HOGESCHOOL ROTTERDAM / CMI

Functional programming 2

TINFUN02

ECTS:	3
Module responsible:	Giuseppe Maggiore

Contents

1 Introduction

This document describes the development of the fully new *Functional programming II* course at *Technische Informatica*, a department within the *Communication and Multimedia Institute* of the *Hogeschool Rotterdam*.

1.1 Course background

The course is intended as an advanced subject for students who are completing their second year. The overall goal is that of strengthening the students' fundamental understanding of programming concepts by offering two new courses (*Functional programming I* and *Functional programming II*) that make use of a programming paradigm, functional programming. This paradigm is unfamiliar to students while at the same time being fundamentally connected with their previous programming knowledge, therefore offering a new perspective on programming as a whole. The students beginning the course have passed numerous exams of basic (imperative) programming and mathematics. Students are particularly used to an effective working and studying style when it comes to practical programming assignments: moreover, students are especially motivated when it comes to such practical assignments since their expectation is to become good programmers.

2 Schematic module description

Module name:	Functional programming 2
Module code:	TINFUN02
Number of ECTS and number of individual study hours:	This module gives 3ECTS, which corresponds to 84 hours. • 8 × 120 minutes frontal lecture • 8 × 120 minutes practicum • 12 × 120 minutes individual study
Examination:	Practical assignments
Course structure:	Lectures and practicums
Required knowledge:	The previous functional programming course.
Learning tools:	• Book: Types and Programming Languages, author: Benjamin Pierce • Book: Friendly F# (Fun with game programming Book 1), author: Giuseppe Maggiore, Giulia Costantini • Text editors and IDE's: Emacs, Notepad++, Visual Studio, Xamarin Studio, etc.
Content:	• The model of an interpreter and of a compiler; • Concepts of formal languages (Chomski hierarchy); • Type 3 and type 2 languages; • Monads as abstraction mechanisms for the management of complex operators such as those found in parser; • Type checking; • Concepts of denotational and operational semantics; • Interpretation; • Code generation.
Module responsible:	Giuseppe Maggiore
Date:	May 27, 2015

2.1 Learning goals

In this subsection we describe the learning goals, connected with Bloom's taxonomy.

Overall learning goal:	The student is able to describe, define, and then translate a programming language to a working interpreter or compiler.
Analysis	The student is able to distinguish the components of a programming language, that is: grammar, type system, and semantics
Advice	The student is able to give advice over the design and realisation of a programming language (learning goal *advice*)
Creation [1]	The student is able to design the structure and architecture of a functional interpreter or compiler (learning goal *design*)
Creation [2]	The student is able to build a working interpreter or compiler in an ML language (learning goal *realization*)
Evaluation [3]	The student is able to communicate in correct Dutch or English, using the correct jargon, about programming languages, compilers, interpreters, type systems, etc. (learning goal *communication*).

[1]Competence *design* in the CMI handbook
[2]Competence *realization* in the CMI handbook
[3]Competence *communication* in the CMI handbook

3 General description and motivation

The overall goal of the course is to provide a detailed answer to the questions:

- **What is a programming language?**
- **How does a programming language concretely work?**

In this section we discuss further the full breadth of what the course covers, plus the desired level of skills achieved by the students.

3.1 Introduction

Students use programming languages as their most important tool to express and automate solutions to problems. During the first courses students learn programming language concepts from an intuitive standpoint. As a consequence of this intuition the use of programming languages by students is often superficial and unsure, because the fundamental questions of the inner working of languages remain unanswered [5].

The goal of the course is to provide a precise definition of the structure (grammar and type system) and interpretation (semantics) [1–3] of programming languages. Moreover, we shall learn how to translate this definition into a working interpreter or compiler written in a dialect of the ML [4] programming language.

Through the knowledge acquired during this course students will gain awareness of the underlying mechanisms of programming languages. Moreover, students will become able to build their own programming languages (such as scripting languages, domain specific languages, etc.).

3.2 Relationship with competencies and professional profile

The course strengthens for the most the competencies of **analysis, advice, design, and realization** within the area of **software development**. Implicitly the course also supports the competency of **process handling**, in that students are expected to organize their work-flow effectively.

The final result is that this course will also strengthen the professional profile of students as software developers.

3.3 Relationship with other teaching units

This module builds over the module of functional programming 1, and is also strongly connected with previous programming knowledge about imperative programming, algorithms, and data structures. Working understanding of basic mathematical reasoning is also a good supporting skill for this course.

3.4 Learning tools

Obligatory:

- Presentations and sources presented during lectures (found on N@tschool);
- Video's of lectures;
- Assignments to work on during practicums (found on N@tschool);
- Text editors: Emacs, Notepad++, Visual Studio, Xamarin Studio, etc.

Facultative:

- Book: Types and Programming Languages, author: B. Pierce;
- Book: Semantics of Programming Languages, author: C. Gunter;
- Book: Friendly F# (Fun with game programming Book 1), authors: G. Maggiore, G. Costantini.

4 Lectures

In this section the topics of the various lectures and the lecture structure is discussed.
Each lecture is roughly divided into a series of phases alternating listening, active participation, and formative feedback, as per the model(s) described in [17,18]. The phases are structured so that attention of the students is maximized through activity and participation. Moreover, through active assignments the students will regularly get formative feedback. This structure is also inspired by [19].

4.1 Course activities

In this section we present a list of topics and activities, divided per-lecture.

Lecture	**Self-study**	**Lecture topics (3 hour each)**	**Lecture activities**	**Learning goals**
Structure of an interpreter and a compiler		Parsing → type checking → code generation → execution	Discussion, formal lecture	analysis, advice
Parsing - part I	*i)* chapters 3 and 1 of the reader *ii)* assignments 0 and 1	*i)* Chomski's hierarchy *ii)* Type 3 languages (regular expressions) *iii)* State machine parser for regular expressions	Discussion, formal lecture, practical exercises	analysis, advice, design, realization
Parsing - part II	*i)* chapter 3 of the reader *ii)* assignment 2A	*i)* Type 2 languages *ii)* Pumping lemma (stack/recursion requirements for type 2 languages) *iii)* Data structures for an *abstract syntax tree* (AST) *iv)* Recursive *top-down* parser with *look-ahead*	Discussion, formal lecture, practical exercises, **handing-in of assignments 0 and 1**	analysis, advice, realization
Monads	*i)* chapter 5 of book Friendly F# *ii)* assignments 2B and 3	*i)* Maybe *ii)* List *iii)* State *iv)* State + Maybe *v)* State + List *vi)* State + List + Maybe *vii)* Parser monad	Discussion, formal lecture, practical exercises, **handing-in of assignment 2A**	analysis, advice, realization
Type checking	*i)* chapters 8 and 9 of TAPL book *ii)* assignments 4 and 5	*i)* Type systems *ii)* Type checking inference rules	Discussion, formal lecture, practical exercises	analysis, advice, design, realization
Semantics	chapters 1 and 2 of SPL book	*i)* Interpretation *ii)* Code generation	Discussion, formal lecture, practical exercises, **handing-in of assignments 2B and 3**	analysis, advice, design
Further static analyses: *abstract interpretation*		*i)* Approximation of semantics *ii)* Abstract domains *iii)* Galois connection	Discussion, formal lecture, practical exercises, **handing-in of assignments 4, 5**	analysis, advice, communication
Total hours:	**63**	**21**		

5 Testing and evaluation

In this section we discuss the testing procedure of this course, and the grading criteria.

5.1 Overall description

At the beginning of each lecture (with exclusion of the first) students will perform a very short, formative test. The tests are made up of either a few (between three and five) multiple choice questions about the topics of the previous lecture or a discussion about the expectations over the topics of the coming lecture. This is done in order to cement and reinforce the knowledge needed for the new lecture, and to connect with the students expectations and previous knowledge.

This module is also tested with a series of summative, practical assignments. The starting Visual Studio solutions for the assignments can be found on N@tschool. Since a purely theoretical handling of the course topics would be of little usefulness within our students' future jobs, we have chosen for an applied form of testing, in order to strengthen the students' programming competencies and add further applied skills to their toolbox.

5.2 Practical (summative) assignments

Foreword and notes:

- The practical assignments determine the final grade.
- The practical assignments are made up of elements of an interpreter (or in he last assignment a compiler) which is either incomplete or wrongly built. The students task is that of extending or fixing such elements.
- The practical assignments must contain extensive, individually written documentation.

This manner of examination is chosen for the following reasons:

- By reading existing sources students must read and reason about code (learning goals *analysis* and *advice*).
- By correcting or extending the sources students must write code (learning goals *design* and *realisation*).
- By writing documentation students must communicate about their code (learning goals *analysis*, *advice*, and *communication*).

The grade of each practicum assignment is determined by:

- The correctness of pure, functional code (use of `mutable` and `ref` is absolutely forbidden) (60%).
- Completeness and clarity of the documentation (20%).
- Use of functional programming patterns and idioms as seen during the lectures (20%).

In this subsection the specific assignments are discussed. These assignments are summative, and make up the whole exam.

Assignment 0 - defining the tokens (handed in at the beginning of the third lecture)
Students must complete the definition of the `Token` data structure.
Grading criteria:

Criterion:	**Requirements for criterion**
Coverage	Defined tokens must cover the whole expressions of the chosen language.
Readability	Defined tokens must have names that reflect standard naming conventions.
Encapsulation	Defined tokens must be grouped into data structures based on functionality.

Assignment 1 - regular expressions parsing (handed in at the beginning of the third lecture) Students must define a tokenizer function to parse the tokens of a small λ-calculus or **imp** program with a simple recursive function:

```
let rec tokenize (l:List<char>) : List<Token> = ...
```

Grading criteria:

Criterion:	**Requirements for criterion**
Correctness	The parser must not crash, and produce the appropriate list of tokens.
Readability	*i*) Functions and identifiers must have names that reflect standard naming conventions; *ii*) Indentation must clearly separate semantically different blocks of code.
Encapsulation	*i*) Separate code blocks handle separate tokens; *ii*) Separate functionality is split into separate functions; *iii*) Long blocks are split into separate functions.

Assignment 2A - type 2 parsing (handed in at the beginning of the fourth lecture) Students must define a function to parse a small λ-calculus or **imp** program with a simple recursive function:

```
let rec parseTerm (ts:List<Token>) : Term * List<Token> = ...
```

Students must also defined their own `Term` data structure as the parsing target.
Grading criteria:

Criterion:	**Requirements for criterion**
Coverage	Defined terms must cover the whole expressions of the chosen language.
Readability of terms	Defined terms must have names that reflect standard naming conventions.
Encapsulation of terms	Defined terms must be grouped into data structures based on functionality.
Correctness	The parser must not crash, and produce the appropriate hierarchical representation of terms.
Readability of code	*i*) Functions and identifiers must have names that reflect standard naming conventions; *ii*) Indentation must clearly separate semantically different blocks of code.
Encapsulation of code	*i*) Separate code blocks handle separate sub-terms; *ii*) Separate functionality is split into separate functions; *iii*) Long blocks are split into separate functions.

Assignment 2B - parsing with a parsing monad (handed in before the end of the last week of lecture) Students must define a function to parse a small λ-calculus or **imp** program with a parsing monad:

```
let rec parseTerm : Parser<Term,List<Token>> = ...
```

Grading criteria:

Criterion:	**Requirements for criterion**
Monad laws	The parser monad must be correctly defined according to the monad laws.
Correctness	The parser monad produce the appropriate term hierarchy.
Readability	*i*) Monadic combinators and identifiers must have names that reflect standard naming conventions; *ii*) Indentation must clearly separate semantically different blocks of code.
Encapsulation	*i*) Separate code blocks handle separate tokens; *ii*) Type 3 and type 2 combinators are split into separate modules; *iii*) Separate parsing units are split into separate combinators; *iv*) Long blocks are split into separate functions.

Assignment 3 - type system (handed in at the beginning of the sixth lecture) Students must define a function to apply the rules of the type system of the λ-calculus or **imp** programming languages:

```
let rec typeCheck : Term -> Option<Type> = ...
```

Students must also defined their own `Type` data structure as the typing information container.

Grading criteria:

Criterion:	Requirements for criterion
Coverage	Defined types must cover the whole type system of the chosen language.
Readability of types	Defined types must have names that reflect standard naming conventions.
Encapsulation of types	Defined types must be grouped into data structures based on semantics.
Correctness	The type-checker must not crash, and produce the appropriate representation of types.
Readability of code	*i*) Functions and identifiers must have names that reflect standard naming conventions; *ii*) Indentation must clearly separate semantically different blocks of code.
Encapsulation of code	*i*) Separate code blocks handle separate sub-terms; *ii*) Separate functionality is split into separate functions; *iii*) Long blocks are split into separate functions.

Assignment 4 - interpretation (handed in at the beginning of the eighth lecture) Students must define a function to determine the final result of the execution of the parsed program:

```
let rec eval (p:Term) : Result = ...
```

Students must also defined their own `Result` data structure as the evaluation result.
Grading criteria:

Criterion:	Requirements for criterion
Coverage	Defined results must cover the whole possible results of the chosen language.
Correctness	The evaluation function must not crash, and produce the appropriate result.
Readability of code	*i*) Functions and identifiers must have names that reflect standard naming conventions; *ii*) Indentation must clearly separate semantically different blocks of code.
Encapsulation of code	*i*) Separate code blocks handle separate sub-terms; *ii*) Separate functionality is split into separate functions; *iii*) Long blocks are split into separate functions.

Assignment 5 - compilation (handed in before the end of the last week of lecture) Students must define a function to write code that, when executed, shall determine the final result of the execution of the parsed program:

```
let compileAndRun (p:Assignment2.Term) : obj = ...
```

Grading criteria:

Criterion:	Requirements for criterion
Correctness	*i*) The compilation function must not crash; *ii*) The resulting code must compile correctly; *iii*) The resulting code must have the same semantic interpretation of the original program.
Readability of code	*i*) Functions and identifiers must have names that reflect standard naming conventions; *ii*) Indentation must clearly separate semantically different blocks of code.
Encapsulation of code	*i*) Separate code blocks handle separate sub-terms; *ii*) Separate functionality is split into separate functions; *iii*) Long blocks are split into separate functions.

5.3 Grades

Assignments 0, 1, 2A, and 4 are strictly needed to get a passing grade. With these assignments the maximum grade possible is a seven. Assignments 2B, 3, and 5 each increase the maximum possible grade by one point.

Score of each assignment The scores of the assignments are calculated based on the expected time that we estimate each assignment will cost to the students.

Assignments:	**Value in grades**
0	1
1	2
2A	2
4	2
2B	1
3	1
5	1

Score and requirements Each requirement of each assignment is judged as either **fully correct** or **fully incorrect**. The total number of **fully correct** requirements is divided by the number of requirements and thus the **correct percentage** of the assignment is determined. The **correct percentage** is then multiplied by the score value of the assignment, and that is then multiplied by the value of the assignment.
For example, consider assignment 2A. The number of requirements of the assignments is 6. Assume that the student has completed the assignment with 4. The score of the assignment is 2, so the student gets a total of points for this assignment of:

$$\frac{4}{6} \times 2 = 1.3$$

5.4 Handing-in

Each assignment must be handed in printed on paper by the deadline indicated in the previous subsection for each assignment, either directly to the teacher or to the *student-balie* of *CMI*. On each assignment must be clearly specified and legible: the name of the student, his student number, and the course code the assignment refers to.

5.5 Herkansing

Herkansing can be done by handing in the assignments before the end of week 1 of the following period (that would be right after the summer holiday).

5.6 Feedback

It is possible to discuss the assignments (and their evaluation) during the practicum lectures, or during week 10 of the period (the same holds for the *herkansing*, but in reference to the following period).

5.7 Quality, validity, and reliability

In this subsection the quality of the exam is motivated: the validity of the content and concepts, the reliability, and the transparency of the exam are all discussed.

Validity of content and concepts The overall goal of the course is to teach students how programming languages work. Since programming languages are clear technical products, then building a full interpreter or compiler (even for an apparently simple programming language) is sufficient to gain such an understanding. Moreover, wrong understanding of the theoretical concepts will most likely lead to practical mistakes, thus this form of testing is also more appropriate for adequate learning of (the consequences of) the theory.

Coverage of goals The practical assignments cover the whole learning goals extensively (see Section A); this can be trivially verified by observing that a programming language is precisely made up of parsing (Assignments 0, 1, 2A, and 2B), type checking (Assignment 3), and execution (Assignments 4 and 5). This means that successful completion of the assignments corresponds to fulfillment of the learning goals.

Transparency of grading criteria The grading criteria are, for the most part, rather objective. Correctness of the code can be verified empirically through testing [10] and implicitly thanks to the ML type checker [2]. This means that students know before hand whether or not their implementation works, and as such can already gauge their grade without having to expect surprises.
Documentation needs to be done according to the current standards for documentation of programming code [9]. Documentation must specify input and output conditions or invariants [8], and how these are connected by the code within each function.
Idioms are also very unambiguously specified. Functions and data structures must: *(i)* adhere to the *single responsibility principle* [6]; and *(ii)* be *referentially transparent* [7]. Both properties can be verified objectively and without appeal to intuition.

References

[1] J. Hopcroft, J. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[2] B. Pierce, *Types and Programming Languages*, MIT Press, 2002.

[3] H. Abelson, G. J. Sussman, J. Sussman, *Structure and interpretation of computer programs*, MIT Press, 1996.

[4] R. Milner, M. Tofte, R. Harper, *The Definition of Standard ML*, MIT Press, 1990.

[5] P. Naur, *Intuition in Software Development*, TAPSOFT, 1985.

[6] T. De Marco, *Structured Analysis and System Specification*, 1979.

[7] H. Søndergaard, P. Sestoft, *Referential transparency, definiteness and unfoldability*, Acta Informatica, 1990.

[8] C. Gunter, *Semantics of Programming Languages*, MIT Press, 1992.

[9] D. E. Knuth, *Literate Programming*, Stanford University Center for the Study of Language and Information, 1992.

[10] G. J. Myers, *The Art of Software Testing*, Wiley, 1979.

[11] E. Moggi, *Notions of computation and monads*, Information and computation, 1991.

[12] G. Hutton, E. Meijer, *Monadic parsing in Haskell*, under consideration for publication in J. Functional Programming.

[13] F. Nielson, H. R. Nielson, C. Hankin, *Principles of Program Analysis*, Springer, 1999.

[14] P. Hilfinger, A. Dave, *CS164 - Programming languages and compilers*, Berkeley University, 2015 edition.

[15] W. Weimer, *CS 4610 — Programming Languages*, University of Virginia, 2015 edition.

[16] M. Rinard, S. Amarasinghe, *Computer Language Engineering (SMA 5502)*, MIT, 2015 edition.

[17] H. Berkel, A. Bax, D. J. ten Brinke, *Toetsen in het hoger onderwijs*, Bohn Stafleu van Loghum, 2014.

[18] L. Dee Fink, *Creating Significant Learning Experiences: An Integrated Approach to Designing College Courses*, Jossey-Bass, Wiley, 2003.

[19] T. Gray, L. Madson *Ten Easy ways To Engage Your Students*, College teaching 55, 2007.

A Examination matrix

The learning goals are covered orthogonally by the assignments. This means that each assignment requires the students to analyze, advise, design, realize, and communicate about a separate aspect of the analysis of programming languages:

Learning goal	Dublin descriptors	Assignments
The student is able to distinguish the components of a programming language, that is: grammar, type system, and semantics	1, 5	All
The student is able to give advice over the design and realisation of a programming language	1, 3, 5	Documentation of 0, 2A, 4/5
The student is able to design the structure and architecture of a functional interpreter or compiler	1, 3, 4	0, 2A/2B
The student is able to realise a working interpreter or compiler in an ML language	2	4, 5
The student is able to communicate in correct Dutch or English, using the correct jargon, about programming languages, compilers, interpreters, type systems, etc.	2	Documentation of all

Dublin-descriptors:

1. Knowledge and insight
2. Application of knowledge and insight
3. Making judgments
4. Communication
5. Learning skills