementation of the STM provides the necessary synchronization to get "seeming atomicity", typically by logging e.g. the accesses done within the transactions (like read and write sets). Basically, the STM has to make sure that every transaction operates on a consistent view of the shared state.

☞ **Exercise 6-CP.4** *Learning objective:* Understand how to implement and use software transactional memory.

FastLane is a special STM implementation mode, which is designed to operate in a concurrent setting with low contention. It assigns exactly one thread the role of being a master. All other threads become helper threads. The master thread has priority over the other threads and is therefore on the "fast lane". Typically, the FastLane mode is combined with other modes, and implementations dynamically switch between the modes.

At the start of a transaction, it is determined whether the current thread is a master or a helper. If the thread is a master, it simply goes ahead and does its read and write actions. Helper transactions are more complicated as they contain all the handling of conflicts with the master thread. Helper threads have to wait until the master has committed, before they can commit their actions, and if the master thread changes something that they wanted to read or write, they will have to abort and retry.

1. Implement a Fastlane STM version.
2. Write a small program that uses STMs for handling concurrency, and use it to test your FastLane implementation.

□

6-FP Functional Programming

The work of Block 6 consists of doing the FP project.

6-CC Compiler Construction

Consider the following PASCAL program:

```
Program fib;

Function fib(n: Integer): Integer;
    Begin
        If n <= 1
        Then fib := 1
        Else fib := fib(n-2) + fib(n-1)
    End;

Var arg, result: Integer;
Begin
    In("Argument? ", arg);
    result := fib(arg);
    Out("Result: ", result)
End.
```

The following PASCAL-specific points should be noted:

- Keywords are case-insensitive;
- Function names and variable names cannot overlap;
- The return value of a function is determined by assigning it to the function name, as if it were a variable.

As before, **`In`** and **`Out`** are special functions introduced for the purpose of this course to provide a limited form of user interaction; they correspond one-to-one to the ILOC instructions **`in`** and **`out`**.

Exercise 6-CC.1 You are asked to write an ILOC program that faithfully simulates the above PASCAL program, in particular including the recursive calls of `fib`.

1. What is the layout of your activation records, i.e., what do you need to store there, and at what relative position? Take EC Fig. 6.4 as a reference for the possible components of an activation record. *(This question is intended to guide you for the next steps; do not invest too much time in a very precise answer.)*
2. Where will you place your activation records: dynamically allocated on the heap, dynamically allocated on the stack, or statically allocated? See pages 283–284 of EC for reference.
3. Write the ILOC code and test it using the ILOC `Simulator`. You may freely take advantage of the ILOC extensions described in Appendix B (in this reader); in particular the usage of labels to refer to line numbers (and to store the return address for a procedure call) and the built-in stack functionality.
4. At around $n = 20$ and above, the simulation starts to get very slow. Explain this phenomenon. □

Challenge exercise. The following exercise does not need to be signed off; however, it is recommended for everyone who plans to include procedures/functions in the language they plan to design for the final project.

Among the provided files you will find `SimplePascal6.g4`, which is the same grammar as in the last block, and `FuncPascal6.g4`, which extends it with function declarations and calls. Note that functions can only be declared on the top level.

Issues you have to address when extending your solution to cope with functions are:

- You should type check your function calls. To enable this, the provided `Type` class also contains a `Function` type. Optionally, you may also want to test if every function always assigns a return value.
- The simple `Scope` class is no longer sufficient, as you have to distinguish local variables (of the functions) from global variables (of the main program). For this purpose, you can use symbol tables as seen before. You should consider carefully at which moment you open the scope of a function: the function parameters should be part of the inner scope.

Exercise 6-CC.2 *This exercise is optional and does not have to be signed off.* Program classes `pp.block6.cc.func.Checker` and `pp.block6.cc.func.Generator` analogous to the ones of last Block (Exercises 5-CC.6 and 5-CC.7) for the grammar `FuncPascal6.g4`. Test your result using the provided example programs (in the lab files). □

6-LP Logic Programming

The work of Block 6 consists of finishing and submitting the LP project.

Block 7

Block 7

7-OV Overview

7-OV.1 Contents of This Block

CP This block discusses lock-free algorithms and data structures, as well as the basics behind memory models.

LP No new material is offered in this block. There is a deadline for submitting the LP project.

FP No new material is offered in this block. There is an FP test.

CC No new material is offered in this block. There is a deadline for signing off the lab exercises of Block 6.

7-OV.2 Mandatory Presence

During the following activities, your presence is mandatory.

7-OV.3 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 4 hours self-study to prepare the CP exercises;
- 2 hours of unsupervised work on the FP project;
- 6 hours self-study to prepare for the FP test.

7-OV.4 Materials for this Block

CP, from JCIP: Chapters 14.5, 14.6, 15 and 16. You might also want to look at some online material, such as:

- A quick overview of lock-free programming problems
- Some more elaboration on the ABA problem.

7-CP Concurrent Programming

Lock-free programming. Besides solving many concurrency-related problems (e.g., data races), locks can also introduce a wide range of problems (e.g., deadlocks and performance degradations). However, there are ways to make containers thread safe without the use of locks, so-called *lock-free* programming (aka lockless program). A program is called lock-free if there always is some thread that is able to make progress in some way.

☞ **Exercise 7-CP.1** *Learning objective:* Understand the challenges of lock-free programming.

In this exercise, we will look at lock-free stack implementations.

1. One of the most simple data structures to make lock-free is a stack. During the lecture we discussed Treiber's stack. Implement a lock-free stack yourself, featuring both a lock-free `push` and `pop` operation. Make sure that your stack implements the following interface:

```
public interface LockFreeStackInterface<T> {
    void push(T x);

    T pop();

    int getLength();
}
```

2. As always: devise a test, and apply it to make sure that your lock-free implemenation works as expected.
3. Can you adjust your implementation to change it into a bounded stack? If yes, explain how you do this. If not, why is this not possible?

□

Barrier implementation. In the previous blocks, we have seen how barriers can be used for synchronization of threads. As with locks, there are many ways to implement a barrier, in both locked and lock-free ways.

Consider the interface `Barrier`:

```
public interface Barrier {
    public int await() throws InterruptedException;
}
```

☞ **Exercise 7-CP.2** *Learning objective:* Understand how synchronizers can be implemented.

In this exercise, we will look at various ways to implement barriers.

1. Implement a cyclic barrier with locks and a `wait/notify` or `await/signal` system. The use of `wait/notify` has been discussed during the lecture of block 2. Furthermore, some additional resources on writing code with `wait/notify` can be found here:
   - Why `wait`, `notify` and `notifyAll` must be called from synchronized block or method in Java.
2. Implement a cyclic barrier using lock-free programming. Similarly to the previous exercise, think very carefully about your design as lock-free programming can introduce very subtle bugs. However, by using some of the classes in `java.util.concurrent` you can simplify your design a lot – decreasing the odds of concurrency bugs.

□

Memory Models. To understand the unexpected behaviours of a program with data races when it is executed with a relaxed memory model, here are a few exercises.

☞ **Exercise 7-CP.3** Motivate your answers!

1. If initially `x` and `y` are `0`, then what are the possible final values of `i` and `j`?

Thread 1	Thread 2
`x = 1`	`y = 1`
`j = y`	`i = x`

2. If initially `answer` = `0`, what can be printed after executing the following threads?

Thread 1	Thread 2
`answer = 42`	`if (ready) then`
`ready = true`	`  println(answer)`

   What would change if `ready` is declared `volatile`?

3. When executing the following threads, can this result in `r1 = r2 = 1` (if initially `x` and `y` are 0)?

Thread 1	Thread 2
`r1 = x`	`r2 = y`
`if r1 > 0 then`	`if r2 > 0 then`
`  y = 1`	`  x = 1`

□

Appendix A

Functional Programming

A-1 GHC installation

On Windows Get the full Haskell Platform **version 8.0.2a**. Go to `https://www.haskell.org/platform/prior.html` And download and install the "Full (64 bit)" version (not the Core version).

On Linux Install the packages ghc and cabal-install using the package manager. Make sure ghc is at least version 7.10 and lower than 8.4 (preferred version 8.0.2a). In case you cannot use an older version, please note that the definition of `Monoid` may different from what is required for this module (see also A-6).

If this does not apply to your situation, follow the instructions at: `https://www.haskell.org/downloads/linux`
Or: `https://www.haskell.org/platform/linux.html`

On Apple Download the full platform installer **version 8.0.2a** from here: `https://www.haskell.org/platform/prior.html`

A-2 QuickCheck

For Functional Programming the QuickCheck tool is used for *random testing*. With random testing, the programmer does not specify inputs for the tests, but instead they are randomly generated.

Installation QuickCheck can be installed using cabal, the package management system in Haskell. Use the following shell commands to install QuickCheck:

```
cabal update
cabal install QuickCheck
```

You can verify the installation of QuickCheck within GHCi:

```
Prelude> :m Test.QuickCheck
Prelude Test.QuickCheck> let prop_show_read x = read (show x) == x
Prelude Test.QuickCheck> quickCheck prop_show_read
+++ OK, passed 100 tests.
```

This codes tests if a value is changed by converting to a String and back.

Propositions Properties are functions that check if, for some given code, a condition holds. For example, for the *reverse* function, the following property must always hold[1]:

```
reverse (reverse xs) == xs
```

To describe such a property in QuickCheck, we write a function that receives any number of inputs and produces *True* when the property holds for the given inputs, or *False* otherwise. Since the input for this property is *xs*, the QuickCheck property can be described as:

```
prop_twicereverse :: Eq a => [a] -> Bool
prop_twicereverse xs = reverse (reverse xs) == xs
```

For any given list this function must produce *True* . This property can be manually tested, for example by feeding it representative lists as input. For example:

```
Prelude> prop_twicereverse []
True
Prelude> prop_twicereverse [1,2,3]
True
Prelude> prop_twicereverse "abc"
True
```

Generating good test inputs is labor intensive, error prone and also not the goal of this course. Instead we use *QuickCheck*, which automatically generates a test set of random inputs and applies them to the proposition. Propositions may have any name, but as a convention we always use the prefix *prop_* to indicate a function is a proposition.

Testing propositions with QuickCheck To use *QuickCheck*, some modules need to be imported. Start your Haskell file (.hs) with the following lines (replace ... by a module name you want to use for this file):

```
{-# LANGUAGE TemplateHaskell #-}
module ... where

import Test.QuickCheck
import Test.QuickCheck.All
```

You may add propositions to this file (e.g., *prop_twicereverse*). Using the Haskell function *quickCheck* you can test your function:

```
> quickCheck prop_twicereverse
+++ OK, passed 100 tests.
```

[1] otherwise reversing a list twice would change the list!

Sometimes it is desirable to specify a *type* for the proposition, since otherwise QuickCheck may use types that do not generate a counter example for incorrect code. For example, try:

```
brokenSort :: Ord a => [a] -> [a]
brokenSort xs = reverse (sort xs)

prop_sort xs = brokenSort xs == sort xs
```

While it is straightforward to generate a counter example, QuickCheck does not always find one. Try `verboseCheck prop_sort` to see why QuickCheck fails to find a counter example. When you restrict the types, QuickCheck is able to find a counter example:

```
brokenSort :: Ord a => [a] -> [a]
brokenSort xs = sort xs

prop_sort :: [Int] -> Bool
prop_sort xs = brokenSort xs == sort xs
```

Filtering inputs Consider the following code:

```
{-# LANGUAGE TemplateHaskell #-}
module Test where

import Test.QuickCheck
import Test.QuickCheck.All

myhead (x:xs) = x

prop_myhead xs = myhead xs == head xs
```

This proposition suggests that *myhead* should have the same behavior as *head*. However, *QuickCheck* does not accept the proposition and generates a counter example (the empty list):

```
Prelude> :l Test.hs
[1 of 1] Compiling Test             ( Test.hs, interpreted )
Ok, modules loaded: Test.
*Test> quickCheck prop_myhead
*** Failed! (after 1 test):
Exception:
  Test.hs:7:1-17: Non-exhaustive patterns in function myhead
[]
```

While *myhead* is correct, the test should not generate the empty list as input. QuickCheck offers an "implication operator" to filter out all inputs that do not satisfy the implication. For example

```
prop_myhead :: Eq a => [a] -> Property
prop_myhead xs = xs /= []  ==>   myhead xs == head xs
```

means that for each list *xs*, QuickCheck only uses the lists that are not empty. Note, that when the implication operator is used, the type *Bool* is replaced by *Property*.

An message such as

```
*** Gave up! Passed only 78 tests.
```

is not always an issue. This indicates that QuickCheck did not find 100 inputs that are accepted by the implication operator quickly enough.

Multiple tests Tests can be combined by *QuickCheck* in a single function, as illustrated by the following example:

```
{-# LANGUAGE TemplateHaskell #-}
module ... where

import Test.QuickCheck
import Test.QuickCheck.All
import Data.List

-- your tests go here, e.g.:
prop_reverse xs  =   xs == reverse (reverse xs)
prop_sort xs = ...

-- QuickCheck collects all previous prop_* tests
return []
check = $quickCheckAll
```

This generates a function *check* that tests *all* propositions that have the prefix $prop\_$.

Higher order functions QuickCheck can sometimes be used to generate *random functions*. For example:

```
{-# LANGUAGE TemplateHaskell #-}
module Test where

import Test.QuickCheck
import Test.QuickCheck.Function
import Test.QuickCheck.All

prop_map :: Eq b => Fun a b -> [a] -> [a] -> Bool
prop_map (Fn f) xs ys = map f (xs ++ ys) == map f xs ++ map f ys
```

Here (Fnf) is here used instead of f to make it possible for QuickCheck to generate functions.

Generators for user-defined data types Many functions you write receive values of types that you defined. You can define QuickCheck generators for these types, as explained in this section. The material in this section heavily relies on applicative functors, which are explained in block 3.

To generate a value of some type `a`, QuickCheck defines the typeclass `Arbitrary`:

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a -> [a] -- not mandatory and not discussed here
```

The function `arbitrary` defines a generator, which can be used by `generate :: Gen a -> IO a` to generate an arbitrary value of type `a` in an IO context[2]. In GHCi you can use it generate values as follows:

```
Prelude Test.QuickCheck> generate (arbitrary :: Gen Int)
17
Prelude Test.QuickCheck> generate (arbitrary :: Gen String)
"\DC3,\627920\SOo\262049\392398\CAN\250651\SYNr\b\t"
Prelude Test.QuickCheck> generate (arbitrary :: Gen String)
"}YX\RS\195435h"
Prelude Test.QuickCheck> generate (arbitrary :: Gen (Char, Int))
('f',10)
```

The `Gen` data type has **`Functor`** and `Applicative` instances. for example, we can use:

```
Prelude Test.QuickCheck> data Test a = Test a deriving Show
Prelude Test.QuickCheck> generate (Test <$> (arbitrary :: Gen Int))
Test 4
Prelude Test.QuickCheck> data Tuple a b = Tuple a b deriving Show
Prelude Test.QuickCheck> generate (Tuple <$> (arbitrary :: Gen Int
                                        <*> (arbitrary :: Gen Int))
Tuple (-1) (-3)
```

This can be used to define `Arbitrary` instances, e.g.:

```
data Tuple a b = Tuple a b deriving Show

instance (Arbitrary a, Arbitrary b) => Arbitrary (Tuple a b) where
  arbitrary = Tuple <$> arbitrary <*> arbitrary
```

QuickCheck offers many functions that you may use to control the generation of data, for example:

- `anyOf :: [Gen a] -> Gen a`
  QuickCheck randomly chooses one generator from a list. For example:

  ```
  data Direction = East | West | North | South deriving Show
  instance Arbitrary Direction where
    arbitrary = oneof [ pure East, pure West, pure North, pure South ]
  ```

- `frequency :: [(Int, Gen a)] -> Gen a`
  QuickCheck randomly chooses one generator from a list using the given frequencies. For example:

  ```
  data MyList a = Nil | Cons a (MyList a) deriving Show
  instance Arbitrary a => Arbitrary (MyList a) where
    arbitrary = frequency
      [ -- probability of 1/3
        (1, pure Nil),
        -- probability of 2/3
        (2, Cons <$> arbitrary <*> arbitrary) ]
  ```

[2]More precisely: it defines an IO action to generate an arbitrary value.

When defining generators for recursive data types, special care is needed. For example, consider the following generator for trees:

```
data BinTree a = Leaf | Node (BinTree a) (BinTree a) a   deriving Show
instance Arbitrary a => Arbitrary (BinTree a) where
  arbitrary = frequency
   [ (1, pure Leaf),
     (2, Node <$> arbitrary <*> arbitrary <*> arbitrary) ]
```

A node is generated with probability of 2/3, which may result in a high probability that arbitrary never terminates. We can use `sized :: (Int -> Gen a) -> Gen a` to address this issue. This function is used to construct generators that depend on a (used-specified) size, e.g., a tree with a certain depth.

```
instance Arbitrary a => Arbitrary (BinTree a) where
  arbitrary = sized arbtree

arbtree :: (Arbitrary a) => Int -> Gen (BinTree a)
arbtree 0 = pure Leaf
arbtree n = frequency
   [ (1, pure Leaf),
     (2, Node <$> (arbtree (n div 2))
              <*> (arbtree (n div 2))
              <*> arbitrary) ]
```

The function `arbtree` generates an arbitrary tree, while making sure the tree has at most the depth `n`. Using `resize`, this size can be specified. For example:

```
Prelude Test.QuickCheck> generate (arbitrary :: Gen [Int])
[13,-20,-21,-27,12,11,-1,-9,15,-24,6,28,-17,-19,-13,-3,-29]
Prelude Test.QuickCheck> generate (resize 2 $ arbitrary :: Gen [Int])
[-2]
```

A-3 IO in Haskell

This section is intended for students who want more background on IO in Haskell.

A-3.1 IO Actions

All functions in Haskell are pure, hence in principle doing IO *from* Haskell functions is impossible. Fortunately, Haskell provides an elegant way to deal with IO in Haskell without violating purety, namely IO actions. The IO actions in Haskell describe what happens when they are later on executed, and pure functions can be used to combine IO actions.

For any given string, the *pure* function **`putStrLn :: String -> IO ()`** generates an IO action that displays this string on the screen (i.e., terminal window). Hence, **`putStrLn`** `"Hello, world!"` is an IO action that does not do anything besides describing that—at the time when it is executed—the string "Hello, world!" should be displayed. To execute the IO actions the GHCi prompt can be used, as is demonstrated by the following example:

```
Prelude> a = putStrLn "Hello, world!"
Prelude> :t a
a :: IO ()
Prelude> a
Hello, world!
```

```
Prelude> putStrLn "Hello, world!"
Hello, world!
```

In this example, **putStrLn** "Hello, world!" is evaluated, which results in the IO action bound to a. The second line executes this IO action.

To summarize, IO in Haskell consists of two steps:

1. Haskell expressions/functions are *evaluated* to define IO actions.
2. IO actions are *executed*.

Besides executing an IO action in GHCi, you may also define a module named Main with an IO action called main. For example:

```
module Main where

main = putStrLn "Hello, world!"
```

This module can be compiled with the GHC compiler to produce a stand-alone executable.

A-4 IO Functor

IO actions can result in values. For example the IO action **getLine** :: **IO String** reads a line of text from the keyboard, and results in a value of type **String** in the IO context of type **IO String**. By default, GHCi displays the value in the IO context after execution. Clearly Haskell functions cannot extract this **String** from the **IO** context, since it does not exist yet: it just describes actions that are not executed yet.

Instead, to operate on values in the IO context, the IO Functor is used to apply pure functions to the result of an IO action. The following example illustrates this:

```
Prelude> :t getLine
getLine :: IO String
Prelude> getLine
test
"test"
Prelude> :t fmap length getLine
fmap length getLine :: IO Int
Prelude> fmap length getLine
test
4
```

Similarly, the Applicative Functor for IO combines multiple IO actions to a single IO action. For example:

```
Prelude> :t (,) <$> getLine <*> getLine
(,) <$> getLine <*> getLine :: IO (String, String)
Prelude> (,) <$> getLine <*> getLine
abc
def
("abc","def")
```

A-5 Sequencing IO

The following is not part of the course and should not be used in the lab, assignment and exam!

Consider the following two definitions:

```
putStrLn :: String -> IO ()

getRev :: IO String
getRev = reverse <$> getLine
```

Below we study how to use the result of executing the IO action `getRev` as input to the IO action **`putStrLn`**. To this end we need a function that sequences these specific IO actions such that the value produced by the executing the IO action `getRev` is used as input to the IO action **`putStrLn`**. Such a function receives both definitions and produces the composition, hence it would be of type **`IO String`** `->` `(`**`String`** `->` **`IO`** `())` `->` **`IO`** `()`. Haskell generalizes this idea, and defines an operator `(>>=)` `::` **`IO`** `a -> (a ->` **`IO`** `b) ->` **`IO`** `b` to sequence IO actions[3]. Usage of this operator is illustrated by the following example:

```
Prelude> rev = getRev >>= putStrLn
Prelude> :t rev
rev :: IO ()
Prelude> rev
abc
cba
```

To sequence two **`putStrLn`** actions, we may use lambda abstraction to provide a (dummy) argument for the second **`putStrLn`** as follows:

```
Prelude> putStrLn "Hello" >>= (\_ -> putStrLn "world!")
Hello
world!
```

Note, the type of `\_ ->` **`putStrLn`** `"world!"` is `a ->` **`IO`** `()`. Similarly, lambda bindings may be re-used in sequencing IO actions, for example:

```
Prelude> getRev >>= (\x -> putStrLn x >>= (\_ -> putStrLn x))
abc
cba
cba
```

Such nested lambda bindings may at first look intimidating, and it is best to study the types of the individual nested expressions in GHCi. Fortunately, to avoid such hard to read deeply nested lambda expressions to sequence IO actions, Haskell offers the do-notation syntax to generate them. To use this do-notation to define `revtwice`, we get:

[3] Actually, the definition in Haskell is slightly more general.

```
revtwice :: IO ()
revtwice = do x <- getRev
              putStrLn x
              putStrLn x
```

It is important to note that the Haskell compiler translates this directly to lambda expressions composed with the >>= operator.

A-6 Monoids in GHC 8.4

In this module we use GHC 8.0.2a. In newer Haskell version, since GHC 8.4, the definition of Monoid has slightly changed. This is illustrated using the following Monoid:

```
data MySum a = MySum { getMySum :: a }
             deriving Show

instance Num n => Monoid (MySum n) where
  mempty = MySum 0
  mappend (MySum x) (MySum y) = MySum (x + y)
```

For GHC 8.4 and newer, the `Semigroup` typeclass has been introduced, where the binary operator is defined. The `Monoid` typeclass now just requires a definition for `mempty`. Note, that the `Monoid` typeclass requires instances to be also an instance of `Semigroup`.

The example given above can be rewritten as follows:

```
instance Num n => Semigroup (MySum n) where
  (MySum x) <> (MySum y) = MySum (x + y)

instance Num n => Monoid (MySum n) where
  mempty = MySum 0
```

A-7 Some standard operators and functions

Below some important operations and functions predefined in HASKELL. See the on-line help function for more, and for a more extensive description. Definitions can be found under `installation|initialisation` in HASKELL-help.

```
negate, abs,
signum          :: Num a => a -> a
+, -, *         :: Num a => a -> a -> a
div, mod        :: Integral a => a -> a -> a
/               :: Fractional a => a -> a -> a
^               :: (Num a, Integral b) => a -> b -> a
abs, exp, log,
sqrt, sin, cos  :: Floating a => a -> a
                   various arithmetical operations and functions

min, max        :: Ord a => a -> a -> a
                   gives the minimum, maximum of two arguments

not             :: Bool -> Bool
&&, ||          :: Bool -> Bool -> Bool
                   boolean operations negation, conjunction, disjunction

isLower,
isUpper         :: Char -> Bool
```

says whether a letter is lower-case or upper-case

isAlpha :: **Char** -> **Bool**
says whether a character is a letter

isDigit :: **Char** -> **Bool**
says whether a character is a digit

isAlphaNum :: **Char** -> **Bool**
says whether a character is a letter or a digit

ord :: **Char** -> **Int**
converts a character to its Unicode number

chr :: **Int** -> **Char**
converts a Unicode number to the corresponding character

toLower,
toUpper :: **Char** -> **Char**
converts a letter to lower-case, upper-case

==, /= :: **Eq** a => a -> a -> **Bool**
>, >=,
<, <= :: **Ord** a => a -> a -> **Bool**
various comparison operations

even, **odd** :: **Integral** a => a -> **Bool**
says whether a (integral) number is even or odd

: :: a -> [a] -> [a]
adds element to the front end of a list (*cons*)

length :: [a] -> **Int**
length of a list

!! :: [a] -> **Int** -> a
list indexing

++, \\ :: [a] -> [a] -> [a]
list concatenation, list subtraction

head, **last** :: [a] -> a
tail, **init**,
reverse :: [a] -> [a]
elem :: **Eq** a => a -> [a] -> **Bool**
tests whether a list contains a given element

concat :: [[a]] -> [a]
concats a list of lists into one list

sort :: **Ord** a => [a] -> [a]
sum :: **Num** a => [a] -> a
minimum,
maximum :: **Ord** a => [a] -> a
take, **drop** :: **Int** -> [a] -> [a]
takeWhile,
dropWhile :: (a->**Bool**) -> [a] -> [a]
various functions on lists

insert :: **Ord** a => a -> [a] -> [a]

inserts an element into an ordered list

and, or :: [**Bool**] -> **Bool**
yields the conjunction, disjunction of a list of booleans

lines :: **String** -> [**String**]
breaks a string at newlines ('\n') into a list of strings

unlines :: [**String**] -> **String**
glues a list of strings with '\n'

fst :: (a,b) -> a
yields the first element of a pair

snd :: (a,b) -> b
yields the second element of a pair

zip :: [a] -> [b] -> [(a,b)]
turns two lists into a list of pairs

zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zips two lists and applies a function to the corresponding elements

map :: (a->b) -> [a] -> [b]
applies a function to all elements in a list

filter :: (a->**Bool**) -> [a] -> [a]
selects those elements from a list which satisfy a property

foldl :: (a->b->a) -> a -> [b] -> a
"folds" a list with a function, starting with a given value. Works from left to right through the list

foldr :: (a->b->b) -> b -> [a] -> b
like foldl, but works from right to left

foldl1, foldr1 :: (a->a->a) -> [a] -> a
like foldl, foldr, with first, last element of the list as starting value. Error for empty list

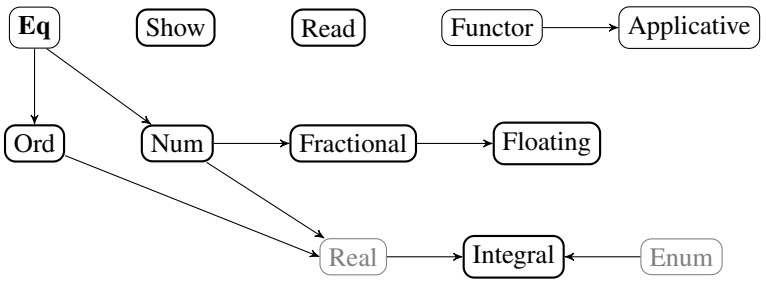
. :: (b->c) -> (a->b) -> (a->c)
function composition

seq :: a -> b -> b
partially evaluates first argument, and delivers the second

error :: **String** -> a
causes error with given string as error message

A-8 Some important type classes

This section gives an overview of the most important type classes used in this module.




```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}

class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
  {-# MINIMAL compare | (<=) #-}

class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS
  {-# MINIMAL showsPrec | show #-}

class Read a where
  readsPrec :: Int -> ReadS a
  readList :: ReadS [a]
  readPrec :: ReadPrec a
  readListPrec :: ReadPrec [a]
  {-# MINIMAL readsPrec | readPrec #-}

class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}

class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  {-# MINIMAL fromRational, (recip | (/)) #-}

class Fractional a => Floating a where
  pi :: a
  exp :: a -> a
  log :: a -> a
  ...

class Functor (f :: * -> *) where
```

```
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  {-# MINIMAL fmap #-}

class Functor f => Applicative (f :: * -> *) where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  (*>) :: f a -> f b -> f b
  (<*) :: f a -> f b -> f a
  {-# MINIMAL pure, (<*>) #-}

class Applicative f => Alternative (f :: * -> *) where
  empty :: f a
  (<|>) :: f a -> f a -> f a
  some :: f a -> f [a]
  many :: f a -> f [a]
  {-# MINIMAL empty, (<|>) #-}
```

A-9 Literature

There are a few good websites where lots of information on functional programming can be found. For example:

- `http://www.haskell.org`
- `http://www.cs.kun.nl/~clean`

From these sites both tutorials and implementations can be downloaded. Some books:

BIRD, R., P. WADLER, *Introduction to Functional Programming*, Prentice Hall, London, 1988

BIRD, R., *Introduction to Functional Programming Using Haskell* (second edition), Prentice Hall, London, 1998

DAVIE, A.J.T., *An Introduction to Functional Programming Systems Using Haskell*, Cambridge UP, Cambridge, 1992

HUDAK, P., *The Haskell School of Expression*, Cambridge UP, Cambridge, 2000

HUTTON, G., *Programming in Haskell*, Cambridge UP, Cambridge, 2016

LIPOVACA, M., *Learn You a Haskell for Great Good!*, No Starch Press, 2011, Online: `http://learnyouahaskell.com/`

RABHI, F., G. LAPALME, *Algorithms, a Functional Programming Approach*, Addison-Wesley, Harlow, 1999

O'SULLIVAN, B., *Real World Haskell*, O'Reilly Media, 2008, Online: `http://book.realworldhaskell.org/read/`

THOMPSON, S., *Haskell, the Craft of Functional Programming*, Addison-Wesley, Harlow, 2011

Appendix **B**

ILOC support

For the purpose of this course, we rely on ILOC as a linear Intermediate Representation, following the line set out in EC. We have taken some steps to make this work smoothly; those are described in this appendix.

- We provide tool support in the form of an assembler and simulator for ILOC code.
- We have added several extensions to the ILOC syntax to help in debugging, stack manipulation and procedure calls.

The extensions and tool support are summarised below. The main reference for ILOC is Appendix A of EC.

B-1 Memory layout and stack

For the purpose of this course, ILOC code memory and data memory are assumed to be strictly separated, and to have their own address space (in other words, the code memory at a given address *i* holds a different value than the data memory at address *i*). Code is assumed to be stored one line per memory location; data memory holds one byte per location.

Data memory is arranged according to Fig. 7.2 of EC: in particular, we have built in support for a *stack* that starts at the top of memory.

Data type representation in memory. ILOC is weakly typed. Essentially, it recognises only two data types

- Integers, stored in by four consecutive bytes of memory, most significant byte first
- Characters, stored in a single byte of memory. (Thus, only the old ASCII coding is supported.)

The weakness of the typing system is apparent from the fact that there is no check in ILOC whether a memory location that is at some point treated as an integer is in the next instance treated as a character. The same holds for registers: it is possible to store an integer in a register, for instance using **`load`**, and subsequently treat it as a character, for instance by applying **`c2i`** to transfer the value to another register; or vice versa. If an integer value is used as a character, only the least significant byte is used (similar to the coercion of an `int` to a `char` in JAVA, except that the ILOC characters are 1 byte only); vice vera, if a character value is used as an integer, it occupies only the least significant byte, so only values in the range 0–255 can ensue.

However, for legacy reasons, the ILOC simulator used in the course can be set to a 4-byte representation of characters in memory, using the method `Machine#setCharSize`. This method can be invoked on the VM

of a simulator (see also §B-5 below): the following lines cause the simulator to run on a VM with a 4-byte character representation.

```
Simulator sim = new Simulator("program.iloc");
sim.getVM().setCharSize(4);
```

Special registers: Program Counter and Stack Pointer. In addition to `r_arp` referred to in EC, we provide the following standard registers:

- `pc` : the *program counter*, which is a special register that holds the address of the instruction to be executed next. `pc` is initialised to 0.
- `sp` : the *stack pointer*, which is a special register used by the **`push`** and **`pop`** operations explained below. `sp` is initialised to the highest memory address plus one.

B-2 Additional naming conventions: Label references and strings

Sections A.1 and A.2 present the syntax of ILOC. To this we have added:

- Label references as `num` operands, represented symbolically by prefixing the label name with `#`.

  Labels can already occur as operands for control-flow operations, in which case they stand for the address of the first instruction at that label. Label references of the form `#label` can be used to store the correct return address for a method call. For instance,

```
        loadI #return => r_1
        jumpI -> method
return: // continuation after call
...
method: // method body, not touching the value in r_1
        jump  -> r_1
```

  will result in an (immediate) jump to the label `method`, followed by a jump back to the address stored in register `r_1`, which has been preloaded with the address of the first instruction at label `return`. (Of course, in a more realistic example the return address would be stored in the activation record of `method`.)

- String operands. These are only used for the special instructions **`in`** and **`out`** discussed below; they are represented in the usual way. Thus, effectively we have extended the grammar rule for *Operand* in Section A.1 of EC in the following way:

$$
\begin{array}{lcl}
\textit{Operand} & \rightarrow & \texttt{register} \\
& | & \texttt{num} \\
& | & \texttt{label} \\
& | & \texttt{string}
\end{array}
$$

  where the token type `string` is only used for those special instructions.

B-3 Additional instructions

We have extended the ILOC instruction set listed in Appendix A with the following additional instructions:

- **`out`** `s_1, r_2` : Prints a line consisting of string `s_1` followed by the content of `r_1` to the standard output.
- **`in`** `s_1` **`=>`** `r_2` : Prints string `s_1` to the standard output, then awaits user input and loads it into `r_1`. Only input that converts to a number is accepted.

- **cout** s_1 : Prints a line consisting of string s_1 followed by the string on top of the stack to the standard output. The string on top of the stack is assumed to consist of a 4-byte length value, followed by character values (first character on top). The size of a single character is 1 byte by default, but can be set in the VM; see §B-1. For instance, in the default setting the value "text" would be represented by the top 8 bytes of the stack: the first four encode the length value 4 (the length of the string "text"), whereas the next four represent the successive characters 't', 'e', 'x', 't' (in ASCII encoding). When the character size is set to 4 bytes, the same value would be represented by the top 20 bytes.
- **cin** s_1 : Prints string s_1 to the standard output, then awaits user input in the form of a string and pushes it onto the stack. For the encoding see the description of **cout** above.
- **push** r_1 : Decreases the value of the stack pointer (register sp initialised at the top of memory, see above) by 4 and stores the value of r_1 at sp. In other words, the following are equivalent:

```
push r_1      ⇔    subI sp,4 => sp
                   store r_1 => sp
```

- **pop =>** r_1 : Loads the value at the atack pointer (register sp, see above) into register r_1 and increases the stack pointer by 4. In other words, the following are equivalent:

```
pop => r_1    ⇔    load sp => r_1
                   addI sp,4 => sp
```

- **cpush** r_1 : Decreases the value of the stack pointer by the size of a character (by default 1, but this can be set on the VM, see §B-1) and stores the least significant byte of r_1 at sp. (The difference with **push** r_1 is thus that only a single byte is pushed, although depending on the character size this may still occupy 4 bytes of memory.)
- **cpop =>** r_1 : Loads the byte value at the atack pointer into register r_1 and increases the stack pointer by the size of a character (by default 1, but this can be set on the VM, see §B-1). (The difference with **pop =>** r_1 is thus that, in the standard setting only a single byte is popped.)

B-4 Initialisation of constants

To take maximal advantage of the symbolic num constants of ILOC (written @name), we allow an ILOC program to start with a sequence of assignments of the form

```
name1 <- number1
name2 <- number2
name3 <- number3
```

where the name? are symbolic num constants (without @-prefix) and the number? are (possibly signed) integers.

Effectively, we have extended the ILOC grammar as

$$
\begin{array}{rcl}
\textit{IlocProgram} & \rightarrow & \textit{AssignmentList}\ \ \textit{InstructionList} \\
\textit{AssignmentList} & \rightarrow & \textit{Assignment}\ \ \textit{AssignmentList} \\
 & | & \varepsilon \\
\textit{Assignment} & \rightarrow & \texttt{name <- int}
\end{array}
$$

B-5 Tooling

The tool support for ILOC (package pp.iloc) includes the following:

- Assembler: Compiles a textual version of ILOC into a Program object.

- `Simulator`: Simulates a `Program` on a given `Machine`.
- `Machine`: Representation of a virtual machine, with a `Memory` of user-controlled size, an unbounded set of registers (with predefined elements `r_arp`, `pc` and `sp`) and a mapping from symbolic constants to values.

Before simulation is kicked off, the `Machine` can be initialised by assigning values to memory, registers and symbolic constants, and also the size of character representations in memory can be changed from the default 1 byte to 4 bytes (see §B-1 above). (Assigning values to symbolic constants can alternatively be done in the assignment list in front of the ILOC instructions: see §B-4.

Appendix C

PP Final Project

The project of Programming Paradigms module consists of the development of a complete compiler for a (source) language that you may define yourself, as long as certain minimal criteria are met. The target machine for compilation is a realistic, though non-existent processor called SPROCKELL, for which a hardware-level simulator is provided in HASKELL. The corresponding machine language is called SPRIL (see the Tooling menu on BLACKBOARD).

Read this appendix carefully to understand what is expected.

- §C-1 describes the source language requirements
- §C-2 describes the possible options for the compilation process
- §C-3 describes what you should do to test your compiler
- §C-4 describes the requirements for the end result (software and report)
- §C-5 describes how the project will be assessed

Global schedule for the project.

- *Block 7:* Choice of language features, sample programs
- *Block 8:* Syntax definition, elaboration (type checking and other analyses),
- *Block 9:* Code generation and testing
- *Block 10:* Report and submission
- *Final deadline:* Friday 5 July 2019. The rules for late submission apply.

During Blocks 7–10, there are regular scheduled hours of assistance. You will be asked to show your progress and discuss your choices at least once during this period.

C-1 Source language

We go through a list of possible language features, for each feature mentioning whether supporting it in your language is optional or mandatory. If you fail to implement a mandatory part, you cannot get a sufficient grade for the project. §C-5 gives a summary and explains how the choice of features influences the expected grade for the project. Some of the optional features have not been treated during the lectures; however, EC Chapter 7 contains extensive pointers on how to implement them.

C-1.1 Data types

One of the first choices to make is which data types your language should support. The list below is a very short summary; section 4.2 of EC has given you an overview, and Chapter 7 of EC is devoted to implementation details.

- *Basic types: integers and booleans (mandatory)* In Blocks 3–5 of the CC strand you have already gained experience with these two most essential of basic types. See, however, the paragraph "Division" below on how to deal with integer division (which is *not* mandatory).

- *Arrays (encouraged).* Arrays make it much more easy to write non-trivial programs in your language. They provide an interesting challenge for the compiler, since it is no longer a priori clear what the size of a value is: obviously, this depends on the number of elements (the array length) as well as the size of the elements themselves. The array length may be fixed as part of the type (as was the case in early versions of PASCAL, for instance), but the may also be determined at run time, when the array is actually created (as is the case in JAVA). Run-time *changes* in array length are hard to support and may be ignored here. Note that, if the length is not fixed at compile time, you have to put array values on the heap, which raises the issue of heap management (see below).

- *Strings (optional).* Though very useful for pragmatic reasons, strings are a second-rate concept in most programming languages. Indeed, they can be seen as arrays of characters, but with their own typical constants and operators. Clearly, at the very least, the string type itself should not dictate the length of its values, making strings comparable to the trickier version of arrays mentioned above.

- *Enumerated types (optional).* These are user-defined scalar types (where the word *scalar* means that the values are singular and not composed). For the compiler, the challenge in supporting enumerated types lies in the choice of suitable syntax and the enforcement of a typing discipline; for code generation, values of enumerated types can be treated just like integer numbers.

- *Records and variants (optional).* A record is a combination of fields, each of which may have its own type. A variant, on the other hand, is a *choice* between different structures. A prime example of the latter are algebraic datatypes in HASKELL.

- *Pointers (optional).* A pointer is the address of a data value, rather than the value itself. The introduction of pointers to a language makes it unavoidable to place values on the heap, as their lifetimes are typically not limited to the lexical scope in which they are created. This again brings up the notion of heap management; see below.

- *Objects (optional).* Objects as in JAVA and other object-oriented languages are essentially pointers to records; for the compiler, they combine the challenges of both. Another layer of complexity is added if one also supports subtyping; in a way, this adds the notion of variants on top of the pointer/record combination.

Heap management. As soon as you have data values that are of unknown size (at compile time), you cannot put them into the allocation record any more; instead, you have to use the heap. This means that you need a mechanism to decide which locations on the heap are actually free at the moment the data value is created. This mechanism, called *heap management*, is a topic we have not covered at all during the course; see, however, §6.6.1 of EC for an extensive discussion.

More interesting yet is the notion of *automatic garbage collection*, as implemented in JAVA. §6.6.2 of EC covers this topic. Traditionally, the programmer had to explicitly deallocate heap memory when it is no longer used for anything; this is a famous source of very tricky errors, as a programmer may either forget to do this, or do it before the value has become unreachable. One of the big advantages of JAVA-like languages is that this task is taken out of the hands of the programmer.

Although you can't avoid having to allocate memory if you decide to implement dynamically-sized data values, for the purpose of this course you may ignore deallocation altogether. This means that your memory will slowly fill up if you repeatedly create such values, even if you abandon them afterwards. This phenomenon is called *memory leak*.

(On the other hand, would it not be extremely cool to write your own garbage collector?)

C-1.2 Simple expressions and variables (mandatory)

Your language should support appropriate expressions for all the data types you have chosen to implement. For the mandatory basic types, this comes down to the set of operators encountered in, e.g., the simple PASCAL fragment you have worked with in Blocks 4 and 5 of the CC strand, *except for division; see below*. For the type extensions, the kinds of operators strongly depend on the type itself. However, for every type you should at least offer:

- Denotations for primitive values of the type. For instance, for array types there should be a way to construct an array (something like `[1,2,3]` for an array of integers), and likewise for strings, records etc.
- Opreators for equality and inequality of values of the type. For instance, it should be possible to compare different values of the same array or record type. Such a comparison should be *content-based* rather than *identity-based*; in other words, it is *not* enough to compare the memory addresses where two (say) array values reside, they must be compared element by element.

Your language should support typed variables. Variable types may be given part of their declaration or inferred by their usage, but the type of every variable should be fixed throughout its scope and determined at compile-time. In other words, your language should be strongly typed.

Your language should support local, nested scopes. It is not sufficient to have a single, global scope level only: within your language, it should be possible to reuse the variable name declared elsewhere with the same or a different type, at which point the two instances have nothing to do with one another. This includes the reuse of a variable name in an *inner* scope; that is, you may not impose a restriction such as the one in JAVA that forbids the reuse of names in nested scopes.

Variables should also be checked for initialisation; that is, at compile time you should ensure that every use of a variable occurs after the variable has received an initial value. The initial value may be assigned explicitly (as for local variables in JAVA) or may be a default value of the type (as for instance variables in JAVA), but it should be well-defined.

C-1.3 Basic statements: Assignments, if and while (mandatory)

These language features are well-known and do not really need explanation. You are encouraged to look around a bit at other languages than the ones you know, to get inspiration about possible syntax and variations. For instance, besides `while`-constructs many languages also offer **`repeat`**, and there are widely differing variants of `for`-statements.

You may choose to allow assignments as expressions as in JAVA. For instance, in JAVA the following is legal:

```
int a = 3 - b = (c = 1) + 2;
```

which simultaneously assigns `1` to `c`, `3` to `b` and `0` to `a`.

Note: If you choose to implement arrays, ensure that the assignment of an array value *copies* the array: it is wrong to just assign a pointer to it, as then the old and new arrays are shared.

C-1.4 Concurrency (mandatory)

For concurrency, you should add features to your language that make it possible to start and stop processes (a.k.a. threads), and to lock and unlock them. These should be compiled to the special SPRIL instructions provided for this purpose (see BLACKBOARD). Each process runs on a dedicated SPROCKELL; in other words, the architecture does not support dynamic thread creation.

Starting and stopping threads can either be offered (in your source language) on the basis of a `fork`/`join`-construct (see, e.g., `https://en.wikipedia.org/wiki/Fork-join_model`) or on the basis of a `parbegin`/`parend`-construct — essentially a block in which the statements should not be executed in sequence but in parallel. It is sufficient to offer concurrency only on the global level, and not within loops or procedures; however, it should be possible to nest threads — i.e., a new thread can be `fork`ed from an already `fork`ed thread or one `parbegin`/`parend` is be allowed within another.

Inter-process communication in SPRIL is exclusively done through `WriteInstr`- and `ReadInstr`/`Receive`-instructions and an atomic `TestAndSet`, which all access a global, shared memory. This means that you have to distinguish between variables that are shared between threads, and hence should be stored in global memory, and variables that are local to a thread, which can therefore be stored locally. This distinction can either be made on the basis of an analysis (during the elaboration phase) about which variables are accessed by more than one thread; or you may choose to build this distinction into the syntax of your language.

Because the global memory is the only medium of communication between SPROCKELL-instances, starting a thread must be encoded in that way as well. The demo program in `DemoMultipleSprockells.hs` shows how this can be done: essentially, a SPROCKELL is made to poll a predetermined global memory address until a non-zero value appears there; this value is taken to be the start address of the code that is to be executed.

To show the correct functioning of the concurrency feature of your language, you should provide at least the following test programs:

- An implementation of Peterson's algorithm for mutual exclusion (`https://en.wikipedia.org/wiki/Peterson%27s_algorithm`)
- An elementary banking system, consisting of several processes trying to transfer money simultaneously. Your implementation should ensure that concurrent transfers do not mess up the state of the bank account.

C-1.5 Division (optional)

Division is not supported in SPRIL (there is a commented-out division operator, which however you may not just uncomment) as its inclusion is very unrealistic for an assembly language; hence it is not a mandatory operator. You may choose to implement "soft division", i.e., generate a loop that implements a division algorithm by using other SPRIL instructions.

C-1.6 Procedures/functions (optional)

The principle of procedures and functions is well known in programming, and has been extensively discussed in the last blocks of the CC strand. There are several variants you may choose to support:

- Procedures only, functions only, or both. Functions have the slight additional complication that the return value should be stored in the activation record.
- Nested procedure definitions. These bring the added complication of access links or global displays, to access variables in enclosing scopes.
- Call-by-reference parameters. These necessitate a level of indirection: not the value itself is placed in an activation record, but the address in memory where the value is stored.

C-1.7 Exception handling (optional)

From JAVA you know the principle that any run-time error is visible within your program as an object of a subclass of `Exception` (strictly speaking, of `Throwable`). This is a very important feature of a language, because it allows you to deal with unpredictable errors in a way that does not completely clutter up your code.

Less powerful but still very useful variants of exception handling also exist. For instance, you may choose not to distinguish between different kinds of errors, or only use a single value from a predefined range to distinghuish them. Also, it is possible to restrict exception handling to a set of predefined exceptions and not allow the user to define and throw his own. In any case, however, an exception handling mechanism involves

- The detection of the error
- The notification of the error (in JAVA: `throw`)
- The handling of the error (in JAVA: **`catch`**).

If you choose to include exception handling in your language, you have to take care of the syntactic aspect (how should a programmer specify the handler, and how does he define and throw his own exceptions if that is supported) as well as the code generation part (detection and handling).

C-1.8 Optimisations (optional)

Though in the CC lecture we have not devoted a lot of attention to possible optimisations, the EC book has several chapters on this topic. You are encouraged to build one or more optimisation phases into your compiler. Candidates are, for instance

- Local value numbering (§8.4.1 of EC)
- Loop unrolling (§8.5.2 of EC)
- Inline substitution (§8.7.1 of EC)

C-2 The compilation process

In the CC strand you have exclusively used ANTLR as parser generator, but you are aware that there are other choices; in fact, you have also learned how to program and generate a parser in HASKELL.

C-2.1 JAVA front-end and code generation

If you choose this option, you have to fully apply the ANTLR-based skills you learned during the CC strand of the module:

- Write lexer+parser rules in ANTLR;
- Write a tree listener, using symbol table and attribute-like rules, that allow you to perform the elaboration phase of the front-end;
- Write a tree listener or tree visitor for the code generation.

In the last step, instead of generating ILOC, you have to generate SPRIL and emit the code as a HASKELL file that essentially contains a list of SPRIL instructions.

C-2.2 JAVA front-end, HASKELL code generation

If you choose this option, you use ANTLR to generate a parse tree, which you then hand over as a HASKELL data structure and process further in HASKELL.

- Write a lexer+parser in ANTLR;
- Write a tree listener in JAVA, using symbol table and attribute-like rules, that allow you to perform the elaboration phase of the front-end;
- Write a tree listener to emit the parse tree as a HASKELL file containing a single data structure;
- Write a HASKELL function that converts this data structure into a sequence of SPRIL instructions.

In the third step, it makes sense to abstract away the parse tree's irrelevant details (such as brackets, commas and other spurious syntactic structure), turning it into an abstract syntax tree (AST). In HASKELL, such an AST can be captured very naturally as an instance of an algebraic data type.

C-2.3 HASKELL front-end and code generation

If you choose this option, you may entirely ignore ANTLR and write your compiler in HASKELL altogether.

- Write a lexer+parser in HASKELL (with ParSec) that generates an AST, encoded as an algebraic data type;
- Perform elaboration (scope and type checking) in HASKELL, on the basis of the generated AST;
- Write a HASKELL function that converts the AST into a sequence of SPRIL instructions.

C-2.4 SPROCKELL extensions

The SPRIL language and its simulator are all implemented in HASKELL. This makes it straightforward to extend the setup. In doing so, you also practice your FP skills and this can again be rewarded by some bonus points — see Table C.2. Whether or not a bonus will be awarded depends on the complexity of your extension; therefore, if you want to take advantage of this, please consult the teaching assistants.

Ideas for extension are:

- To add new instructions that ease the task of code generation.
- To change the channel mechanism that handles requests from parallel SPROCKELLs to the global memory, for instance by adding randomness.

C-3 Testing

You can increase your confidence in the correctness of a compiler by applying a series of tests. Of course, testing can only ever show the presence of errors, not their absence. Nevertheless, careful testing is useful and necessary in situations where formal verification is not possible. The advantage of testing is that it can be carried out independently of the way the compiler is specified and constructed.

Ideally, the tests should consist of all possible programs. Unfortunately, in most languages an infinite number of programs can be written, so all we can hope to do is to judiciously select a subset of all programs, called a test set, that is in some way representative of the language. A test set should not only contain correct programs, but also programs that contain errors, so that we can see how the compiler handles incorrect input. Errors in a program can occur in the:

- syntax (e.g. spelling errors in the lexical syntax, or language-construct errors)
- context constraints (e.g. declaration, scope and type errors)
- semantics (e.g. run-time errors)

Each of these classes of errors can be tested for separately.

C-3.1 Testing for syntax errors

This class of errors is about the first stages of compilation: scanning and parsing. For every language feature, you can consider the following (types of) test cases (the more comprehensive the better):

- One or more instances of that feature that should be syntactically correct. For such test cases, apart from checking that they pass the scanner and lexer, you can also check that the parse tree has the expected shape.
- One or more instances of that feature that should be syntactically incorrect. For such test cases, you can check that they are indeed rejected by the scanner or lexer.

For testing this class of errors, you do not have to invoke the full compiler, but just the scanning and lexing phases.

C-3.2 Testing for contextual errors

This class of errors is about scoping and typing: the things you check during the elaboration phase. What you should test strongly depends on the scoping and typing rules of your language. You may consider the following (types of) test cases (the more comprehensive the better):

- In a context where an identifier can appear (for instance: inside a block or procedure body), a test case where that identifier has been declared;
- In a context where an identifier can appear, a test case where that identifier has not been declared;
- For contexts that expect a certain type (for instance: an expression operand, **`if`**- or `while`-condition or procedure parameter), a test case where a simple expression of that (correct) type is used;
- For contexts that expect a certain type, a test case where a simple expression of a wrong type is used.

For testing this class of errors, you do not have to invoke the full compiler, but just the scanning, lexing and elaboration phases.

C-3.3 Testing for semantic errors

This class of errors is about run-time behaviour. Typically, you should include programs that are supposed to run correctly and of which the expected outcome is known, as well as programs that are known to contain a run-time error. You may consider the following (types of) test cases:

- Simple algorithms that calculate some value, for instance the number of days of the month February in any particular year (involving a test for leap years); whether or not a given number is prime. If you have implemented arrays, the scope for algrithms of the above kind becomes much larger.
- An algorithm that you expect to run into an infinite loop. Note that this is problematic to test automatically, as by definition your test will not terminate if the behaviour is as expected. In JUNIT, there is a way around this: the `@Test`-annotation has a parameter `timeout` that you can set to avoid tests that do not terminate; e.g.

  ```
  @Test(timeout = 1000)
  public void testSomething() {
      while (true) ;
  }
  ```

  will cause the method `testSomething` to terminate after 1000 milliseconds and flag an error.
- Algorithms that you expect to generate a run-time error, for instance division by zero.

In the test suite for this class of errors, ideally every feature of your language should occur at least once in a correctly running program (meaning that you actually test whether correct code is generated for that feature, at least in the particular context of your test). The test involves both running the compiler, including code generation, and running the generated code on the SPROCKELL virtual machine.

C-3.4 Automatic versus manual testing

All the tests above are essentially *system tests* (the system under test is your compiler). This is not the same as a *unit test*, where you focus on part of your system; in JAVA, typically a single class.

During the CC strand you have written and executed a lot of JUNIT tests, which as the name implies are mostly unit tests. The great thing about a test support library like JUNIT is that it runs a large test suite automatically, so it becomes very easy to test often and test thoroughly. JUNIT *can* also be used for system testing; however, in some cases setting up a system test is a lot of work, and a manual test is preferable. This may very well be the case for some of the tests described above, especially where you have to run a combination of JAVA, HASKELL and SPROCKELL.

> *For the PP project, if you have a reasonable collection of test programs of the categories described above, but they can only be run manually, this translates to the default 0 for the testing aspect (in a range from* $-1,5$ *to* $+1,5$*, see Table C.3). Automated tests are a bonus but not a requirement.*

C-4 The final product

The product you should submit for the final project consists the developed software and a printable report. These should be uploaded to BLACKBOARD in a single zip-file. The general conditions for late submission apply.

C-4.1 Software

The submitted zip-file should contain the following elements (in a well-structured directory hierarchy):

- *The report*, in PDF-format.

- *A README-file* with instructions on using the compiler. Such instructions may include, but are not limited to, an overview of the directories and files necessary for execution and the required steps for installation and invocation. Upon following these instructions, an end user should be able to obtain and invoke use a working compiler. *If this requirement is not met, the submission will not be graded; non-compiling programs are not accepted.*
- *Full grammars* as well as the code files generated therefrom by your chosen parser generator (ANTLR or otherwise).
- *Source code* of all classes programmed by you, in a single directory hierarchy, also including sources you reused from the CC laboratory. The code should meet the following criteria:
  - Compiles and executes without errors*
  - Implements all mandatory language features*
  - Contains documentation (in the form of JAVADOC or HASKELL comments)*
  - Meets common coding standards, for instance regarding naming, package structure, and visibility of fields.

  The criteria marked * are necessary to score more than 5,0 for the project.
- *Auxiliary code* of all predefined classes and libraries, insofar they are not part of the standard JAVA/HASKELL runtime environment.
- *Results of all tests.* See §C-3 for a discussion on what to test. For *correct* test programs, the result should consist of
  - The source code of the program itself
  - The generated SPRIL code
  - Some test runs

  For *incorrect* test programs, the result shoud consist of
  - The source code of the program itself
  - The output generated by the compiler for the program (i.e., the error messages)

Again, take care that your code compiles and runs after following the instructions in the enclosed `README`-file. *We should not have to change anything in your source code!* Typical cases where this goes wrong are: names and paths of files or other URLs, like host machines and servers. *Test this before submission.*

C-4.2 Report

The report should give insight in how the language has been defined, and how any problems occurring during the construction of the compiler were solved. The report should contain the following parts; for each part, list who of you was responsible, or whether the responsibility was shared equally.

- *Front page.* Clearly list the authors, including (for each student) first and last name and student number.
- *Summary* of the main features of your programming language (max. 1 page).
- *Problems and solutions.* Problems you encountered and how you solved them (max. 2 pages).
- *Detailed language description.* A systematic description of the features of your language, for each feature specifying
  - Syntax, including one or more examples;
  - Usage: how should the feature be used? Are there any typing or other restrictions?
  - Semantics: what does the feature do? How will it be executed?
  - Code generation: what kind of target code is generated for the feature?

  You may make use of your grammar specification as a basis for this description, but note that not every grammar rule necessarily corresponds to a language feature.

- *Description of the software:* Summary of the JAVA classes and/or HASKELL files you implemented; for instance, for symbol table management, type checking, code generation, error handling, etc. In your description, rely on the concepts and terminology you learned during the course, such as synthesised and inherited attributes, parse tree properties, tree listeners and visitors.
- *Test plan and results.* Overview of the test programs (which themselves are part of the submitted software, as described in §C-4.1). Here you should convince the reader that you have done systematic and extensive testing using correct and faulty test programs, and document how he can re-run the tests.
- *Conclusions*. Your personal evaluation of the language you have defined, as well asl the module as a whole. This is the right place to put your personal thoughts about what you have learned, what you liked and did not like about the module. *The content of the conclusion will not be graded; feel free to write whatever you like. However, the absence of a critical evaluation may decrease your grade.*

Appendices. In addition to the above, your report should also contain the following appendices:

- *Grammar specification.* The complete listing of your grammar(s), in the input format of your chosen parser generator (ANTLR or otherwise).
- *Extended test program.* The listing of one (correct) extended test program, as well as the generated target code for that program and one or more example executions showing the correct functioning of the generated code.

Check the readability of your listings, if necessary by putting them into landscape mode. If you used tabs, make sure the tab stops are the same for your editor and your printout.

(The reason to include listings in your report of files that are also provided in your zip-file is primarily to make it easier to assess your work. The idea is that the report can be used as the basis of the assessment; ideally, it should not be necessary to study your code separately.)

C-5 Assessment

The grade for the final project depends on:

- The language features supported in your programming language (§C-5.1);
- The degree to which you have applied your HASKELL skills (§C-5.2);
- The quality of the final product (code and report) (§C-5.3).

Concretely, the final grade is calulated as a basic grade with modifications; the basic grade is determined by the chosen language features, the modifications by the other two parameters. *The final grade for the project cannot exceed a 10; if the calculated grade is higher — which is possible and has occurred multiple times in the past — it will be "rounded" down to a 10.*

C-5.1 Assessment of the language features.

Table C.1 shows how the *basic grade* of the final project is determined. It should be noted that the basic grade for a language offering the mandatory features only is 6.0; this is sufficient for the final project. We hope you will tackle at least one extension; after all, the basic language is essentially what you already addressed in Block 5 of the CC strand (though you have to generate a different instruction set now), plus rudimentary concurrency features from the CP strand. *If you fail to implement any of the mandatory language features, including the concurrency feature, you cannot get a sufficient grade for the project.*
For a language feature to be awarded (full) points, test program(s) must be included — see §C-3 and §C-5.3.

The suggested (though not enforced) order in which you should consider extensions is:

1. Arrays
2. Procedures/functions, preferably nested, with or without call by reference
3. Strings

Table C.1: Basic grade determined by language features.

Language feature	*Max*
Mandatory part	6.0
Soft division	+0.3
Procedures/functions	+1.0
— also nested	+0.5
— with call-by-reference	+0.5
Exception handling	+1.0
Other extensions	+1.0
Arrays	+1.0
Strings	+0.5
Enumerated types	+0.5
Records	+0.5
Pointers	+1.0
Objects	+1.5
Optimisations	+2.0

4. Other extensions

Additional statement kinds such as `switch`-statements, `for`-statements or **`repeat/until`**-statements are regarded as "more of the same" and do not yield extra points; however, they can be used as reasons to round the final grade up.

The "optimisation" entry in Table C.1 should be read as a maximum: to earn this maximum, you should either implement two of the simpler optimisation strategies, or a more complex one, and also demonstrate their effect on a test program.

C-5.2 Use of HASKELL

The final project is meant to integrate the CC, FP and CP strands of the module. CP is integrated through the concurrency feature of your language; FP is integrated because you are at minimum asked to generate code in the form of a `.hs`-file that can serve as input to the SPROCKELL simulator built especially for the module. However, you are encouraged to go beyond this, by choosing one of the options described in §C-2.

Table C.2 describes how this choice of options can influence your grade.

Table C.2: Grade modifications based on HASKELL usage

Activity	*Max*
JAVA code generation of SPRIL (default)	+0.0
HASKELL code generation	+0.5
HASKELL front-end	+1.0
Extending SPROCKELL	+0.5

C-5.3 Quality of the final product

Not only the choice of language features and compilation process can affect your grade, but also how well you work them out. Aspects of quality are:

- How well have you tested your product? This regards the quality of your tests; see §C-3.
- How well have you structured and documented your grammar (the syntax specification underlying your language) and your code (the hand-written classes that perform the elaboration and code generation phases)? As a computer scientist, writing readable and well-documented code should become second nature; this is one opportunity to show your skills. Consult §C-4.2 to see what you are expected to hand in.

- How well have you written up your results? This regards the quality and completeness of the report you handed in; see §C-4.2.

Each of these aspects will be assessed separately, and can modify you grade either positively (if the aspect is covered particularly well) or negatively (if it is ignored or done badly). The ranges of modification are listed in Table C.3.

Table C.3: Grade modifications based on quality of code and report

Quality aspect	*Range*
Quality of testing	-1.5 . . . +1.5
All tests automatically executable	+0.5
Structure/readability of grammar	-1.0 . . . +0.5
Structure/readability of code	-2.0 . . . +1.0
Completeness/readability of report	-2.0 . . . +1.0

C-5.4 Example scenarios

Here are two examples of how concrete projects may be graded.

John and Mary are hard-core programming freaks, and they just loved the intricacies of concurrent programming. HASKELL, however, never conquered their hearts and minds. They cram in as many features as they can, but forget to test them well, so that in the end some things are not working properly. Also, they completely forgot to work on the report, but they have a hard deadline in the form of holidays (John is planning to visit Sicily with his parents and Mary has planned a trekking tour on Iceland with some friends) so pulling some all-nighters or late submission is not an option.

Their final grade is calculated as

- Mandatory part, functions, ref-parameters, objects, optimisation; $6.0+1.0+0.5+1.5+1.0=10.0$
- Exception handling was planned but didn't work out: $+0.0$
- JAVA code generation of SPRIL: $+0.0$
- Good grammar, poor quality code, really poor tests, minimal report: $+0.7-1.5-1.5-1.5=-3.8$
- Final grade: $10.0+0.0+0.0-3.8=6.2$

Alice and Bob loved the FP side of the module. They choose to implement functions without call by reference and arrays as part of their language, to define the grammar of their language in ANTLR but to do code generation in HASKELL; and also to extend the SPROCKELL architecture. In addition, Alice is a control freak who is only happy when she can extensively test everything, but she is not hot on documenting code she herself already understands, whereas Bob writes beautifully and loves producing a well-written report. However, they do not plan well and in the end submit the result a day late.

Their final grade is calculated as

- Mandatory part, functions, arrays; $6.0+1.0+1.0=8.0$
- HASKELL code generation and SPROCKELL extension: $+0.5+0.5=+1.0$
- Badly structured code, good tests, well-written report: $-1.0+1.3+0.8=+1.1$
- Late submission: -1.0
- Final grade: $8.0+1.0+1.1-1.0=9.1$

C-5.5 Feel like a challenge?

If you do not find this project challenging enough or the requirements unnecessarily restrictive, if you do not want to be constrained by the imperative straightjacket or if you want to propose your own variation: this is possible in principle, but do contact the teacher(s) in time.

Appendix **D**

FP Project

In the FP project we revisit most topics from the exercises. Please read this *entire* document before you start programming.

The project consists of a few parts wherein you create (some of) the following using Haskell:

- Your own parser combinator library (Section D-3)
- Basic parsers defined using your parser combinators (Section D-4)
- An embedded domain specific language (EDSL) with deep embedding for a simple functional language called *µFP*, or "microFP" (Section D-5)
- An evaluator for your EDSL (Section D-5)
- A parser for a grammar for µFP defined using your parser combinators (Section D-6)
- Some further features, including test code (Section D-7)

D-1 µFP

Within the project you develop a parser and evaluator for a language called µFP. µFP is a (pure) functional language with the following grammar:

⟨*program*⟩ ::= (⟨*function*⟩)+
⟨*function*⟩ ::= `identifier` (`identifier` | `integer`)* `':='` ⟨*expr*⟩ `';'`
⟨*expr*⟩ ::= ⟨*term*⟩ | ⟨*term*⟩ (`'+'` | `'-'`) ⟨*expr*⟩
⟨*term*⟩ ::= ⟨*factor*⟩ | ⟨*factor*⟩ `'*'` ⟨*term*⟩
⟨*factor*⟩ ::= `integer`
| `identifier` (`'('` ⟨*expr*⟩ (`','`⟨*expr*⟩)*`')'`)?
| `'if' '('` ⟨*expr*⟩ ⟨*ordering*⟩ ⟨*expr*⟩ `')' 'then' '{'` ⟨*expr*⟩ `'}' 'else' '{'` ⟨*expr*⟩ `'}'`
| `'('` ⟨*expr*⟩ `')'`
⟨*ordering*⟩ ::= `'<'` | `'=='` | `'>'`

The identifiers in this language start with a lower case letter, followed by zero or more numbers and/or lower case letters. Features of the language include multiple arguments, pattern matching, higher order functions and partial function application. Some important differences with respect to Haskell are:

- Function definitions *must* end with a semicolon.
- Function application is not a space; instead the arguments are supplied between parentheses as comma separated list. Note, that this may look a bit funny when partial application is used.
- If-expressions use parentheses for the Boolean expression, and braces around the two alternative expressions.
- No built-in division.

Below are some valid functions for μFP:

```
fibonacci 0 := 0;
fibonacci 1 := 1;
fibonacci n := fibonacci (n-1) + fibonacci (n-2);

fib n := if (n < 3) then {
           1
     } else {
       fib (n-1) + fib (n-2)
     };

sum 0 := 0;
sum a := sum (a-1) + a;

div x y :=
   if (x < y) then
     {
       0
     } else {
       1 + div ((x-y), y)
     };

twice f x := f (f (x));
double a := a*2;

add x y := x + y;
inc := add (1);
eleven := inc (10);

fourty := twice (double, 10);


main := div (999, 2);
```

D-2 Submission and grading

D-2.1 Rules

- Both students in a group are expected to contribute to all parts of the project, and individual students can be asked to explain all code.
- Without *all* the mandatory features you cannot receive a grade above 4.
- Submission follow the submission instructions (Section D-2.4). Up-to one full point could be subtracted when they are not strictly followed. When a submission severely deviates from the instructions, it is considered as “no submission” and is not graded.
- All features are in the correct file, and comments clearly indicate where a feature is implemented using the feature identifier (**FPxx.yy**). If we do not easily find or test a feature, no points are rewarded for it.

- All submissions are checked for code similarity (both manually and using software) and for other types of fraud. Suspicion of fraud or enabling it (e.g., sharing solutions in a open repository) is always reported to the Board of Examiners. For fraud we use the definition below.
- Some parts need to be signed off during the supervised sessions (see Section D-2.2). The hard sign-off deadline is on 05-06-2019.

Definition of fraud, from the Faculty of Applied Science (TUD):

> Intentional acts or omissions on the part of a student, which render correct or fair evaluations of his/her knowledge, insight of skills totally or partially impossible.

D-2.2 Assessment

The grading of your work is based on the following criteria:

- Readability and style of the Haskell code,
- Use of the appropriate functional programming concepts and constructs,
- Use of the appropriate abstractions, e.g., higher order functions, lambda abstraction, type classes, (Applicative) Functors and Monoids,
- Comments and small tests should be available for all functions and features,
- Features (indicated with FP1.1 to FP5.6); the *maximum* points are indicated in the tables within the sections D-3 to D-7.

The grade is calculated as follows:

$$\text{grade} = \max(\min(100, x), 10)/10,$$

where x is the number of points you received (again, see the tables below). In order to give you some flexibility in what topics to focus on, you can earn more than 100 points. Use this flexibility wisely because of the assessment criteria above. Note, that it may be better to omit features if otherwise the readability of your code as a whole would suffer.

> **Important:** before you submit your work, the following must be signed off during the supervised project sessions (hard deadline: 05-06-2019):
>
> - The definition of the EDSL (FP3.1),
> - The set of mandatory features,
> - When you choose to implement error handling: show the data types you have in mind for FP5.1.
>
> Note, that signing off means that the TA is aware of your progress. It is not an indication that your code is correct.

D-2.3 Suggested and mandatory features

Some of the features are mandatory since these are essential for (almost) all other features and the structure of the project (see Table D.1). Some of the optional features are (strongly!) suggested to implement, since they make later exercises significantly easier (see Table D.1). Section D-7 contains some extensions that are best postponed until all the suggested features are implemented. The tables with features below indicate next to the points also if a feature is mandatory (M) or strongly suggested (S).

D-2.4 Available code and submission

On Canvas you can find the file `FP-Project.zip`, which contains files that you need to change. You may use all the code in these files, and also GHC, twentefp-eventloop-trees, and the entire Prelude (the standard

	Features
Mandatory	FP1.1, FP1.2, FP1.3, FP1.5, FP1.6, FP2.2, FP3.1, FP3.2, FP4.1
Strongly suggested	FP1.4, FP2.1, FP2.3, FP2.4, FP3.3, FP4.2

Table D.1: Mandatory and suggested features

libraries) except for the parser combinator functions. Note, using ParSec or any other predefined parser combinators is not allowed.

Since the files are processed automatically, do not rename the files, take care of file name capitalization, etc. At submission, use .zip (**important**: PKZip — not another compression format) with *exactly* the following directory structure and files:

- `sxxxxxxxx_syyyyyyyy/PComb.hs`
- `sxxxxxxxx_syyyyyyyy/BasicParsers.hs`
- `sxxxxxxxx_syyyyyyyy/MicroFP.hs`

Here `sxxxxxxxx` and `syyyyyyyy` should be replaced by the student numbers of the two students in the group (use `sxxxxxxxx` instead of `sxxxxxxxx_syyyyyyyy` when you work alone). For your convenience, you may consider using `MakeZip.hs` from `FP-Project.zip` to create the .zip-file.

D-3 Parser combinator library

In `PComb.hs` you can find some example code for parser combinators. Change/add/extend the parser combinator library (`PComb.hs`) to support (some of) the features below:

	Points	Description
FP1.1	2 (M)	The parser can receive a "stream" (see `Stream`) of **`Char`**s and result in some type `a`. This implies that a parser is of type `Parser a`, where `a` is the type of the parse result.
FP1.2	3 (M)	The parser has an appropriate **`Functor`** instance.
FP1.3	1 (M)	The function `char :: Char -> Parser Char` parses a single (given) **`Char`**.
FP1.4	2 (S)	The function `failure :: Parser a` is a parser that consumes no input and fails (produces no valid parsing result).
FP1.5	2 (M)	The parser has an `Applicative` instance for the sequence combinator.
FP1.6	2 (M)	The parser has an `Alternative` instance that tries as few alternatives as possible (i.e., until the first parser succeeds).
FP1.7	5	The parser has a `Monoid` instance that tries *all* alternatives.
FP1.8	3	Define a `Monoid` wrapper for `Parser a`, which combines parsers sequentially (e.g., using the `Applicative`). You may assume (use a class constraint!) that `a` is also a Monoid.
Total	**20**	

Hints/suggestions:

- `many` and `some`: study look at how they are recursively defined (for this, look them up on Hoogle). The definitions rely on lazy evaluation, while pattern matching constructors enforces evaluation. You can work around this by using `runParser`.

D-4 Basic Parsers

Next to the core parser combinators, several basic parsers/combinators are needed to make our parsing library useful. To accomplish this, define the following data types and functions in `BasicParsers.hs`. Make sure to use already defined parsers and combinators, instead of operating on the input stream directly.

	Points	Description
FP2.1	5 (S)	Define the parsers `letter :: Parser` **`Char`** that parses any (alphabetical) letter, and `dig :: Parser` **`Char`** that parses any digit.
FP2.2	2 (M)	The following parser combinators: [1pt] `between :: Parser a -> Parser b -> Parser c -> Parser b` runs the three parsers in sequence, and returns the result of the second parser. Similar to `between` in ParSec. [1pt] `whitespace :: Parser a -> Parser a` receives a parser `p` and uses it to parse the input stream, while skipping all surrounding whitespaces (space, tab and newline). For example, `whitespace (char 'a')` can be used to parse the input `'a'` surrounded by whitespace.
FP2.3	3 (S)	The following parser combinators: [1pt] `sep1 :: Parser a -> Parser b -> Parser [a]`. The parser `sep1 p s` parses one or more occurrences of `p`, separated by `s`. This can, for example, be used to parse a comma separated list. [1pt] `sep :: Parser a -> Parser b -> Parser [a]`. The parser `sep p s` works as `sep1 p s`, but parses zero or more occurrences of `p`. [1pt] `option :: a -> Parser a -> Parser a`. `option x p` tries to apply parser `p`; upon failure it results in `x`. Similar to `option` in ParSec.
FP2.4	10 (S)	The following parsers and combinators: [3pt] `string :: String -> Parser String` parses an given **`String`**, similar to the function `char`. [2pt] `identifier :: Parser String` parses a given identifier surrounded by whitespace. [2pt] `integer :: Parser Integer` parses an integer surrounded by whitespace. [1pt] `symbol :: String -> Parser ()` parses a given **`String`** surrounded by whitespaces. [1pt] `parens :: Parser a -> Parser a` parses something using the provided parser between parentheses, i.e., `(...)`. [1pt] `braces :: Parser a -> Parser a` parses something using the provided parser between braces.
Total	**20**	

D-5 Embedded Domain Specific Language

Define an EDSL and interpreter that correspond to the μFP grammar by defining at least the following data types and functions in `MicroFP.hs`:

	Points	Description
FP3.1	5 (M)	A set of type/data constructors that represent an EDSL (deep embedding) for μFP, i.e., it describes the same functionality. Note, that this EDSL can be made very compact compared to the grammar. Be creative in your design!
FP3.2	2 (M)	Define the following functions in your μFP EDSL. These must correspond to the definitions in `functions.txt` • `fibonacci` receives a single argument "n", and expresses the calculation of the n-th fibonacci number. • `fib` receives a single argument "n", and expresses the calculation of the n-th fibonacci number. • **`sum`** receives one argument "a", and calculates the sum from 1 to a • **`div`** receives two arguments "x" and "y", and divides "x" by "y" • `twice` receives two arguments"f" (a function) and "x" (a value), and calculates $f(f(x))$ • A data structure that describes `add`, `inc` and `eleven` from `functions.txt`
FP3.3	3 (S)	`pretty :: ? -> String` is a pretty printer that generates a textual representation that corresponds to the grammar of μFP. Here `?` corresponds to your EDSL.
FP3.4	15	`eval`, which is a evaluator for your μFP EDSL without support for partial application, lazy evaluation, pattern matching and higher order functions. Since pattern matching is not (yet) supported, you may assume constants are not used at the left-hand side (e.g., `fibonacci` does not work with your evaluator yet).
Total	**25**	

Hints/suggestions:

- FP3.1: Define built-in μFP functions for the operators, e.g., "_add" corresponds to `+`.
- FP3.4: Define a function `bind` that receives an expression, a list of (variable) names and a list of values/expressions. The function replaces each occurrence of a variable by the corresponding value/-expression.
- FP3.4: Define a function `reduce` that receives an expression and reduces it to a simpler expression, and uses `bind` where needed.

D-6 Parser

Define a parser that translates a textual representation of μFP to your μFP EDSL. For this, define the following functions in `MicroFP.hs`, where `?` corresponds to one or more types from your μFP EDSL:

	Points	Description
FP4.1	10 (M)	Define the parsers `factor :: Parser ?`, `expr :: Parser ?`, `term :: Parser ?` and the remaining parsers needed to parse μFP. Make sure `functions.txt` is parsed properly and leads to similar definitions as in FP3.2.
FP4.2	5 (S)	`compile :: String -> ?` tokenizes and compiles a textual representation of μFP to your EDSL.
FP4.3	5	`runFile :: FilePath -> [Integer] IO Expr` results in an `IO` action that reads the specified file, compiles it, and finally uses `eval` (from FP3.4) to evaluate it with the list of integers as input to the μFP function. When the file contains multiple functions, use the *last* function in the file.
Total	**20**	

D-7 Further features

	Points	Description
FP5.1	15	Implement error handling as follows: • A parsing failure results in an error of the form: "Parse error at 4:3, expected 'a"', where 4 is the line number and 3 the column where the error occurred, • expected alternatives can be reported (e.g., when `<|>` is used), for example: "Parse error at 4:3, expected 'a' or 'b"', • a parser combinator `(<?>) :: Parser o -> String -> Parser o` that creates error messages; as is also available in ParSec, • appropriate changes to all the tokenizer code to support error handling, • when parsing fails, `compiler` uses `error` to report the error message. Note that error handling has a major impact on every aspect of the project.
FP5.2	7	Implement pattern matching for `eval`. The μFP functions `sum` and `fibonacci` must work correctly.
FP5.3	3	Implement the function `patmatch`, which rewrites function definitions with pattern matching (in your EDSL) to a single definition with `if`-expressions. The μFP functions `sum` and `fibonacci` must work correctly.
FP5.4	5	Implement partial application for `eval`. The μFP function `inc` must work correctly.
FP5.5	5	Implement higher order functions for `eval`. Now the μFP function `twice` works correctly **(FP5.4 must be implemented first)**.
FP5.6	5	Test your compiler framework using QuickCheck as follows: • Make your EDSL an instance of `Arbitrary` • Generated function names are strings of length 3 containing 'f', 'g', 'h' and/or 'k'. Generated identifiers are strings of length 3 containing 'a', 'b', 'c', 'd' and/or 'e'. • Define a QuickCheck property that checks if a randomly generated EDSL expression, after pretty printing and compilation, remained the same. Make sure the data structures have a variable, but moderate size. • Define a QuickCheck property that checks if a randomly generated EDSL application/program, after pretty printing and compilation, remained the same. Make sure the data structures have a variable, but moderate size.
Total	**40**	