CHALMERS | UNIVERSITY OF GOTHENBURG

Lollipop

A general purpose, functional programming language with linear types

Bachelor's thesis in Software Engineering and Computer Science

EDVARD HÜBINETTE
JOHAN ANDERSSON
JONATHAN JOHANSSON
MARIE KLEVEDAL
MIKAEL MALMQVIST

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden 2016

BACHELOR'S THESIS IN SOFTWARE ENGINEERING AND COMPUTER SCIENCE

Lollipop

A general purpose, functional programming language
with linear types

EDVARD HÜBINETTE
JOHAN ANDERSSON
JONATHAN JOHANSSON
MARIE KLEVEDAL
MIKAEL MALMQVIST

Department of Computer Science and Engineering

CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

Göteborg, Sweden 2016

Lollipop
A general purpose, functional programming language
with linear types
EDVARD HÜBINETTE
JOHAN ANDERSSON
JONATHAN JOHANSSON
MARIE KLEVEDAL
MIKAEL MALMQVIST

ISSN 1654-4676
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Cover:
The logotype of Lollipop

Lollipop
A general purpose, functional programming language
with linear types
Bachelor's thesis in Software Engineering and Computer Science
EDVARD HÜBINETTE
JOHAN ANDERSSON
JONATHAN JOHANSSON
MARIE KLEVEDAL
MIKAEL MALMQVIST
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Abstract

This project designs and implements a general-purpose functional programming language with linear types, called Lollipop. The purpose is to investigate how linear types can be a part of modern programming languages. The language should also act as a pedagogical platform for developers to get acquainted with the concept, because linear types are not implemented in any mainstream language on the market. The focus is not to deliver a well-polished product ready for deployment, but rather to evaluate the development process of linear types in a functional language with its complications. The language should, however, have a rigorous type system and have the basic functionality of a functional language, as well as the addition of linear type variables and basic user interaction. The compiler front end of the language was developed using BNFC while the interpreter, inner abstract syntax tree, conversion from surface syntax, type inferencing and other extensions were written in Haskell. The project was done using agile development cycles and milestones and resulted in a working proof-of-concept having the planned usability, albeit with some blemishes. Lollipop can be used as a base for further development as well as a tool for learning the basics of linear types for the common developer.

Keywords: Functional programming, Linear types, BNFC, Haskell

Sammanfattning

Det här projektet uformar och implementerar ett universellt, funktionellt programmeringsspråk med linjära typer vid namn Lollipop. Det har gjorts med syftet att undersöka hur linjära typer kan integreras med moderna programmeringsspråk. Det kommer även att tjäna som en pedagogisk plattform för utvecklare att bekanta sig med konceptet linjära typer. Detta behövs eftersom de inte är implementerade i något av de vanligt förekommande språken på marknaden, vilket gör att det huvudsakligen kan ses som ett begrepp inom den akademiska sfären. Målet är inte att leverera en produkt redo för marknaden utan att utvärdera och dokumentera utvecklingen av språket och dess problematik. Språket ska dock vara försett med ett avancerat typsystem och den grundläggande funktionalitet som ingår i funktionella programmeringsspråk, samt linjära typer och elementär användarinteraktion. Språkets kompilatorfront skapades med BNFC medan programtolken, intern språkrepresentation, konvertering från ytsyntax samt typkontroll är skrivna i Haskell. Utvecklingen skedde med hjälp av agil utvecklingsmetodik samt milstolpar och projektet har resulterat i ett fungerande konceptbevis som omfattar den planerade användbarheten, om än med några skönhetsfläckar. Resultatet kan användas som en bas för vidare utveckling och för att få en grundläggande förståelse av linjära typer.

All code for this project can be found at github.com/m0ar/lollipop

Acknowledgements

We would like to thank our thesis advisor Mr. Jonas Duregård of the Department of Computer Science and Engineering at Chalmers University as he has been of great importance for this project. While always letting us choose our own direction, he has been dedicated and constantly willing to share his ideas and experience when we were in need.

Glossary

Abstract syntax	Simplified representation of the source code that only keeps the significant parts needed for analysing.
AST	An abstract syntax tree represents the source code with a tree representation of the abstract structure.
BNF	Backus-Naur Form, a notation used for context free grammars.
BNFC	BNF converter, a tool used to generate a compiler front-end from a BNF grammar.
Compiler	A program that transforms programming code from a source language into a target language, most often into object code.
Context free grammar	A grammar which rules are independent on the context in which they are applied.
General purpose programming language	A programming language that is applicable in several different domains.
Grammar	A set of rules that regulate what is a valid language.
Interpreter	A program executing code written in a high-level language.
Lambda calculus	A family of formal systems for the notation of any computable function.
Lexer	A program that converts a string to a list of tokens.
Lexical preprocessor	A program that is run after the lexer and before the parser. It modifies the list of tokens produced by the lexer to the version which the parser requires.
Linear logic	A branch of logic where each truth value may only be used exactly once.
Parser	A program that converts a list of tokens to an AST.
Programming paradigm	A style of computer programming which is based on a coherent set of principles.
Race condition	A situation that occurs when two or more threads try to change shared data simultaneously resulting in an unpredictable and, potentially undesirable, outcome.
REPL	Short for read, evaluate, print-loop. A part of a language tool chain where code can be entered and evaluated repeatedly.
Semantics	The meaning of syntax in a language.
Syntactic sugar	Expressions that can be expressed in terms of other expressions and hence are superfluous.
Syntax	Rules of how symbols can be combined legally in a language.
Transpiler	A program converting source code from one programming language to another.
Type checker	A program that checks another program for type errors.
Type inferencer	A program that given an expression determines its type, by checking the types of its sub-expressions and using the type rules for the language in which the expression is written.
Type system	A formal system stating a set of rules in addition to the grammar of how expressions and functions can legally be used.

Contents

1 Introduction

As of today, there are a number of different programming paradigms commonly used in mainstream programming languages. Likewise, there are a great deal of different type systems, setting the most fundamental structural rules of how values are legally used in a language. These type systems are implemented for various purposes, thus providing a language with its own characteristics. A type system spawned from linear logic that has not yet seen big impact in modern programming are the linear type systems. Allowing variables to be instantiated from linear types is an interesting feature enabling unique control of variables.

The concept of linear types is a fairly new one and a direct result from rigorous research in the field of linear logic and type theory; it has not yet been made available to the programming community as a whole. This can be an obstacle for uninitiated programmers having no academic background or knowledge in lambda calculus, linear logic or type theory wanting to explore the concept of linear types, which is something the project aims to change. The language developed to fulfil this has been named Lollipop.

1.1 Purpose

The purpose of the project is to define and implement a general-purpose functional programming language with linear types. This does not exclude the use of non-linear types within the type system, as this would pose obstacles in doing some of the most basic things in programming. Furthermore, the formal system of lambda calculus of functional programming suits the integration of linear types well thanks to linear types having its roots in linear logic [1]; linear logic can be expressed as a resource-aware lambda calculus [2]. Therefore the functional paradigm, which is based on lambda calculus, is a suitable choice for the language.

Lollipop should work as a simple platform providing programmers with an intuitive functional programming language, enabling practical hands-on experience with the concepts of linear types thus making it easier for programmers with non-academic backgrounds to understand and learn how they behave and can be beneficial in every-day programming. The result should be a simple proof-of-concept of how linear types can be integrated into a general-purpose functional programming language. This will aid both future development and in making linear types more available for programmers, taking the concept from the academic environment to the programming community.

2 Technical background

In classical engineering, mathematical and computational models are often used in order to find an optimal solution to some specific problem of interest. What tools and techniques to use depend wholly on the field in which the discipline of engineering is applied. In software engineering specifically, programming languages are used in order to solve complex problems. These programming languages can be described by relatively few concepts and can be seen as a "syntactic realization of one or more computational models" [3] where these computational models have a rather obvious relationship to their syntax through the semantics of the language. In this chapter a brief introduction to what a programming language actually is and what intricate parts it consists of will be given, as well as an overview of the problem as a whole.

2.1 Programming languages

A programming language can be seen as the tool used by programmers to write programs, where a program can be seen as a set of human-readable instructions[4]. All programs furthermore contains a number of declarations — common ones being functions, constants and variables. Programs are parsed and compiled into machine code which the computer finally executes.

There are many programming languages used today and their purposes and paradigms vary — there is truly no programming language for all purposes. Some are used for creating high-level applications used on mobile platforms while others may be more suited for writing code for low-level applications, enabling closer control over machine hardware. These paradigms can furthermore be considered as the realization of a computational model, where the most common models today are imperative, functional and logic [3].

A term that is often used for describing some programming languages is “general purpose”. This description aims to capture the purpose of the language to be able to create a wide variety of programs. Essentially these languages are designed not to be limited to any specific application domain, rather they are applicable in most domains.

2.2 Types and type systems

The type system is a fundamental set of rules on how different values are used within a language. A type is a primitive type, a compound type or a function type. Some of the most common primitive types represent integers, floating point numbers, boolean values and characters. Combining these primitive types creates compound types, e.g., a type String could be a list of type Char. Values in a statically typed language always have a fixed type, whereas in a dynamically typed language the type of a variable is not decided until runtime [3].

Table 2.1: Type declaration written in Haskell of two different functions: `x` and `largerThan3`.

```
x :: Integer          largerThan3 :: Integer → Bool
x = 42                largerThan3 i = i>3
```

Table 2.1 illustrates two function declarations followed by their function bodies in Haskell [5]. To the left a function that always returns an integer (namely 42). Here, `x` is the function name and `::` denotes the type declaration to the Integer-type. To the right is a function that takes an integer and returns a boolean. Following is the functions types where the first type, Integer, is the argument of the function. The second type, following the type constructor "→", is the return-type of the function which in this case is Boolean.

A common approach is to pass values into functions as arguments that are later used in whatever fashion the programmer chooses. This way, there are no constraints on how many times these variables can be used, it might

be used a lot or not at all. This is an intuitive and practical way of dealing with parameters; however, this method can lead to a number of issues. Firstly, there is no guarantee that the result of a function ever will depend on the input-arguments. Secondly, there is no control over duplication or destruction of references to variables, which can lead to tricky situations regarding synchronization and sane representation of finite resources [6].

Most typed programming languages today use types in the way previously described, making it the norm in modern type systems. In contrast to this way of handling types and variables, the concept of linear types has spawned. The idea comes from the research area of linear logic and its key concept is the *consumption* of variables [7]. A variable of a linear type needs to be consumed in the same context as it was introduced — this essentially sets a requirement on the programmer to include the linear variable exactly once in possible expression results, which threads it through the program. Linear types can therefore model finite resources in a more logical way where they cannot be duplicated nor destroyed, but only transformed and passed along.

This concept of consumption eliminates the need for garbage collection of such variables, since there is always exactly one reference to it at any given time, which means that the memory space used by it gets free directly after usage. This also enables safe use of destructive updates on mutable data structures since there is a guarantee that only one reference to it exists. This furthermore provides safer I/O streams, as these can be modelled as linear, hence allowing only one active reader/writer on the stream simultaneously [6]. Also, providing the rule of using each linear variable exactly once ensures that functions will depend on the given parameters, thus eliminating possible bugs in the code caused by a programmer accidentally using them the wrong number of times.

The theory of linear types is well researched and several thorough papers on the subject have been published. The concepts treated in these publications are rather complex and demands previous knowledge in lambda calculus, logic and type theory. There are, at the time of writing, not any mainstream general-purpose programming languages using linear types. The language Clean [8] is one of the few languages with an implementation, but there are a handful of other programming languages, such as Idris [9] and Rust [10], starting to introduce related functionality inspired by the concept. This scarce ecosystem poses a large obstacle for programmers wanting to familiarize themselves with linear types — there are simply not many programming languages formalizing it, implying a need for more general-purpose programming languages making use of them within their type system.

2.3 Analysis of the problem

Defining and implementing a complete general purpose programming language is very time-consuming, as there are a large amount of details needed to be taken into consideration. The purpose and application of the language needs to be decided upon, as well as the core functionality in form of functional and non-functional requirements. There are moreover a number of other issues that need to be addressed, such as if an interpreter or compiler should be written, what unique functionality to include in the language, how the type system should be designed and what the syntax should look like. A programming language can generally be divided into two parts - its front end, covering the syntax of the language, and its back end, consisting of the internal structure and heavy logic.

2.3.1 Syntax and conventions of programming languages

For a programming language to be attractive to programmers, there has to be a lot of consideration behind how the syntax of the language is constructed. Key aspects such as what symbols to use in different cases, how indentation should be handled, what naming rules to use and conventions need to be decided upon. These are only a fraction of all the questions that need to be addressed before writing the concrete syntax of a language. Another thing to keep in mind is the fact that programmers are more comfortable using a new language if it behaves in a similar way to one they already know. That is why both syntax and conventions of a new language would benefit greatly from being similar to an already well-known language.

2.3.2 Internal language functionality

An important part of any programming language is how it is internally structured and what functionality it offers to the programmer. A well written back end gives the language a rigid base and increases the possibility for further development. Moreover, extensive support for basic operations, including both mathematical and logical, needs to be added for both pre-defined and user-defined data types. A number of these operations are suitable to be defined as infix-operators evaluated directly in the interpreter while others, most often prefix operators and functions, easily can be defined in a standard library for usage in all programs. Some languages, such as Haskell, allow the user to define their own infix operators.

An internal feature that is of paramount importance for any programming language is the evaluation strategy used by the interpreter. This determines when and how expressions and arguments are evaluated and is a crucial decision to be made early in the development process. There are a number of different evaluation strategies used in modern programming languages, where some well known strategies are *call-by-value*, *call-by-name* and *call-by-need*.

Call-by-value is a strict evaluation strategy which always evaluates expressions before using them and is utilized in many programming languages today, such as C and C++ [11]. Having an expression constructing a list and returning its head would in a language using call-by-value evaluate the whole list before returning the head element. This can in some cases be extremely inefficient, especially when dealing with large data structures.

Call-by-name is a non-strict strategy evaluating expressions as they appear in functions. Expressions passed into functions are not evaluated straight away, but rather substituted directly into the function body instead [12]. An example of when this is used is when calling a higher-order function with another function as its only argument. The argument function will not be evaluated by the interpreter until it appears within the higher-order function. Call-by-name does also allow evaluation of expressions making use of non-terminating computation, such as computing a finite subset of an infinite data structure.

Call-by-need is another non-strict evaluation strategy similar to call-by-name, with the only difference of it memorizing evaluated expressions. If an expression has been evaluated at some point, its value is stored for later use, which naturally makes it faster than call-by-name. Call-by-need is also known as lazy evaluation and is popular in pure side-effect-free functional programming [13].

2.3.3 Testing

In order to pinpoint potential design issues in software, user tests, or usability tests are often conducted[14]. Usability tests for a programming language could consist of small samples of typical programming problems, which the testers would have to solve using the language under development.

Another aspect of testing is that of the internal functionality of software. In order to ensure quality, testing is an efficient tool[15]. Something that can be used for this type of testing is a so called test suite, consisting of a number of test cases[16]. A test suite could in this manner be used to efficiently test expected behaviour of software.

2.4 The impact of programming language development in society

The development of new programming languages and the further development of existing ones constantly provide job opportunities for programmers. Furthermore, the introduction of a new, user friendly and well functioning programming language can greatly increase the coding experience and thereby working environment for programmers. Programming languages with an intuitive syntax and well functioning debugging tools can increase the efficiency of software development, and thus the economical revenue of companies. However, when programming languages get better at bug handling, people working as debuggers might lose their job, which can be seen as a limitation in society. Resource aware software designed using linear types can help limit resource use and do more efficient computations, which can aid sustainability.

3 Scope

Implementing a full-fledged general purpose programming language is a considerable undertaking, especially in the time frame of a bachelor thesis project running over 5 months. Adding to that, the implementation of linear types demands a lot of research as none of the group members have previous knowledge in the area. Consequently the writing of an interpreter instead of a compiler was decided upon early in the project, as this allows more valuable time to be allocated to crucial components of the project. This is because it is simpler to interpret expressions in a host language than to compile the language to low-level code. Modules such as the type checker, grammar, and interpreter will as a consequence of this this get a bigger focus.

The scope was furthermore narrowed down to employing the paradigm of functional programming due to its clean mathematical definition and foundation in lambda calculus. This is practical due to the fact that the design of the type system is based on the same ideas. Lollipop contains a subset of the functionality available in Haskell. Some of the main features found in Haskell that are excluded from the scope of this project are records and the myriad of predefined functions found in the Haskell prelude. Throughout the whole course of the project, reassessments were also made on what to include as well as exclude, due to time constraints; this is elaborated upon in section 8.4.

4 Requirements

The functional requirements stated in this chapter describe what Lollipop should do, while the non-functional requirements describe in what way it should do it. Hence, the non-functional requirements serve as rules of the language's behaviour. The Language goal briefly describes the overall goals of the language, which naturally overlaps with the projects purpose in many ways.

4.1 Language goal

As explained in the purpose, the goal of the project is to create a lightweight proof-of-concept of a functional language with support for linear types, while still being generally applicable to a broad problem domain. The use of linear types should enable the programmer to represent finite resources in a safe, more reasonable way where the type system prevents improper usage. The type system should also help prevent race conditions on shared resources, since if the file system is represented as linear there can only be exactly one reference to it at any time. Lollipop should be an easy way to get acquainted with linear types to make the concept accessible to the broad developer community.

4.2 Functional and non-functional requirements

In order to clearly understand what the programming language should accomplish, or rather what a programmer using the programming language should be able to accomplish, functional requirements have been set. These requirements offer a definite and thorough description of the most essential functions of Lollipop.

The functional requirements were decided upon early on in the process during the planning phase and have continuously been revisited and refined throughout the project. The functional requirements were split into separate groups with respect to the language, interpreter and type checker.

The language should

1. have a rigorous type system able to detect common type errors as well as linear type errors.
2. be able to handle recursion.
3. support lazy evaluation.
4. support pattern matching and guards.
5. have basic support for linear types.
6. handle basic user I/O.
7. support anonymous functions.

The interpreter should

1. have a Read-Evaluate-Print-Loop (REPL).
2. be able to evaluate well-typed programs.
3. be able to detect common type errors and provide basic feedback.

The type checker should

1. accept well typed programs.
2. be able to detect common type errors and provide basic feedback.

In order to elaborate on exactly how some of the functional requirements are to be fulfilled, the following list of non-functional requirements has been decided upon as well.

Non-functional requirements

1. The language should utilize linear types in an intuitive way inside type declarations used in functions
2. The grammar should be built using BNFC.
3. Haskell should be used to convert surface syntax to an internal representation
4. The type checker should be implemented using Haskell.
5. The interpreter should be implemented using Haskell.

5 Method

Defining and implementing a complete functional programming language is a long process. It requires rigorous planning and deep knowledge in many different areas from type theory to compiler construction. Therefore, the project was divided into two phases — a planning phase and an implementation phase. This division was done in order to ensure that the project was carried out in a well structured and organized manner, where goals and milestones were met in time. This was mainly done by deciding upon and setting important deadlines for the different features of the programming language and agreeing on base functionality.

The project members had little or no previous knowledge of how writing of a programming language was done, hence the first month of the project focused on gathering relevant information and reading up on core theory. Some of the studied areas included lambda calculus, context-free grammar, type systems, type checking and interpreters. In addition, linear logic and types were studied more closely to implement the unique twist of the language. The main sources of information were academic papers and books on the subjects.

Programming languages normally take years to develop, while this project only stretched over a few months. For this reason, reports from similar projects were also studied, to get a better insight of what to focus on and get some clues on how to structure the work process.

5.1 Planning phase

The first weeks of the project mainly consisted of planning out the rest of the project. Usually, three meetings a week were held to work with the planning in order to set up deadlines and common goals together with the language specifications. During these meetings the whole group was gathered as the specifications were a central part that would shape the rest of the project.

Early on the proposed twist, setting Lollipop apart from other languages, was decided, as well as a suitable programming paradigm and a general language style. Furthermore, an extensive list of all the features to include in the language was written. The features found on this list were classified as either high, medium or low priority. The list did also cover what primitive types to include in the type system of the language, as well as most shorthands and pre-defined functions in the Lollipop standard library, *Sugar*. When sufficient information on type systems had been gathered and the integration in the language had been discussed, the type rules were decided upon.

In order to visualise the progress towards the goal of the language, reasonable use cases were planned out and put together. The purpose of these were to keep the project on track and enable the group to focus on prioritized issues striving towards the common goals. The use cases were essentially a combination of concrete targets and problem descriptions, elaborated from what the group found to be common problems in programming languages today. Moreover, functional and non-functional requirements were developed in order to describe how the programming language should behave and to specify what a programmer using the language should be able to do.

5.2 Implementation phase

The implementation phase was the main part of the project, thus requiring the largest amount of time and resources. Resources were distributed for the first stage of the implementation. Two group members were allocated to the definition of the grammar and the remaining three laid the foundation for the internal AST and the interpreter. Later on all members contributed to all parts of the project.

As a visual aid for the following subsections, figure 5.2 shows how the presented parts of Lollipop fit together. It is a rough model of the work flow functions in Lollipop.

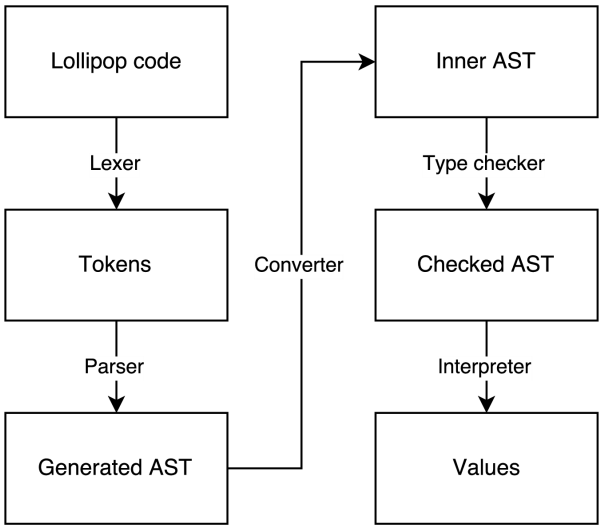

Figure 5.1: A rough model of the language that visualizes how the different parts of Lollipop fit together.

5.2.1 Syntax and grammar

For the implementation of the grammar and thus the creation of the concrete syntax of the language, the compiler construction tool BNFC [17] was utilized. Given a grammar, BNFC generates the compiler front-end for the language; that is, a parser, lexer and a complex yet useful abstract syntax tree. This was later converted into a less complex, minimal and optimized AST by the converter. The grammar shown in Table 5.1 is an example of BNFC code that generates the data structures for code literals.

Table 5.1: Example of how the Integer, Double, Char and String literals are defined in the grammar of Lollipop. Code written in BNFC.

```
LitInt.     Literal  ::= Integer ;
LitDouble.  Literal  ::= Double ;
LitChar.    Literal  ::= Char ;
LitString.  Literal  ::= String ;
```

As the interpreter and AST were written in Haskell, this was also used together with BNFC to generate the parser, lexer and the abstract syntax. BNFC additionally has support for Java and C/C++, but using one of these languages would introduce extra steps in the conversion of the abstract syntax, which seemed unnecessary as opposed to using Haskell. For every addition or change in the grammar rules, a test program, consisting of expressions with the intended syntax, were run to confirm that the grammar represented the language correctly.

5.2.2 The interpreter and converter

Alongside the grammar in BNFC, the interpreter was one of the first components to be implemented. It uses the internal AST, which represents the language without any syntactic sugar — that is, without unnecessary

identifiers and redundant ways of representing equal expressions. The interpreter receives a representation of an expression to be interpreted, evaluates it and returns the resulting value.

The interpreter was written in Haskell and built in an iterative manner, in parallel with the internal AST and environment for the language. For each new feature to be added to the language, three steps were executed in the back end of the language: firstly, a suitable representation of it in the inner AST was decided on; secondly, a function to evaluate it was implemented in the interpreter; and lastly, tests were carried out ensuring correctness in the implementation and expected behaviour.

In order for the interpreter to be able to read and interpret the parsed and lexed code efficiently, a small converter program was written. The sole purpose of the converter was to translate the surface AST, produced by the BNFC-generated parser, into abstract internal syntax compatible with the interpreter. As mentioned earlier, this enables a better representation of a minimal yet effective interpreter as there are a number of features in the syntax that can be modelled in the same way internally; an example being guards, that are easily translated into equivalent case-expressions.

Because of the functional paradigm, the decision to include lazy evaluation strategy, also known as call-by-need evaluation, in the interpreter made sense. Many functional programming languages, such as Haskell, use the same strategy [12]. Since the Lollipop interpreter and AST were written in Haskell, the call-by-need evaluation strategy could therefore be inherited. It is automatically applied to every evaluation of expressions written in the language.

5.2.3 Type system & inference

The type system written for Lollipop was based on the famous Hindley-Milner type system, with adaptations for the specific syntax of the language. There are several open source projects that have implemented the type inference algorithm W from the original paper [18] for the Hindley-Milner type system, these have been valuable resources for the minimal implementation which the functionality then was extended upon. This was expanded for linear types as well, which had to be researched since it is a new area. A major source of inspiration for this was the work of Wadler [19] in *Linear types can change the world*, but applied in a larger and practical context which posed new challenges.

When most of the grammar and interpreter code had been written, the development of the type checker began. A type checker generally ensures that programs written in a language do not contain any type errors when they are executed — it ensures that a program is type correct at runtime. At success the type checker passes the program to the interpreter and at fail it returns an error message to the programmer. Furthermore, type checking of linear variables was done in a different way since the type systems differ, but the support of type rules from [19] helped greatly. However, since linear types have completely new limitations in the way they have to be used, non-standard checking techniques for this had to be implemented in the type checker.

5.3 Testing

As for software products in general, there are certain demands from the users which have to be satisfied, together with functional requirements set by the person ordering the product. In the case of a programming language, the demands by the user come in the form of readability and writeability. This was ensured in the development of the language by conducting cognitive walkthroughs with several test users. The functional requirements were in this project set by the development team.

5.3.1 Cognitive walkthroughs

As briefly mentioned, cognitive walkthroughs were continuously carried out throughout the later parts of the project, with testers having a background in programming. The walkthroughs consisted of a theoretical part as

well as a practical part. The objective of the theoretical part was to evaluate the readability of the language by letting the users answer a set of questions, while the objective of the practical part was to evaluate the writability of the language by letting the testers write code in Lollipop.

The first set of questions in the theoretical part gave an indication of the experience level of the tester, while the remaining gave an indication of how intuitive and easy a sample of Lollipop code was to follow. The practical part consisted of a number of assignments which the testers would have to complete, using only Lollipop code and a limited set of pre-defined functions. Two assignments included in the practical part were writing of a function summing a list of integers and writing of a recursive function computing the nth Fibonacci number.

After completing both the theoretical and practical parts, the testers filled out a form of their initial impressions of the language, what they found good and bad in the syntax and their thoughts on the readability and writability. Notes were taken from these sessions and the issues found by the testers were later fixed in the implementation of the language.

5.3.2 Test suite

Debugging and testing were mainly done using a small test suite, which was expanded continuously throughout the project. The test suite consisted of a collection of standard programs and functions intended to run in pure Lollipop code. The purpose of this was to test the programming language and evaluate the functional requirements, as well as to showcase expected behaviour. Thorough tests on all parts of the language, from type-checking to correct evaluation, were performed.

For the purpose of correctness, each test in the test suite consisted of a Lollipop program intended to pass or fail. Some of the tests were written using an incorrect syntax and should therefore be reported as incorrect by the parser. Other tests were written in a correct syntax but containing type errors, which were to be reported by the type checker. There was also a group of tests representing programs that were both syntactically correct and well typed, but containing errors caught by the interpreter. This could for example arise when using a variable, not yet introduced in the environment as seen in Table 5.2. Given that there is no global definition of `x`, the variable can not be found in the environment, thus causing an error in the interpreter. Finally, there were a number of test cases representing "good" programs, expected to pass parsing, type checking and interpreting, intended to return an expected value when finally being evaluated.

Table 5.2: Example of a function trying to add the not-yet-introduced variable `x` to 42.

```
function bar :  Int
bar := x + 42
```

5.4 Tools

A number of third party tools for version control, communication, report writing and storage handing were used during the project. The use of such tools eased the overall workload and ensured a more streamlined, efficient work flow. These were all decided upon at an early stage in the project during the planning phase, in order to establish an agreed upon way-of-work.

Git — version control

In a larger project with several contributors and more complex components, there is a need for better control of the different versions of the source code, enabling backtracking in the project. In order to mange the code and enable an agile methodology, Git [20] was used during the development of the language. Git can be described as a decentralized version control system allowing collaboration between several developers on a single code project. There are a plethora of different repository hosting services for Git, and the one used for this project was Github [21].

Trello — task board

As the development was done in an agile manner, a need arose for a shared board for managing all different tasks of the project. Instead of using the most common approach with a white board along with sticky notes, a more modern tool accessible at any time over the internet was preferred. As most team members already had used Trello [22] before, it was a straight forward choice. Trello is best described as a web-based application used as a tool for project management, enabling users to create various boards and notes to attach to them. For example, a board for tasks to be done in the converter and one for the interpreter were used. In this way, all the different tasks could easily be overviewed and the work load became more transparent and manageable.

Mendeley — reference manager

When writing a technical report on an advanced subject, much research has to be done. Naturally, when conducting extensive research, there quickly arises a need for a good reference manager. Mendeley [23] was used in order to keep track of and manage all these references when writing the report. Mendeley can be described as a software for managing references and collaborating on bibliographies. It enables sharing of articles and publications in groups. It can moreover automatically create reference lists in the form of BibTeX for easy use in LaTeX.

Overleaf — writing collaboratively

In order to efficiently collaborate on the report, Overleaf [24] was used. Overleaf is a cloud tool for real-time collaboration on LaTeX projects. It enables simultaneous access to the LaTeX project for several people at once. To efficiently keep the documentation up to date, this has been immensely useful.

Google Drive — file sharing

For managing meeting agendas, notes and similar pieces of documentation, Google Drive [25] was used. This is an easy and intuitive way to get shared file directories in the cloud, to make sure that all project members have the most recent version of all files.

Slack — communication

The main tool for long distance communication was Slack [26]. It is available via the web for computers and via an app for smart phones. This was very useful when questions arose while working alone or in smaller groups with the project, as other members quickly could be consulted.

6 Implementation of the language Lollipop

Lollipop was developed with the intention to be a general-purpose programming language acting as a proof-of-concept of how linear types can be implemented in such a language. It shares many similarities with the two well established languages Haskell and Miranda. Two of the distinct features that are shared between the three are their functional paradigm and their lazy evaluation strategy. Lazy evaluation, also known as call-by-need, is used in order to avoid repeated evaluation of expressions, significantly increasing performance and enabling use of infinite data structures.

6.1 Grammar

The grammar of Lollipop is written in Backus-Naur Form, or BNF, which is one of the primary notation styles for context-free grammars that is used to define the syntax of programming languages [27]. BNFC (BNF Converter) uses the grammar to generate the lexer and parser for the language which will convert written code in text form into the abstract syntax. Some concrete syntax examples, and corresponding grammar rules, can be seen in Table 6.1. The BNF grammar consist of a set of derivation rules composed of, from left to right, the *rule label*, *value category*, *production arrow* followed by a sequence of productions that consist of *terminals* (string literals) and *non-terminals* (identifiers). The digit after each value category and non-terminal denote the precedence level [28].

Table 6.1: Concrete syntax (to the left) and BNF grammar (to the right) of four simple expressions in Lollipop.

`3 * 4`	`EMul. Exp7 ::= Exp7 "*" Exp8 ;`
`3 / 4`	`EDiv. Exp7 ::= Exp7 "/" Exp8 ;`
`3 + 4`	`EAdd. Exp6 ::= Exp6 "+" Exp7 ;`
`3 - 4`	`ESub. Exp6 ::= Exp6 "-" Exp7 ;`

A quality that is desirable in writing a programming language is unambiguity within the context-free grammar, meaning that for every sentence that belongs to the language generated by the grammar there is at most one parse tree for that sentence [29]. Unambiguity is desirable due to the fact that when the AST is linearized into a string from an ambiguous grammar, it is impossible to uniquely determine which AST represented that string, so the semantics, which derive from the AST, are lost [30]. A famous example of ambiguity is *the dangling else problem*, which arises with nested if statements. This is solved by the parser convention of always applying a shift when a shift/reduce problem arises [31]. As Table 6.2 illustrates, depending on the derivation order of the expressions, there are two possible parse results. In the first expression the parentheses do not matter, but in the other three it may alter the evaluation of the expressions significantly. This is were precedence levels enter the picture. "The precedence level regulates the order of parsing, including associativity" [28, p.3] — that is, a higher precedence level binds more tightly than a lower precedence level. Thus, as Table 6.1 shows, multiplication and division bind stronger than addition and subtraction and will be evaluated before them.

As mentioned in the quotation, precedence levels also adjust the parsing order for expressions involving operators with the same precedence according to the associativity of the operators. Two example expressions of this can be seen in the second and fourth row in Table 6.2 where both of them would be interpreted as the leftmost result because both of the operators are left associative in the language. The grammar of the language has been defined adhering to the above mentioned principle of precedence levels in order to ascertain its unambiguity.

In order to know when the different constructs of a Lollipop program, such as function definitions and data type declarations, are terminated, semicolons are used at the end. Inserting semicolons after every line, however, can be arduous and it does not add meaning to the evaluation of the program except for line termination. Hence, layout syntax, that inserts a semicolon at the end of every line of program code, was used. Layout syntax works like a

Table 6.2: Example of two different parse results (to the right) of four expressions (to the left).

`a + b + c`	`(a + b) + c`	`a + (b + c)`
`a - b - c`	`(a - b) - c`	`a - (b - c)`
`a - b * c`	`(a - b) * c`	`a - (b * c)`
`a - b + c`	`(a - b) + c`	`a - (b + c)`

preprocessor and "is a means of using indentation to group program elements" [28, p.8]. Several layout syntax pragmas can be used in the BNFC file to specify how the layout syntax works for the specific grammar. The ones used in Lollipop are `layout toplevel` and `layout` "`of`", where the former aids with the semicolons as terminators for every line of code and "`of`" is a keyword to start a layout list which inserts brackets and semicolons in `case of` expressions. The layout resolver, that handles layout syntax, is run between the lexer and the parser.

6.2 Types

The core of the type system of the language was implemented with basic support for primitive data types, but was also made expandable with support for composite data types. Such types are composed by primitive types and various other composite data types in order to build expressions. Construction of expressions are done easily due to the small number of primitive types available. These primitive types are the most basic building blocks of the programming language and consist of integers (int), floating point numbers (double) and characters (char). While the boolean type is most often implemented as a primitive type in programming languages as well, Lollipop supports it using constructors, resulting in a more elegant representation in the inner structure, yet no difference to the end user. Another exception is the String type, which is defined in the same way as it is in Haskell, as a list of characters.

6.2.1 Declaring data types

An important feature in a general programming language is user defined data types. Like Haskell, Lollipop supports algebraic data types for this purpose. This is a must have in any programming language being referred to as being general purpose. Declaring new data types is a powerful tool of bringing good structure and readability to programs, as new data types can eliminate the need to identify objects with primitive data types. An example of this could be the definitions of a colour. A colour could be defined as a tuple of integers, or as a predefined colour in a data type, the latter being much more sensible for the user and easier to understand. In Lollipop, a data type can be defined using one or more value constructors. The different value constructors are separated using pipe-characters (|). An illustration of the syntax of data types in the language can be seen in Table 6.3.

Table 6.3: Example of two predefined data types in Lollipop's standard library called sugar. The `Boolean` type consist of two value constructors, `True` and `False`. The `Cake` type constructor, with its type parameter `a`, can either be a `Lie` or a `Sweet a`. `Cake` is similar to Haskell's `Maybe` type constructor.

```
datatype Boolean := True | False
datatype Cake a  := Sweet a | Lie
```

6.2.2 Type inferencing

A type checker was written in order to ensure the type correctness of programs written in Lollipop. This was done using type inference with a modified version of Algorithm W [32], that was first proposed by Hindley-Milner, as its core [18]. In order to work properly with the rest of the compiler-back end, the type inferencer was also written in Haskell. As briefly mentioned in the previous chapter, the implementation of the type inferencer and the type system were done at the same time iteratively, to ensure correctness of both parts. One of the reasons for this being the fact that the type inferencer uses type rules stated in the type system in order to check whether or not expressions written in the language are valid.

The type checker is an *error-reporting checker*, returning an "OK" given that the checked program is type correct, and an error message if it is not. Another type of type checker is the *rude checker*, which only returns boolean values of TRUE or FALSE, stating a successful or unsuccessful type check. A third alternative is the *annotating checker*, which returns the AST with which it was called, but with type information for each expression and sub-expression[33]. The annotating checker would be the most suitable for this project, see chapter 8.4.3 for the reason of this and why it was not used.

In order to ensure that the deduction of the data types in the language is done in an automatic and correct way, type inference was introduced. This makes programming easier, as type declarations do not need to be done. A downside to not explicitly specifying the types though would be that it becomes harder for a reader of the code to see what types that are used in a function. The type inference goes through the expression and looks up the most general type of it. This could however fail, and if so, an error is returned.

6.2.3 Linear types and linear type checking

As described in Technical background, the concept of linear types introduces a very specific constraint on the type system. Variables of linear types must be used exactly once - they have to be consumed; however, there are no limitations of in what context or extent linear types can be used, as long as they follow the simple rule of consumption.

Linear types are declared by using an "i" in front of the type identifier. `iInt`, `iDouble` and `iChar` are examples of linear representations of the primitive types of Lollipop. In type declarations of functions these types are accompanied by a $\multimap$-symbol, denoting the consumption of the linear variable before it. This lets the type checker know that the parameters of these types have to follow the linear rules of consumption. If the requirements of this rule are not fulfilled, an error is prompted to the programmer.

Because of the specific type rule of linear types, the type checker has to perform rigorous tests of expressions containing linear variables. First, all local variables have to be extracted and mapped to their respective linear type, letting the type checker know they are to be checked. Second, the actual check has to be made, where the type checker records all uses of every variable in an expression. If a variable of a linear type is not used exactly once, the type checker reports this to the programmer.

6.3 Loli — the Lollipop interpreter

In order to run and execute programs written in the language they need to be interpreted into code which the computer can understand. This is done sequentially through a number of steps, by a program written in Haskell known as the Lollipop interpreter — Loli. Assuming the code is written in legal Lollipop syntax as described in the grammar, the interpreter parses the program and evaluates it using the variable-to-value environment. The interpreter furthermore makes use of several crucial components which all are described more in-depth in the rest of this section.

At first the minimal interpreter, the small AST and the handling of the variable-value environment were all placed in the same file. As the file became larger in size and the code more complex and comprehensive, it was refactored and the code for each part was broken out into a separate file. The features of the language were divided into three groups and the implementation of the interpreter had a deadline for each feature group, called milestone. The first group consisted of basic features, such as lambda calculus, constructors and the printing functionality. Milestone two and three then extended the language mainly with features that were not necessary for the language to work, but made it simpler to use.

6.3.1 Abstract syntax tree

The abstract syntax tree, or AST, is the underlying data structure of how a program is internally structured[13]. This mean that it offers an abstract representation of the declarations and expressions of which the program is constructed. The construction of an AST of a program in Lollipop is done in a minimal and concise manner, avoiding unnecessary syntactic sugar. An example of this is how an if-statement can be translated into a case-expression, as seen in Table 6.4, where the if-statement, to the left, is translated into an equivalent case-expression, to the right.

Table 6.4: Translation of an if-statement into a case-expression.

```
if x == 2                    case x == 2 of
then Print "x is 2"            True → Print "x is 2"
else Print "x is not 2"        False → Print "x is not 2"
```

The AST for the language was written in pure functional Haskell code, but could also have been written in another, tentatively lower-level, language, such as C for better control over the memory space. The main reason why Haskell was chosen was that parts of its functionality enables flexible language development, but also due to the many similarities between Haskell and the language which Lollipop was intended to be. There were furthermore other advantages of using Haskell when representing the AST for the language, such as the inheritance of the lazy evaluation strategy and the Glasgow Haskell Compiler, GHC, which is has rigorous concurrency and parallelism support among its features.

6.3.2 The conversion from generated AST to inner AST

As mentioned in the Method chapter, a converter for the abstract syntax was needed for the interpreter to be able to interpret the written programs. An alternative approach to this would be to skip the conversion step altogether and write the interpreter of the language for the BNFC-generated abstract syntax straight away. Although this seemed like a reasonable approach at first, this would prove to be a major disadvantage over constructing a new, minimal abstract syntax. The main reason being that this would enable better optimization and a more compact core data structure, which would be much easier to handle during the evaluation of expressions than the generated one. As for most programming languages, there are a number of ways of writing identical expressions using different syntax. This is often referred to as syntactic sugar. For example, expressions making use of pattern-matching found in functional languages can easily be translated into case-expressions, hence eliminating the need for a data type modelling pattern-matching in the abstract syntax. This is not something that a generated syntax takes into account, making it less efficient and thus containing unnecessary syntactic sugar.

The converter module mainly consists of functions taking the grammar representation of data structures and returning the corresponding data structures using the inner AST. Structures containing sub-components call the converting function recursively for those components and builds their result with the converted version of them. An example of that is seen in Table 6.5, where the expression for the logical operator `or` is converted by the function `cExp`, which converts expressions.

Table 6.5: Example of how an expression is recursively converted.

```
cExp (A.EOr e1 e2) = D.EBinOp D.Or (cExp e1) (cExp e2)
```

As the inner AST is much more compact than the AST generated from BNFC, some structures do not have any directly corresponding representation in the inner AST, and are therefore converted — desugared — into other structures. An example of how if-then-else expressions are desugared into case expressions can be seen in Table 6.6.

Table 6.6: This is how an if-then-else expression in the outer AST representation is converted to a case expression in the inner AST.

```
cExp (A.EIf e1 e2 e3) = D.ECase (cExp e1) [   ((D.PConstr "True" []), (cExp e2)),
                                              ((D.PConstr "False" []), (cExp e3)) ]
```

6.3.3 The value environment

The interpreter uses a mapping from variables to values to store the values of a program. This mapping is called the environment. All variable names must be unique within a scope. If another value with the same variable is inserted to the environment, the old value stored with that variable will be removed. The inner AST of Lollipop contains four different kinds of values: literal values, used to represent literals; I/O values, used in functions handling I/O actions; function values, representing functions that take one value and return another; and constructor values, representing value constructors.

Before the evaluation of the program can start, all user defined top level functions and data type constructor declarations in it are converted to values and added to the environment, together with the declarations defined in sugar. This is done from Loli. Functions that take several arguments are curried so they can be represented by a function value. The constructors of the data declarations are added to the environment after having been converted to their corresponding representation as values. Local variables are added during evaluation.

The built-in constructors and functions of the language, called the start environment, are a constant part of the environment. These functions cannot be expressed in terms of other functions, and must therefore be evaluated explicitly in Haskell. Most of the functions represent unary or binary operators that are part of the Lollipop syntax, such as !, + and >>=, but the I/O functions printChar and readLine are also included, as they are implemented using Haskell I/O functions. The constructors for tuples, triples and lists have a special syntax in the language and are therefore also a part of the start environment

6.3.4 Evaluation from expression to value

As soon as the environment, containing all user defined and built-in top level function as well as data type declarations, is built the functions can be called. This is done from the function `repl` in Loli. To allow tests on the interpreter without using the front end, the interpreter module contains a function that builds the environment and then invokes the main function, which is a requirement for the back-end test programs to have. Test programs sent to this function must be in the inner AST representation.

In order to evaluate expressions, the interpreter module has a function called `eval`. Given an environment and an expression it returns the resulting value.

Expressions evaluted by eval as they are represented in the AST

- Literal expressions (**ELit** Lit) (Lit is a data type representing literals) are evaluated by encapsulating the literal Lit into a literal value (VLit Lit).

- Lambda expressions (**ELam** Var Exp) are evaluated as function values (VFun), which maps the variable Var to a given value in the environment. Exp is then evaluated given this new environment.
- Let-in expressions (**ELetIn** Var Exp Exp) are evaluated by evaluating the first expression and mapping its value to Var in the environment, thus updated it a recursive manner, allowing laziness in ELetIn expressions; and then evaluating the second Exp using the updated environment.
- Variable expressions (**EVar** Var) are evaluated by returning the value which maps to the variable Var in the environment. The value can either be a function (VFun), added before the evaluation of the program, or a liternal (VLit) added during the evaluation of an ELam, ELetIn or ECase expression.
- Application expressions (**EApp** Exp Exp) are evaluated by first evaluating the first expression, which must return a function value VFun, then evaluate the second expression and lastly apply the returned function to the value of the second expression. The evaluation is implemented with support for partial application.
- Unary operator application expressions (**EUnOp** Op Exp) are evaluated in a way similar to EApp expressions; however, as the operators are part of the start environment, they require special treatment, and can therefore not be modelled directly as EApp expressions. The operators are stored in the start environment as function values, which are looked up by the evaluator. The expression of the EUnOp is then evaluated to a value which is applied on the function.
- Binary operator application expression (**EBinOp** Op Exp Exp) are evaluated as EUnOp expressions, using partial application to first receive the function produced by applying the operator on the first expression, and then apply the second expression on that function.
- Constructor expressions (**EConstr** ConstrID) are evaluated to the value which is stored with the ConstrID in the environment. All such values were added to the environment before the evaluation of the program, either as one of the built-in constructors or as a user-defined.
- Case-of expressions (**ECase** Exp [(Pattern, Exp)]) are evaluated by first evaluating its first expression, which value is to be cased on. This evaluation order is strict, so case expressions cannot be defined in a recursive way like let-in expressions. It then finds the first case which pattern matches the value and adds the variable-value bindings introduced by the pattern to the environment, before evaluating the expression corresponding to the matching pattern.

6.4 Sugar — the standard module

Sugar is the pre-loaded standard library of Lollipop. It contains all essential functions one can come to expect being built into a language. Many of the functions found in sugar was inspired by those found in Haskell's equivalent, Prelude.[1] With this as a solid base, corresponding functions were implemented in Lollipop code and they are automatically imported as a package into every program when the interpreter is started. Over 40 functions were implemented, including everything from list operations and logic to arithmetics, somewhat extensive but a small amount compared to a production-ready library.

6.5 Test suite

The test suite was incrementally extended to represent the current features in the latest working version of the language. When new features were added to the code base, corresponding tests were also added to ensure correctness. The test suite acts as a control system to verify that the front- and back-end of the language are able to represent the wanted functionality.

[1] The Haskell standard library, Prelude: https://hackage.haskell.org/package/base-4.8.2.0/docs/Prelude.html

7 Result

The outcome of the project can be summarized as a successful implementation of a typed programming language, called Lollipop, supporting linear types. It can be categorized as a general-purpose programming language, as it aims to be broadly applicable in various application domains. Along with the actual language, an interpreter was written in Haskell, as well as a type checker investigating whether Lollipop code is correctly typed. The type inference algorithm is an extension of the famous Damas-Milner algorithm, to cover the full internal syntax together with the linear types. Moreover, a test suite was implemented as well as a standard library (`sugar.lp`) where all the default methods are defined, a natural albeit smaller equivalent of Haskell's Prelude.

An example of two functions written in the language can be seen in Table 7.1. `foldr` takes a function `(a → b)`, an accumulator `b` and a list `[a]`, returning a result where every element has been folded into the accumulator by the given function. The function `and` uses `foldr` in order to check if all elements in a list are defined as `True`. The functions start with their respective type declaration, followed by each function definition where arguments can be found on the left side of `:=` and the expression to be evaluated on the right side.

Table 7.1: Functions `foldr` and `and`, where `and` utilizes `foldr` to evaluate its expression.

```
function foldr :  (a → b) → b → [a] → b
foldr f b [ ] := b
foldr f b (x:xs) := f x (foldr f b xs)

function and :  [Boolean] → Boolean
and xs := foldr (\x y → x && y) True xs
```

7.1 Syntax

As the syntax was designed with readability in mind, as well as to easily be recognized by experienced programmers in functional programming languages, it shares many similarities with that of Haskell and Miranda. Table 7.2 is a comparison of how the function `takeWhile` is implemented in Lollipop, Haskell and Miranda. There are a number of similarities in how functions in the three languages are implemented, but there are also some significant differences between them. An example of where the three languages differ is their representation of guards, which handle conditional execution of expressions. Lollipop and Miranda begin their guard-declaration with the expression intended to be executed, followed by the predicate, but with different keywords. Lollipop uses *when*, while Miranda uses *if*. As for Haskell, the guards-notation begins with | (a pipe), followed by the different predicates and ends with the expression associated to each predicate.

Table 7.2: Comparison of Lollipop, Haskell and Miranda implementations of `takeWhile`, which returns elements from the front of a list while it satisfies the predicate function.

Lollipop
```
function takeWhile :  (a→Boolean)→[a]→[a]
takeWhile _ [ ] := [ ]
takeWhile f (x:xs)
     := (x:(takeWhile f xs)) when f x
     := [ ]
```

Haskell
```
takeWhile ::  (a→ Bool)→[a]→ [a]
takeWhile _ [ ] = [ ]
takeWhile f (x:xs)
     | f x = x:(takeWhile f xs)
     | otherwise = [ ]
```

Miranda
```
takeWhile ::  (*→bool)→[*]→[*]
takeWhile f [ ] = [ ]
takeWhile f (a:x)
     = a:takeWhile f x, if f a
     = [ ], otherwise
```

7.2 Type inference

Type inference was implemented in the language to allow anonymous functions. This together with type checking of entire functions ensures that the inferred type of the body of a function corresponds to the declared type in the function head. The type inferencer was based on Milner's algorithm W but is heavily modified and extended. Things that have yet to be implemented are type inference of recursive let expressions, as well as type inference of linear types (but type *checking* of them works).

7.3 Linear types

Support for linear types was added to the type system of the language. The implementation is minimal yet flexible, giving the programmer the possibility of denoting types as linear directly in the type declaration of functions. The implementation is moreover modular and easily extensible for future projects and serves as a light-weight demonstration of the concept, rather than a full fledged feature.

Table 7.3: A concrete example of a linearly typed variable x being illegally used. It may be used once, and only once, in the function body.

```
function foo :  iInt -o iInt
foo x := x + x
```

The type checker uses an algorithm in order to check whether variables of linear types fulfil the type rules, successfully raising errors on illegal usage, as in Table 7.3. This makes Lollipop a proof-of-concept of how linear types can be implemented in programming languages in the future. Though there are, as previously mentioned, a few programming languages using similar concepts within their type system, Lollipop is a step in making linear types available to the uninitiated programmer.

Another, more extensive, example of the usefulness of linear types can be seen in Table 7.4. The small program models a coin and a list of coins, and the functions below use linear versions of them. A coin is modelled to always have an owner and the function `giveTo` replaces the current owner of a coin with a new one. The function transfer updates the owner of all the coins in a list. The implementation marked with (1) is a correct implementation of the non-base case of such a function. The implementation marked with (2), however, both adds the original coin and the coin with the updated owner to the returning list. In that way, it gives the coin two owners, and hence duplicates to coin. This is not a desirable property of coins. With linear types, this can be prohibited — the second implementation will trigger a type error as the linear variable `c` is used twice. Without linear types, such an implementation cannot be prohibited.

Table 7.4: Example of how linear types can model coins and lists of coins. The data types are not defined as linear, but used as linear in the functions below. The function `giveTo` is correctly implemented, as well as the base case and the first implementation (1) of the non-base case of the function transfer. The second implementation (2) of the transfer function would however trigger a type error in Lollipop, as is uses the linear variable `c` twice in its body.

```
datatype Coin := C Owner
datatype List := Cons Coin List | Nil

function giveTo :  iCoin -o Owner -> iCoin
giveTo (C ow1) ow2 := C ow2

function transfer :  iList -o Owner -> iList
transfer Nil _ := Nil
transfer (Cons c cs) ow := Cons (giveTo c ow) (transfer cs ow)          (1)
transfer (Cons c cs) ow := Cons c (Cons (giveTo c ow) (transfer cs ow))  (2)
```

The initially planned application of linear types in threading of I/O operations is not part of the final result, as it was more complicated than first expected to implement. Instead, a simpler solution of explicit sequencing was chosen.

7.4 Loli — the REPL

Loli is the successful implementation of a read, evaluate, print-loop for the language. It can be seen as an interactive evaluation environment for Lollipop code. This is where the programmer loads modules and runs the actual code, as the language does not come with a compiler that generates executables. Loli catches different kinds of exceptions from the rest of the tool chain and tries to handle them. This means that most of the time it gracefully manages syntax errors, erroneous typing, incorrect module formats and other common user faults. The code is then converted to the internal AST, where the interpreter evaluates all expressions and returns the computed result. It works efficiently utilizing a lazy evaluation strategy, conveniently inherited from the implementation in Haskell.

Loli is loaded in a terminal environment and all user interaction is done by a set of pre-defined commands. The most basic commands include *:l*, *:r*, *:t*, and *:q*. The user loads a module using *:l*, reloads a module using *:r*, displays information about type declaration using *:t* and exits Loli by using *:q*. Examples of this can be seen to the left in Table 7.5. To the right there are a number of function calls being made, returning an evaluated value. As seen, every command is followed by a prompt, such as "Successfully loaded sugar" or the result of an evaluation.

Loli can be compared to the GHCi-environment available for Haskell, using similar types of commands. When entering a command, both interpreters require a ":" followed by a command name and possible input parameters.

Table 7.5: Example of how commands in Loli can be used. To the left `:l` loads a module, `:r` reloads a module, `:t` shows information about a the type declarations of a function and `:q` exits Loli. To the right a number of function calls are made.

```
>:l sugar
Successfully loaded sugar
sugar>:r
Successfully loaded sugar
sugar>:t foldr
foldr :  (a → b) → b → [ ] a → b
sugar>:q
```

```
sugar>isOdd 97
True
sugar>9+2*3/2
12.0
sugar>print "hello" » print "world"
hello
world
```

The commands shown above are also supported in GHCi, as well as a number of other commands. The rather limited number of commands found in Loli were chosen due to their relevance to the project with respect to the minimal implementation of the language and the scope of the project.

Table 7.6 is a comparison of the commands *:l* and *:t* in GHCi (to the left) and in Loli (to the right). As seen, GHCi is slightly more informative than Loli in its printouts, otherwise the two interpreters behave in similar ways.

Table 7.6: Comparison of basic commands in GHCi (to the left) and Loli (to the right).

```
Prelude> :l main.hs
(1 of 1) Compiling Main    ( main.h, interpreted )
ok, modules loaded:  Main.
Main> :t foo
foo ::  Double → Int
```

```
>:l main
Successfully loaded main

main>:t foo
foo :  Double → Int
```

7.5 Results from cognitive testing

A number of important notes were taken from the feedback of the test users. As previously mentioned, the user feedback was gathered while performing cognitive walkthroughs. From the nature of how the cognitive tests were carried out, it was mainly the readability and writability of the syntax which could be tested during the walkthroughs.

An interesting insight from the tests was that the vast majority of the testers found the *when*-keyword in the guard-notation very intuitive and even preferred it over the pipe-syntax (|) used in Haskell. This was not so much constructive feedback, as it was a reassurance of the syntactic choices made when designing the syntax of the language. Other more eye-opening feedback was the need of a preprocessor. The test users felt the task of inserting semicolons (;) at the end of expressions tedious and that it lowered the overall writeability of the language by a great deal. The choice of making the preprocessor a secondary priority was quickly reconsidered.

8 Discussion

This chapter evaluates the result of the project and describes the difficulties and excluded features of it. It also discusses what could have been done differently and speculates in the future of the language Lollipop.

8.1 Summary of the project

The purpose of the project was to construct and implement a small but generally applicable programming language with support for both linear and non-linear types. The intention of integrating linear types into the type system of the language was to give programmers, not previously familiar with the concept, a first platform to explore the concept, thus spreading it from the academic world to the programming community. The project strived for minimalism in the language, putting focus on the construction of an environment for programmers to explore the concept of linear types in, rather than building a language for extensive software development.

Today, linear types mainly exist in the academic world and the concept has not gotten full impact in any larger programming language, hence not reaching the uninitiated programmer. The theory of linear types is based on linear logic and an understanding of the underlying theory requires knowledge in lambda calculus, logic and type theory. This creates an obstacle for programmers outside the academic world who want to get an understanding of and explore the concept.

A functional programming language called Lollipop has been implemented. As the concept of linear types and the science of writing programming languages were new to the majority of the project group, the first weeks of the project were mainly spent on studying the subjects. The designing of the language also took place during these weeks. Thereafter, the process of writing the grammar and the interpreter of the language started. After implementing a first version of both the front-end and the back-end, they were connected by converting the surface AST into the inner AST that the interpreter used. Halfway into the project the implementation of the type checker commenced.

To evaluate how well the purpose was reached, tests — in the form of example programs — of the language were written. Before the back-end and front-end were connected, separate tests were written for the interpreter and the grammar. After connecting the parts, a test file containing data declarations and functions was written to test the entire process of parsing and interpreting programs, and later also type checking. These tests captured a number of bugs in the code.

To test the readability and writability of the syntax and semantics of the language, cognitive tests were carried out during the project. The tests consisted of example functions which the test persons were asked to interpret or implement, before providing feedback by filling out a form of what they found good and bad in the language.

8.2 Analysis of the result

As mentioned, the project resulted in a small, yet sweet language called Lollipop. Alongside the language an interpreter called Loli was written. As Loli is an interactive evaluation environment for Lollipop code and uses a read-evaluate-print-loop it handles the main interaction with the language for the programmer. Given a program, Loli calls the lexer and parser to get a syntax generated AST representation of the program, which is passed to the converter which returns the internal AST representation. In this representation, the program is then passed to the type inferencer that infers types for expressions with no explicit type declaration and checks than the program is type correct. Finally, Loli calls the evaluator which evaluates the program and returns its value, which is printed by Loli as the last step in an iteration of the read-evaluate-print-loop. Due to the limited time frame of the project, Loli supports a rather limited number of commands and is not at all as versatile as its counterpart used with Haskell, GHCi.

8.2.1 Evaluation of how well Lollipop functions

As this is a small project carried out by beginners within the field of language development and type theory, Lollipop lacks many of the features of greater languages. A more extensive account of the excluded features of the language can be read further down in this chapter. Although being a small language, Lollipop can still be considered as general-purpose, as it is applicable in a several application domains. It is written in a modular way, facilitating the implementation of future extensions.

Here follows a list of how well the different parts of the tool chain, that the implementation of the language consists of, function:

- The interpreter **Loli** successfully manages to pass the test program between the other parts of the tool chain. It mostly manages to catch the exceptions that are thrown when something in the loaded program is incorrect, such as type errors or parse errors.
- The lexer and parser generated from the **grammar** written in BNFC successfully parse all syntactically correct Lollipop programs. The grammar accepts a greater set of programs than the ones valid in the language, but this is hard to avoid when writing grammars and is therefore not considered a failure. The grammar also contains a preprocessor, which removes the requirement to terminate all rows with a semicolon when writing programs in Lollipop, which is considered a great advantage.
- The **converter** module manages to convert all structures in the BNFC generated AST into a suitable representation in the internal AST. It weeds out many of the invalid programs that the grammar accept, removes syntactic sugar and behaves as intended in general. The converter is therefore considered successfully implemented.
- The **type inferencer** successfully manages to check that top level functions are type correct by inferring the type of the function body and check that it matches the declared type of the function. It can infer types for anonymous functions but not for recursive let-in expressions or linear types. The type inferencer is therefore not considered fully, but hitherto successfully, implemented.
- The **evaluator** successfully manages to evaluate all correct Lollipop expressions that are sent to it. It uses the lazy evaluation technique and also has support for partial application of functions. It is considered successfully implemented.
- Variables can be declared as being of **linear types** in function declarations and used in a linear manner. The type checker correctly raises type errors when the linear rules of consumption – that a linear type is used exactly once – are not followed.

8.2.2 The importance of user feedback

The cognitive tests were executed at a point in the project when there was a stable version of the grammar and the evaluator of the interpreter and the language was almost as extensive as at the final point of the project. There was no working version of the type inferencer at that point, but the users still had to provide type declarations for the functions for the program to parse. Also, the support for user defined data types was not yet implemented at that point, so the programs that the test persons were asked to write or read only consisted of functions. Moreover, the preprocessor had not yet been written, which enforced the users to terminate each row with a semicolon (;). This lowered the writeability of Lollipop according to the test persons, which made a preprocessor a higher priority than it had been before the user tests. Despite the tediousness of writing semicolons, the grade the test persons set on the writeability averaged at a 4 out of 5. There has not been any more user tests after the addition of the preprocessor, but hopefully the writeability has increased since then. The readability and intuitiveness of programming in the language also received an average grade of 4 out of 5. An important note to make is that all test persons had previous experiences in functional programming.

Two syntactical differences between Haskell and Lollipop are the notation of guards, where Haskell uses the bar (|) notation and has the predicate to be guarded upon first in the guard expression, while Lollipop uses the *when* keyword and has the predicate last in the guard expression; and the marking of definitions, where Haskell uses a single equality mark (=), while Lollipop uses a colon followed by an equality mark (:=). The *when* notation in guards was appreciated and found intuitive by the majority of the test users, while the (:=) notation of definitions was not appreciated by one of them. However, as the (:=) was not disliked by a majority of the test users, and was found intuitive by the developers of Lollipop, as it is often used in mathematical definitions, the choice of notation was not changed after the feedback.

As the language has continued to develop after the user tests were carried out, the result of them are not completely representative of how the language would be graded at the end of the project. They could however give a hint of how the language would be received by the programming community if made available. An example of a feature that was never given any feedback on was linear types. The purpose of Lollipop was to provide a way for programmers to explore and gain an understanding of the concept of linear types, but no cognitive user tests have been conducted after the support for them was added to the language. This makes it difficult to evaluate how well Lollipop serves as an intuitive first platform to explore the concept, which of course is a big disadvantage. The Lollipop syntax for linear types is yet considered clear and intuitive, with the "i"-notation for linear types and the lollipop (-o) notation for linear functions, so user feedback would probably not affect the syntax in a greater extent anyway. The semantics of linear variables and functions would naturally not be modified even if the test persons would have found it unintuitive, as they have to follow the type rules of linear types. So after all, user tests would probably not influence how linear types are implemented in the language much. Although, this would be interesting to hear, in order to get an indication of how well Lollipop would serve as a first platform for exploration of the concept of linear types.

8.3 Difficulties

The main difficulty during the planning phase was the limit of time in relation to the group members' sparse amount of previous knowledge within the fields of linear types and writing interpreters and type checkers for functional languages. Researching took a great amount of the given time, leaving less time for the actual implementation. The lack of previous experience also made it difficult to create a realistic time plan, as it was hard to predict how time consuming each step would become.

During the first part of the implementation phase, while writing the grammar, there was an issue regarding the test suite: without a working grammar, the interpreter could not be tested using a test file written in Lollipop syntax. Instead, a smaller test-program was written (in the AST) in order to ensure that the interpreter functioned as expected. The test-program consisted of a collection of functions testing the core functionality of evaluation in the interpreter. These programs were essentially constructed by using the corresponding data structures defined in the AST designed in Haskell.

Another issue regarding the testing was that of the interpreter and evaluator. As the interpreter requires programs to be type correct, such programs had to be provided in order to test the evaluation. With this said, programs could not be generated using any Haskell testing library, such as QuickCheck, as these would generate random programs, not knowing if they are type correct or not.

The converter compared all the data types of the, by BNFC, generated AST with those in the internal AST, making reasonable assumptions about how the different parts of the AST would translate. This proved to be far more complex than it had seemed from the beginning, as a great deal of optimization was needed, as user syntax in most cases was unnecessarily complex and contained much "syntactic sugar". As the grammar file, and to some degree also the inner representation, was modified during the project, the converter had to be rewritten multiple times as the AST's changed.

Implementation of type checking and type inferencing was thought to be somewhat difficult from the beginning, but was later on proven to be a much more difficult task then initially expected. A lot of work was put into even

understanding the concept of how this was to be implemented, leading to very time consuming work. With a lot of support from the supervisor, this became doable, even though it still was a real challenge.

8.4 Excluded features

Lollipop is a small language and lacks many of the features that larger languages such as Haskell have, making it less widely applicable. The purpose was however not to make a great new language for software development, but rather a small platform for the exploration of the concept of linear types. Due to limited time scope of the project, delimitations were naturally made in order to focus on the most important features.

Early in the process of the project, possible features of the language were discussed and divided into three priority groups: hot, that mainly consisted of features that were required for the language to be general purpose, plus the support for linear types; cool, which were features that would be nice to include in the language, but not necessary; and cold, which were features of low priority and would probably not be implemented. As the project proceeded, the priority groups changed slightly, so some of the hot features fell down the priority list while some of the cool ones were found more necessary.

8.4.1 Syntax

Module-notation and the possibility for modules to import other modules was for the main part of the project considered a hot feature. But towards the end of the project, it was prioritized down. Implementing module-notation and support for the "import" command would consume much time and extra work, as it probably would require an extra phase in the process; as the contents in the imported module would have to be added to the importer module before the evaluation of it, it could not be evaluated together with the other commands in the existing phase. It was not considered worth to sacrifice this extra amount of time, when other — also considered important — features were also left to be implemented at that point. The absence of modules forces the programmer to write all the code for a program in one single file, which discourages larger programs. Although Lollipop is mainly meant to be a platform for exploring the concept of linear types and not a language for development of large programs, module notation is a desirable property in the language, and is therefore a reasonable future extension of the language.

In Lollipop, function guards must always have a default case, that is, a case without a predicate. In Haskell this is written by using *otherwise*. The expression in the default case is evaluated if none of the predicates to the other cases are fulfilled. This enforcement of a default was not intended at first, but made the implementation of the converter easier. Of course, it can be tedious for the programmer to always be forced to write a default guard when there is no need for any, but it is not considered a great delimitation on the writeability of the language.

Although Lollipop has support for user defined data types and functions, it does not support user defined infix operators, such as + or >>=. This has never been of high priority, as such operators, although intuitively more suitably represented as infix, easily could be modelled as prefix functions. The support for this feature would add much work in the implementation of the grammar, which was not the focus of this project.

Partial application of infix operators is not supported in Lollipop. This can be of some annoyance when calling higher-order functions, as e.g. the expression `any (>2) [1,2,3,4]` is illegal in the language. However, this is not considered a great disadvantage as that expression easily can be rewritten with an anonymous function: `any (\x -> x>2) [1,2,3,4]`. It would however make a nice future extension of the language.

8.4.2 Type system

Support for type synonyms was for the majority of the project part of the prioritized list of features. As the implementation of the type system was not commenced until late in the process, and other type system features —

for instance type inference and linear types — were considered more important, this feature was however prioritized away. This is not considered a great loss because, however useful when using large data types, type synonyms do not add any functionality to a language.

A more useful feature that was also planned but prioritized away is type classes with the "deriving" command. This had a higher priority than type synonyms, but turned out to be a difficult feature to implement, which there was no time for. This is however a possible future extension of the language.

8.4.3 Linear types

One field in which linear types can be useful is I/O handling. Initially, I/O in Lollipop was planned to be handled using linear types to ensure safer I/O streams. However, as I/O was one of the highest-priority features that had to be implemented early in the implementation phase, and linear types were implemented late, I/O was implemented in another way. As the implementation of linear types was heavily delayed from schedule, there was no time to reimplement I/O to utilize them.

The check that a linear variable is used exactly once is not exhaustive. It only goes through the syntax of a program and checks that the variable occurs exactly once there, but as Lollipop is lazy, an occurrence of a variable in he syntax does not ensure that the variable is evaluated, i.e. consumed.

The chapter describing the implementation of the language mentioned that an error-reporting type checker was used for Lollipop, while an annotating checker would have been more desirable. An annotating checker returns an AST with the corresponding type for each expression and sub-expression, which can then be passed to the evaluator. Type information in the evaluation phase is required for checking that a linear variable is actually consumed, and not just present in the syntax. There was however not enough time to implement an annotating checker, which was why an error-reporting checker was chosen instead.

8.5 What could be done differently

Given the outcome of the project, there are some parts of the language which may seem unnecessary and could have been assigned with a lower priority in order to focus more on the feature of linear types. Syntactic sugar, such as if-statements and guards, could have been left out. As mentioned, the implemented feature of linear types is a minimal one as the work on it did not start until later in the project. The main focus lay on implementation of a general purpose language and less focus lay on the implementation of linear types, which is something that could have been done differently, since this was intended to be the unique feature of Lollipop. This is the main reason why some intended features of linear types, such as handling of I/O-streams were left out.

Another area which could have been started on earlier was the implementation of a type checker. As this did not seem as such a time consuming and demanding task at first, this was also something that was postponed until a later part of the project. The implementation of the type checker could have been done more in parallel with the interpreter. It could essentially have been started on as soon as the AST and the environment were created, as these are the only two modules that the type checker is using.

8.6 Future of Lollipop

Given the current state of the language, with its minimal implementation of linear types, further work on the support of them with implementation of type inference is necessary. An analysis of the type correctness of linear types in the language and their as potential uses in general is also necessary. As mentioned in chapter 3 (Scope), there are several essential features of a thorough programming language that has been left out from Lollipop,

such as module notation and type classes that could serve as future extensions of the language. Something that would give the developers more control of the language, rather than being so dependent on Haskell, would be to write a compiler, intended to replace the current interpreter. A further expansion of the language could also be to broaden the current limitation that the end users need to declare the use of linear types in the type declaration of functions, allowing linear types in anonymous functions and in let-in expressions. Raising the restriction of type declarations for every top level function in general is yet another addition worth to consider.

9 Conclusion

A working prototype language, Lollipop, is now complete with a majority of the planned functionality implemented. The language can serve as an initial platform for both uninitiated as well as more experienced programmers interested in the practical uses of linear types in a functional programming environment. The minimal implementation makes it a general and extensible proof-of-concept which is useful when used as support for future development.

The feedback from cognitive user tests was positive and indicated that a more complete version of the language would be well received. Unfortunately no tests were performed after the implementation of linear types, so it is difficult to evaluate the reception of this among end users. From a technical standpoint it works well and our conclusion is that there should be no major pitfalls here given a sane syntactic notation and intuitive errors.

Future research should be directed towards the practical implementation of linear types in programming languages, especially to ascertain interaction with type inference and implementation details of a language. In a shorter time frame there are particularly two proposed extensions of the language. First would be to create a transpiler to Haskell in order to benefit from the highly evolved GHC, when doing this it could also be possible to make performance increasing conversions from the guarantees that linearity provides in some cases. Second would be to further develop the type system, lifting the constraint of forced type declaration and possibly integrating inference for linear types.

Linear types is an area with a lot of potential and we are looking forward to seeing exciting progress in the upcoming years.

References

[1] J.-Y. Girard, *Linear logic*, 1987. DOI: 10.1016/0304-3975(87)90045-4. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397587900454.

[2] P. Lincoln and J. Mitchell, Operational aspects of linear lambda calculus, *[1992] Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science* 1992, 235–246, 1992. DOI: 10.1109/LICS.1992.185536. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=185536.

[3] A. A. Aaby, Theory introduction to programming languages 2004, 2004. [Online]. Available: http://staffweb.worc.ac.uk/DrC/Courses%202006-7/Comp%204070/Reading%20Materials/book[1].pdf.

[4] M. Rochkind, *Advanced UNIX Programming*. Pearson Education, 2004, ISBN: 9780132466134. [Online]. Available: https://books.google.se/books?id=vlM72G-1VDkC.

[5] "Haskell", [Online]. Available: https://haskell.org (visited on 03/30/2016).

[6] Http://c2.com/, *Linear types*, 2014. [Online]. Available: http://c2.com/cgi/wiki?LinearTypes (visited on 03/02/2016).

[7] P. Wadler, A taste of linear logic, *Mathematical Foundations of Computer Science 1993* 1993, 1993. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-57182-5_12.

[8] "Clean", [Online]. Available: http://clean.cs.ru.nl/Language_features (visited on 05/13/2016).

[9] "Idris", [Online]. Available: http://idris-lang.org (visited on 03/30/2016).

[10] "Rust", [Online]. Available: https://www.rust-lang.org/ (visited on 03/30/2016).

[11] S. S. Muchnick, *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997, ISBN: 9781558603202. [Online]. Available: https://books.google.se/books?id=Pq7pHwG1_OkC.

[12] E. D. Reilly, *Concise Encyclopedia of Computer Science*, E. D. Reilly, Ed. John Wiley & Sons, 2004, 2004, p. 875, ISBN: 0470090952, 9780470090954. [Online]. Available: https://books.google.se/books?id=5Jaa1BVverIC.

[13] P. Van-Roy and S. Haridi, *Concepts, Techniques, and Models of Computer Programming*. Prentice-Hall, 2004, ISBN: 9780262220699. [Online]. Available: https://books.google.se/books?id=_bmyEnUnfTsC.

[14] J. Rubin, D. Chisnell, and J. Spool, *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*. Wiley, 2011, ISBN: 9781118080405. [Online]. Available: https://books.google.se/books?id=l%5C_e1MmVzMb0C.

[15] R. Craig and S. Jaskiel, *Systematic Software Testing*, ser. Artech House ITS library. Artech House, 2002, ISBN: 9781580537926. [Online]. Available: https://books.google.se/books?id=2%5C_gbZYZcZXgC.

[16] M. J. Harrold, R. Gupta, and M. L. Soffa, A methodology for controlling the size of a test suite, *ACM Trans. Softw. Eng. Methodol.* vol. **2**, no. 3 Jul. 1993, 270–285, Jul. 1993, ISSN: 1049-331X. DOI: 10.1145/152388.152391. [Online]. Available: http://doi.acm.org/10.1145/152388.152391.

[17] "BNFC", [Online]. Available: http://bnfc.digitalgrammars.com (visited on 03/30/2016).

[18] R. Milner, A theory of type polymorphism in programming, *Journal of Computer and System Sciences* vol. **17** 1978, 348–375, 1978.

[19] P. Wadler, Linear types can change the world, *IFIP TC* 1990, 1990. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.55.5439&rep=rep1&type=pdf.

[20] "Git", [Online]. Available: https://git-scm.com/ (visited on 03/02/2016).

[21] "Github", [Online]. Available: https://github.com/m0ar/lollipop/ (visited on 03/30/2016).

[22] "Trello", [Online]. Available: https://trello.com (visited on 03/30/2016).

[23] "Mendeley", [Online]. Available: https://mendeley.com (visited on 03/30/2016).

[24] "Overleaf", [Online]. Available: https://overleaf.com (visited on 03/30/2016).

[25] "Google drive", [Online]. Available: https://google.com/drive/ (visited on 03/30/2016).

[26] "Slack", [Online]. Available: https://slack.com (visited on 03/30/2016).

[27] D. Grune and C. Jacobs, *Parsing Techniques: A Practical Guide*, ser. Monographs in Computer Science. Springer New York, 2007, ISBN: 9780387689548. [Online]. Available: https://books.google.se/books?id=05xA_d5dSwAC.

[28] M. Forsberg and A. Ranta, The labelled BNF grammar formalism, *Converter* 2005, 1–13, 2005. [Online]. Available: http://www.cs.chalmers.se/Cs/Research/Language-technology/BNFC/doc/LBNF-report.pdf.

[29] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. ADDISON WESLEY Publishing Company Incorporated, 2007, ISBN: 9780321547989. [Online]. Available: https://books.google.se/books?id=WomBPgAACAAJ.

[30] D. Grune, K. van Reeuwijk, H. Bal, C. Jacobs, and K. Langendoen, *Modern Compiler Design*. Springer New York, 2012, ISBN: 9781461446996. [Online]. Available: https://books.google.se/books?id=zkpFTBtK7a4C.

[31] A. Ranta, *Implementing Programming Languages: An Introduction to Compilers and Interpreters*. College Publications, 2012, ISBN: 9781848900646.

[32] M. Grabmüller, Algorithm W step by step, [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.65.7733.

[33] A. Ranta, *Lecture 7: type checking*, 2011. [Online]. Available: http://www.cse.chalmers.se/edu/year/2011/course/TIN321/lectures/proglang-07.html (visited on 05/31/2016).

Part I

Appendix

Appended is the full surface syntax definition, report generated from BNFC.

The Language grammar

BNF-converter

May 12, 2016

This document was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

The lexical structure of grammar

Literals

TypeId literals are recognized by the regular expression ⟨*upper*⟩(⟨*letter*⟩ | ⟨*digit*⟩ | '_' | '')*

Id literals are recognized by the regular expression ⟨*lower*⟩(⟨*letter*⟩ | ⟨*digit*⟩ | '_' | '')*

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in grammar are the following:

```
case      datatype  else
function  if        import
in        let       not
of        then      type
when
```

The symbols used in grammar are the following:

```
;     (     )
:     :=    |
_     []    [
]     ,     -
++    ^     *
/     +     <
>     <=    >=
==    !=    &&
||    >>=   >>
{     }     \
->    ()    -o
```

Comments

Single-line comments begin with $--$.
Multiple-line comments are enclosed with $\{-$ and $-\}$.

The syntactic structure of grammar

Non-terminals are enclosed between $\langle$ and $\rangle$. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

$\langle Program \rangle$ ::= $\langle Import \rangle$ `;` $\langle Program \rangle$
| $\langle Program1 \rangle$

$\langle Program1 \rangle$ ::= $\langle Declaration \rangle$ $\langle Program1 \rangle$
| $\langle Declaration \rangle$
| `(` $\langle Program \rangle$ `)`

$\langle Import \rangle$ ::= `import` $\langle Id \rangle$

$\langle Declaration \rangle$ ::= `function` $\langle Id \rangle$ `:` $\langle Type \rangle$ `;` $\langle ListDef \rangle$
| `datatype` $\langle TypeIdent \rangle$ $\langle ListId \rangle$:= $\langle ListConstr \rangle$ `;`
| `type` $\langle TypeIdent \rangle$ $\langle ListId \rangle$:= $\langle Type \rangle$ `;`

$\langle ListConstr \rangle$ ::= $\langle Constr \rangle$
| $\langle Constr \rangle$ | $\langle ListConstr \rangle$

$\langle Def \rangle$ ::= $\langle Id \rangle$ $\langle ListArg \rangle$:= $\langle Exp \rangle$
| $\langle Id \rangle$ $\langle ListArg \rangle$ $\langle Guards \rangle$

$\langle ListDef \rangle$ ::= $\langle Def \rangle$ `;`
| $\langle Def \rangle$ `;` $\langle ListDef \rangle$

⟨*Arg*⟩ ::= ⟨*Pattern*⟩

⟨*ListArg*⟩ ::= ϵ
| ⟨*Arg*⟩ ⟨*ListArg*⟩

⟨*Cons*⟩ ::= `(` ⟨*TypeId*⟩ ⟨*Id*⟩ ⟨*ListId*⟩ `)`
| ⟨*TypeId*⟩

⟨*Guards*⟩ ::= `:=` ⟨*Exp*⟩ `when` ⟨*Exp*⟩ ⟨*Guards1*⟩

⟨*Guards1*⟩ ::= `:=` ⟨*Exp*⟩ `when` ⟨*Exp*⟩ ⟨*Guards1*⟩
| `:=` ⟨*Exp*⟩
| ϵ

⟨*Constr*⟩ ::= ⟨*TypeIdent*⟩ ⟨*ListTypeParameter*⟩

⟨*TypeParameter*⟩ ::= ⟨*Type2*⟩

⟨*ListTypeParameter*⟩ ::= ϵ
| ⟨*TypeParameter*⟩ ⟨*ListTypeParameter*⟩

⟨*Pattern1*⟩ ::= `_`
| ⟨*Id*⟩
| ⟨*TypeId*⟩
| ⟨*Literal*⟩
| `[]`
| `[` ⟨*ListPat*⟩ `]`
| `(` ⟨*Pattern*⟩ `,` ⟨*Pattern*⟩ `)`
| `(` ⟨*Pattern*⟩ `,` ⟨*Pattern*⟩ `,` ⟨*Pattern*⟩ `)`
| `(` ⟨*TypeIdent*⟩ ⟨*Pattern1*⟩ ⟨*ListPattern1*⟩ `)`
| `(` ⟨*Pattern*⟩ `)`

⟨*Pattern*⟩ ::= ⟨*Pattern1*⟩ `:` ⟨*Pattern*⟩
| ⟨*Pattern1*⟩

⟨*ListPattern1*⟩ ::= ϵ
| ⟨*Pattern1*⟩ ⟨*ListPattern1*⟩

⟨*ListPat*⟩ ::= ⟨*Pattern*⟩
| ⟨*Pattern*⟩ `,` ⟨*ListPat*⟩

⟨*ListPattern*⟩ ::= ϵ
| ⟨*Pattern1*⟩ ⟨*ListPattern*⟩

⟨*Literal*⟩ ::= ⟨*Integer*⟩
| ⟨*Double*⟩
| ⟨*Char*⟩
| ⟨*String*⟩

⟨*ListLiteral*⟩ ::= ⟨*Literal*⟩
| ⟨*Literal*⟩ `,` ⟨*ListLiteral*⟩

⟨*ListTypeId*⟩ ::= ⟨*TypeId*⟩
| ⟨*TypeId*⟩ `,` ⟨*ListTypeId*⟩

⟨*ListId*⟩ ::= ϵ
| ⟨*Id*⟩ ⟨*ListId*⟩

⟨*Exp11*⟩ ::= ⟨*Id*⟩
| ⟨*Tuple*⟩
| ⟨*Literal*⟩
| ⟨*Cons*⟩
| `[` ⟨*ListExp*⟩ `]`
| `[]`
| `(` ⟨*Exp*⟩ `)`

⟨*Exp10*⟩ ::= ⟨*Exp10*⟩ ⟨*Exp11*⟩
| ⟨*Exp11*⟩

⟨*Exp9*⟩ ::= `not` ⟨*Exp10*⟩
| $-$ ⟨*Exp10*⟩
| ⟨*Exp9*⟩ $++$ ⟨*Exp10*⟩
| ⟨*Exp9*⟩ `:` ⟨*Exp10*⟩
| ⟨*Exp10*⟩

⟨*Exp8*⟩ ::= ⟨*Exp8*⟩ `^` ⟨*Exp9*⟩
| ⟨*Exp9*⟩

⟨*Exp7*⟩ ::= ⟨*Exp7*⟩ `*` ⟨*Exp8*⟩
| ⟨*Exp7*⟩ `/` ⟨*Exp8*⟩
| ⟨*Exp8*⟩

⟨*Exp6*⟩ ::= ⟨*Exp6*⟩ $+$ ⟨*Exp7*⟩
| ⟨*Exp6*⟩ $-$ ⟨*Exp7*⟩
| ⟨*Exp7*⟩

⟨*Exp5*⟩ ::= ⟨*Exp5*⟩ $<$ ⟨*Exp6*⟩
| ⟨*Exp5*⟩ $>$ ⟨*Exp6*⟩
| ⟨*Exp5*⟩ $<=$ ⟨*Exp6*⟩
| ⟨*Exp5*⟩ $>=$ ⟨*Exp6*⟩
| ⟨*Exp6*⟩

⟨*Exp4*⟩ ::= ⟨*Exp4*⟩ == ⟨*Exp5*⟩
| ⟨*Exp4*⟩ != ⟨*Exp5*⟩
| ⟨*Exp5*⟩

⟨*Exp3*⟩ ::= ⟨*Exp3*⟩ && ⟨*Exp4*⟩
| ⟨*Exp4*⟩

⟨*Exp2*⟩ ::= ⟨*Exp2*⟩ || ⟨*Exp3*⟩
| ⟨*Exp3*⟩

⟨*Exp1*⟩ ::= let ⟨*Id*⟩ := ⟨*Exp2*⟩ in ⟨*Exp*⟩
| ⟨*Exp2*⟩ >>= ⟨*Exp1*⟩
| ⟨*Exp2*⟩ >> ⟨*Exp1*⟩
| case ⟨*Exp*⟩ of { ⟨*Cases*⟩ }
| if ⟨*Exp2*⟩ then ⟨*Exp2*⟩ else ⟨*Exp*⟩
| \ ⟨*Id*⟩ ⟨*ListId*⟩ −> ⟨*Exp*⟩
| ⟨*Exp2*⟩

⟨*Exp*⟩ ::= ⟨*Exp1*⟩

⟨*ListExp*⟩ ::= ϵ
| ⟨*Exp*⟩
| ⟨*Exp*⟩ , ⟨*ListExp*⟩

⟨*Cases*⟩ ::= ⟨*Pattern*⟩ −> ⟨*Exp*⟩ ; ⟨*Cases1*⟩

⟨*Cases1*⟩ ::= ⟨*Pattern*⟩ −> ⟨*Exp*⟩ ; ⟨*Cases1*⟩
| ⟨*Pattern*⟩ −> ⟨*Exp*⟩

⟨*Tuple*⟩ ::= (⟨*Exp*⟩ , ⟨*Exp*⟩)
| (⟨*Exp*⟩ , ⟨*Exp*⟩ , ⟨*Exp*⟩)

⟨*Type2*⟩ ::= ⟨*TypeIdent*⟩
| (⟨*Type*⟩ , ⟨*ListType*⟩)
| [⟨*Type*⟩]
| ()
| (⟨*Type*⟩)

⟨*Type1*⟩ ::= ⟨*Type2*⟩ −> ⟨*Type1*⟩
| ⟨*Type2*⟩ −o ⟨*Type1*⟩
| ⟨*Type2*⟩

⟨*Type*⟩ ::= ⟨*Type*⟩ ⟨*Type1*⟩
| ⟨*Type1*⟩

⟨*ListType*⟩ ::= ⟨*Type*⟩
| ⟨*Type*⟩ , ⟨*ListType*⟩

$$
\begin{array}{lll}
\langle \mathit{TypeIdent} \rangle & ::= & \langle \mathit{TypeId} \rangle \\
& | & \langle \mathit{Id} \rangle
\end{array}
$$