Editorial support for first-year students in programming languages at Aarhus University

Jeppe Welling Hansen, 20061245

Master's Thesis, Computer Science
August 2013
Adviser: Olivier Danvy

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

This thesis presents an editorial support system developed for first-year students following the programming-languages course at Aarhus University. The system is built for students writing Scheme code in Emacs. The editorial support system is developed as a mode for Emacs called the `bnf-mode`.

The `bnf-mode` provides editorial support by checking the syntax correctness of the Scheme code written by the student, but it is generic. Given any context-free grammar of fully-parenthesized expressions in BNF, the `bnf-mode` provides an interactive environment in which one can check whether the content of the current buffer conforms to the given BNF.

The `bnf-mode` was evaluated by the students attending the programming-languages course in the spring of 2013.

Resumé

Dette speciale præsenterer et redigeringsværktøj, udviklet til førsteårsstuderende, som følger programmeringssprogskurset ved Aarhus Universitet. Værktøjet er udviklet til studerende, som skriver Scheme kode i Emacs. Redigeringsværktøjet er udviklet som en udvidelse til Emacs kaldet `bnf-mode`.

Værktøjet hjælper med koderedigering ved at kontrollere syntaktisk korrekthed af Scheme kode, skrevet af den studerenede, men værktøjet er også generisk. Givet en kontekstfri grammatik af parentesudtryk i BNF, tilbyder værktøjet et interaktivt miljø, hvori det er let at kontrollere, om Emacs-bufferens indhold accepteres af den givne BNF .

Værktøjet blev evalueret af de studerende, som fulgte programmeringssprogskurset i foråret 2013.

Acknowledgements

This thesis combines my enthusiasm for programming and computer science, and my interaction with the students who followed the programming-languages course in the spring of 2013. My adviser, Olivier Danvy, made it possible to materialize this thesis project in the context of his course and to test it in collaboration with the students following the course. He provided enthusiastic advice, great guidance, motivation on rainy days, but most importantly he gave his time to listen, talk and laugh. For that I am truly grateful.

It was rewarding to see the students using the software implemented as part of this thesis project, the `bnf-mode`. Among them, Søren Christensen, Manuel Rafael Ciosici, Christopher Riis Bubeck Eriksen, Alex Fuller Fischer, Kristján Rafn Gunnarsson, Chris Haysen, Sine Jespersen, Ulloriaq Kleinschmidt Knudsen, Martin Berndt Nøhr, Tobias Sommer, and Sebastian Sander Søndergaard provided valuable feedback on the early versions of the `bnf-mode`.

Casper Svenning Jensen let me follow his TA sessions throughout the 7 weeks of the course and TA Lau Skorstengaard provided instrumental feedback on the `bnf-mode`.

Through a strike of luck I shared an office with Kent Grigo. I remember fondly our late discussions about Scheme, Elisp, type errors and functional programming.

My parents, Ole and Conny, provided tireless love, enthusiastic support, and endless solicitude throughout my whole education, as well as a peaceful place in the countryside which I first got to truly appreciate during my time as a university student. I am grateful to Casper and Stine, my siblings, for their always enjoyable company, cheerful spirit, and supporting shoulders.

Thank to computer scientist, fellow triathlete, and roommate Steffen Daniel Jensen both for enlightening discussions about programming languages and for his reasoned enthusiasm.

Over the last year, I have spent many hours every week at the stadium swimming pool, and during that time, Lifeguard Jakob Martini Jørgensen has proved a great swimming teacher and friend.

Finally, nothing happens in a vacuum, and so I want to extend grateful thanks to my fellow students, to my lecturers and teaching assistants, to our system administrators, and to the staff who make the Department of Computer Science at Aarhus university such an inspiring place where to study.

Jeppe Welling Hansen,
Aarhus, August 14, 2013.

Contents

Abstract iii

Resumé v

Acknowledgments vii

1 Introduction 3
- 1.1 Expressing computation . . . 3
- 1.2 Teaching programming languages in 7 weeks . . . 4
- 1.3 Equalizing the students . . . 4
- 1.4 The contribution of this thesis . . . 5
- 1.5 Overview . . . 8
- 1.6 Prerequisites . . . 8

2 Editorial support through BNF-based syntax checking 9
- 2.1 Overview . . . 9
- 2.2 Avoid advanced syntax for novice students . . . 10
  - 2.2.1 Avoiding novice errors by restricting the programming language grammar . . . 10
  - 2.2.2 Complete function definitions . . . 11
- 2.3 Error precision: Grammatical evaluation and constraints . . . 12
  - 2.3.1 Evaluating input syntax against a BNF . . . 12
  - 2.3.2 The plain-text evaluation vs. the S-expression evaluation 13
  - 2.3.3 Imposed constraints for improved error feedback . . . 14
- 2.4 Grammars supported by the `bnf-mode` . . . 16
  - 2.4.1 Grammatical root . . . 17
  - 2.4.2 Built-in productions . . . 19
- 2.5 Grammatical validation of input syntax . . . 19
  - 2.5.1 General grammatical checks . . . 20
  - 2.5.2 BNF error report level . . . 21
  - 2.5.3 Partial-match order . . . 21
  - 2.5.4 Resolving references . . . 22
  - 2.5.5 Kleene star and plus optimized for common case . . . 23
  - 2.5.6 Adjustments for Scheme . . . 25
- 2.6 Editorial environment . . . 25
- 2.7 Summary . . . 26

3 Implementation **29**
3.1 Implementation language . . . 29
3.2 Limitations of Elisp . . . 29
3.2.1 Overview . . . 30
3.3 Elisp: one language 4 distinct function representations . . . 31
3.3.1 Function representation with dynamic scope . . . 31
3.3.2 Function representation using the lexical-let construct with lexical-binding disabled . . . 31
3.3.3 Function representation using the let construct with lexical-binding enabled . . . 32
3.3.4 Function representation using the lexical-let construct with lexical-binding enabled . . . 34
3.3.5 Summary . . . 34
3.4 Stack neutral functions in Elisp by function transformations . . . 34
3.4.1 The original function . . . 35
3.4.2 CPS transformation . . . 36
3.4.3 Trampoline transformation . . . 37
3.4.4 Defunctionalization . . . 39
3.4.5 Benchmarks . . . 41
3.5 Parsing . . . 44
3.5.1 BNF-AST . . . 45
3.5.2 S-expression-AST . . . 45
3.5.3 Parsing: common cases . . . 45
3.5.4 Handling Scheme's syntactic sugar . . . 46
3.5.5 Handling string literals . . . 46
3.5.6 The BNF parser . . . 48
3.5.7 The S-expression parser . . . 48
3.5.8 Limitations of the parsers . . . 49
3.5.9 Summary . . . 49

4 User evaluation **51**
4.1 Errors caught by the `bnf-mode` . . . 51
4.1.1 Many students, many versions of Emacs . . . 52
4.1.2 Missing function application . . . 52
4.1.3 Missing `else` in a `cond`-statement . . . 52
4.1.4 Missing else-branch in an `if`-statement . . . 53
4.1.5 Discovering unexpected errors . . . 53
4.1.6 Overwriting built-in functions . . . 53
4.1.7 Undetected errors . . . 54
4.1.8 Summary . . . 54

5 Related work **57**
5.1 Form over function . . . 57
5.1.1 Summary . . . 57
5.1.2 Analysis . . . 58
5.2 Measuring the effectiveness of error messages designed for novice programmers . . . 59

5.2.1 Summary 59
5.2.2 Analysis 59
5.3 The structure and interpretation of the computer science curriculum 59
5.3.1 Summary 60
5.3.2 Analysis 60
5.4 We Scheme 61
5.4.1 Summary 61
5.4.2 Analysis 61
5.5 DrScheme: a programming environment for Scheme 62
5.5.1 Summary 62
5.5.2 Analysis 63
5.6 JazzScheme: evolution of a Lisp-based development system . . . 63
5.6.1 Summary 63
5.6.2 Analysis 64
5.7 Identifying and correcting Java programming errors for introductory computer science students 64
5.7.1 Summary 64
5.7.2 Analysis 65
5.8 BlueJ 65
5.8.1 Summary 65
5.8.2 Analysis 65
5.9 Debugging: the good, the bad, and the quirky - a qualitative analysis of novices' strategies 66
5.9.1 Summary 66
5.9.2 Analysis 67
5.10 Use of a syntax checker to improve student access to computing . 67
5.10.1 Summary 67
5.10.2 Analysis 68
5.11 Summary of related work 68

6 Conclusion **71**

Primary Bibliography **72**

Secondary Bibliography **73**

7 Appendix **77**
7.1 Subset of Scheme 77
7.2 Installing the `bnf-mode` 79

Chapter 1

Introduction

The goal of this thesis project is to provide editorial support for first-year students following the programming-languages course at Aarhus University [A2]. In this course, the programming language Scheme is used as the language of discourse and Emacs is used as the development platform, in combination with the Petite Chez Scheme interpreter.

Before elaborating further on the goal and contributions of this thesis project (Section 1.4), the context in which this thesis project was made is presented (Section 1.1 to 1.3).

1.1 Expressing computation

At the very first lecture, the first-year students are subjected to the following mantra:

> *A programming language is a notation for expressing computation.*

They are then explained the meanings of "notation" and of "computation", and by the end of the first lecture, they have been told that:

> A program is something written according to this notation for expressing computation. It is executed by a computer to carry out a computation. A computation is some kind of processing operation over some data. Data are representations of information. [A1]

The students are immediately made aware of the computer-science analogue of the tower of Babel: programming languages not only reflect various forms of computation, e.g., imperative, functional, logical or object-oriented; they also reflect the preferences of their designers and the needs of their users, i.e., the programmers. Their common point is that they are all used for expressing computation.

Computation can be expressed in many ways. For example, a programming-language designer might enable direct access to the underlying architecture of the computer, allowing the programmer to access, e.g., its memory. Other programming languages have completely abstracted away the underlying architecture of the computer, enabling the programmer to focus on the problem to be solved, instead of how to process domain-specific representations.

Some programming languages are designed with *type systems* to state which kind of values a variable or reference can hold or refer to. A type system is made to improve the correctness of a program, e.g., to avoid using arithmetic operators on a string value. Type systems can take many forms:

- static type annotations in the source code,
- types that are automatically inferred by the compiler, or
- dynamic types that are not known in advance, but depend on the values assigned to variables at run time.

Each programming language has its own syntax, semantics, type system, run-time environment and memory management. Some languages are made for solving problems within a specific domain, while others are more generic. But they all have something in common: they are all are used to express programs for processing data, i.e., representation of information, and they are all used as a notation for expressing computation.

1.2 Teaching programming languages in 7 weeks

Teaching programming languages in a 7-weeks course to first-year students is not straightforward because of the tower of Babel of programming languages. No matter which programming language the students are to be taught, they need to learn and remember many new terminologies, keywords, and constructs before they can write their first program. Turing-recipient Peter Naur was quoted for saying: "*Syntax is the first thing one forgets about a programming language*". In the programming-languages course, the radical choice was made not to introduce many different programming languages, but to use one single programming language with a minimal syntax: Scheme. In particular, the course considers a well-defined subset of Scheme (Appendix 7.1) in order to introduce as little syntax for the students as possible. Besides having a minimal syntax, Scheme is special because the source code of a Scheme program is represented by nested Scheme lists, making it simple to represent a Scheme program as Scheme data. For example, the very first exercise the students are asked to do in Week 1, is to run a tower of self-interpreters written in Scheme, for a subset of Scheme to measure the cost of interpretive overhead.

In the course, the students are introduced to context-free grammars expressed in Backus-Naur Form (BNF). In general, the course defines the syntax of a programming language with a BNF. It starts off from very small BNFs and simple examples and progressively builds up more advanced BNFs and examples. As a proof of concept, in Week 4, the students are asked to implement a self-applicable syntax checker for the subset of Scheme used for the self-interpreter of Week 1.

1.3 Equalizing the students

The students entering the course all have different backgrounds. Some have been writing programs for many years already, while others have next to zero

experience with programming. Because next to none of the students have prior experience with Scheme, they are put on the same footing when they enter the course.

Some students might already be familiar with a few of the mainstream editors that exist for software development when they enter the programming-language course, but again no students are favored at the expense of others. As an effort to equalize the students, another radical choice was made: the students are not introduced to a familiar or mainstream editor; instead, they are introduced to Emacs.

As reviewed in Chapter 5, a range of stand-alone systems have been developed over the years to facilitate the teaching and learning of programming-languages for novice programmers. These systems offer a wide variety of tools and help for novice programmers when they are to write their first programs. However, many of these systems are typically only used in a learning/teaching setting, which compounds Peter Naur's statement: why spend time on the syntax of a programming-language that one will forget as the first thing, and why spend time on throw-away tools?

The radical choice of introducing Emacs to the students is made to support the pedagogical goals of the course: to equalize the students by not favoring anyone in particular. As an extra benefit, the students become familiar with an advanced text editor, that supports a huge range of programming languages out of the box, and can be extended with custom-made add-ons, as demonstrated in this thesis project.

Put together, Scheme and Emacs form a common ground for the students: they are all faced with the same challenge on the same footing within the same time line.

1.4 The contribution of this thesis

The challenge for the students following the programming-languages course at Aarhus University is to write programs that are:

1. syntactically correct with respect to a BNF and
2. semantically correct as well.

The focus in this thesis project is the syntactic correctness of programs through editorial support. The error support in Petite Chez Scheme consists of tracing facilities and an interactive debugger along with simple error messages printed in the Read-Eval-Print loop (REPL). These error messages mention terms that are unfamiliar to the students, which causes confusion about what is causing the error. Furthermore, programs that are syntactically incorrect with respect to a predefined subset of Scheme, are not always caught by the Petite Chez Scheme interpreter, and these syntactically incorrect programs provoke weird errors, that are reported literally, and hence is incomprehensible for the students, which is counter-productive.

The editorial support system developed in this thesis project handles syntax errors by considering a user-defined grammar in BNF, and decides whether the

Input	Grammar

`(node (leaf)`
`      (node (leaf 2)`
`            (leaf 3)))`
	⟨root⟩ ::= ⟨tree⟩
	
	⟨tree⟩ ::= (node ⟨tree⟩ ⟨tree⟩)
	          | (leaf <number>)

Figure 1.1: A grammar for binary trees and an example of an invalid input.

Input

```
(seq
 (disj
  (atom 1)
  (star (any)))
  (seq
   (plus (atom))
   (empty)))
```

Grammar

```
⟨root⟩   ::= ⟨regexp⟩
⟨regexp⟩ ::= (atom <integer>)
           | (any)
           | (seq ⟨regexp⟩ ⟨regexp⟩)
           | (var ⟨name⟩)
           | (disj ⟨regexp⟩ ⟨regexp⟩)
           | (star ⟨regexp⟩)
           | (plus ⟨regexp⟩)
           | (empty)
 ⟨name⟩  ::= x | y | z
```

Figure 1.2: A grammar for regular expressions and an example of an invalid input.

code in the current buffer conforms to that grammar. Because the system is based on a user-provided grammar in BNF, the system has been named: the *bnf-mode* .

The `bnf-mode` is developed primarily to improve the error feedback in Scheme. It is, however, generic. Given any context-free grammar of fully-parenthesized expressions, such as binary trees (Figure 1.1), regular expressions (Figure 1.2), boolean expressions (Figure 1.3) or even a subset of a programming-language such as Scheme (Figure 1.4), it will provide an interactive environment in which one can conveniently check if what is written in the buffer conforms to a given BNF.

The `bnf-mode` is built as an optional syntax checker, that the user can apply at any time, i.e., before sending the Scheme code to the Petite Chez Scheme interpreter, or to check if some subset of the written syntax conforms to the specified BNF. The goal of the `bnf-mode` is to help novice students to better find and understand syntax errors in their Scheme code, by highlighting syntax errors directly in the source code, but also by pointing at the offended grammatical rule in the BNF-buffer, giving them a hint to what kind of syntax might be missing, is incorrect, or is superfluous. The students using the `bnf-mode` with the given subset of Scheme from the course will receive syntax errors when they use syntax which is not specified in that subset of Scheme. Receiving precise syntax errors feedback will help the students to better recover from syntax errors, and working with a minimal syntax of Scheme will prevent them from introducing many of the typical novice errors, as discussed in Section 2.2.1.

The `bnf-mode` is meant to help the students to better understand the correspondence between the syntax of programs and the grammar defining this

Input

```
(conj
  (neg (var a))
  (disj(neg (var c))))
```

Grammar

```
⟨root⟩   ::= ⟨b-exp⟩
⟨b-exp⟩  ::= (var ⟨name⟩)
           | (conj ⟨b-exp⟩ ⟨b-exp⟩)
           | (disj ⟨b-exp⟩ <b-exp>)
           | (neg ⟨b-exp⟩)
⟨name⟩   ::= a | b | c | d | e | f
```

Figure 1.3: A grammar for boolean expressions and an example of an invalid input.

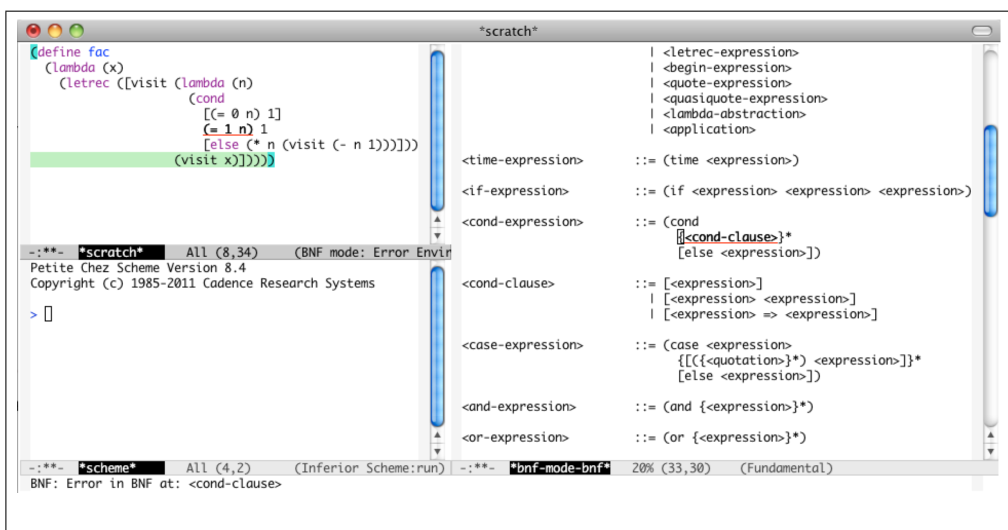

Figure 1.4: The `bnf-mode` when used with a grammar for a subset of Scheme. The `cond`-case is missing its square brackets. The left most occurrence of invalid syntax is highlighted as the point of the error, and the violated grammatical rule is the reference to `<cond-clause>`. It found `(= 1 n)` in the input, but expected something that matches one of the choices in the production `<cond-clause>`. Note the underlining in both the input buffer containing the Scheme code and the grammar buffer, where the grammatical violation is underlined.

syntax.

To improve the error feedback of syntax errors, the aims of the `bnf-mode` is to point directly at the location of a syntax error in the source code, which makes it much easier for the students to locate and correct the error. For example, the `bnf-mode` shows the users where they might be missing a parenthesis, where they are missing an else-branch of an `if`-statement, or it lets them know if their code is syntactically correct. As reviewed in Chapter 4, the `bnf-mode` was evaluated by students following the programming-languages course in the spring of 2013. The students used the `bnf-mode` to check that their functions were syntactically correct, but they also used it for debugging purposes. Many of the students found syntax errors in their Scheme code, because they were using language constructs that was not in the restricted subset of Scheme.

1.5 Overview

The grammar-based syntax checker is presented in Chapter 2. Some of the interesting implementation specific details of the `bnf-mode` are presented in chapter Chapter 3. User evaluations are presented in Chapter 4. Related work is presented in Chapter 5, where various approaches for teaching programming languages are reviewed along with some editorial-support systems, some of which are built for teaching purposes.

1.6 Prerequisites

The reader is expected to be familiar with the following terms: Backus-Naur-Form (BNF); continuation-passing style (CPS); defunctionalization; trampoline transformation; thunks for delaying computation; closures (deep and flat); abstract syntax trees (AST); and Emacs Lisp (Elisp).

Chapter 2

Editorial support through BNF-based syntax checking

A system that assists a user in the process of composing program code or in some way makes it convenient to write programs, or avoid introduction of errors, is referred to as an *editorial support system*. This chapter presents an editorial support system for assisting first-year students when writing Scheme code in Emacs. The system is built as a syntax checker for Scheme in Emacs, but it is generic. The system is generic because it takes any fully-parenthesised grammar in BNF and performs syntax checking based on that grammar. Syntax checking is performed on *input syntax* written by the user in some language defined by a grammar in BNF, which is provided by the user. If a grammar for a subset of Scheme is inserted, the system works as a syntax checker for that subset of Scheme. Syntax errors are handled by pointing directly at grammatical violations, both in the input syntax and in the provided grammar.

For first-year students the syntax of Scheme is initially alien partly because:

- the syntax is the first thing that the students forget about Scheme,
- but also because they do not understand the semantic of the syntax.

To make the syntax less intimidating, the grammar that the students are familiarized with is restricted to a subset of Scheme. By reducing the grammar for Scheme, there is also less syntax which the students have to remember and understand. When the students are subjected to a minimal syntax, they are more likely to remember and understand it, which results in less errors and less frustration. This is further discussed in Section 2.2, but first an overview of the chapter is presented (Section 2.1).

2.1 Overview

Section 2.2 presents some of Schemes language-constructs that might seem incomprehensible for novice programmers. Section 2.3, presents two evaluation strategies for evaluating input syntax against a grammar while also introducing a number of constraints on the grammars accepted by the `bnf-mode` for utilizing precise error location. Once the grammatical constraints has been introduced,

the accepted grammars for the `bnf-mode` are presented in Section 2.4. When the accepted grammars has been outlined, the precise grammatical checks performed by the `bnf-mode` is described in Section 2.5. When the grammatical evaluation has been defined, the editorial environment is described and presented in Section 2.6.

2.2 Avoid advanced syntax for novice students

Restricting the grammar of a programming language can prevent a novice programmer from using advanced language-constructs, which the programmer does not understand. Avoiding advanced language-constructs will in return cause less errors to be introduced, which again causes less frustration for the programmer. Less frustrations for the programmer will increase the feeling of success. In this section some general language-constructs are presented, that could easily lead a novice programmer to introduce errors, and thus should be avoided.

2.2.1 Avoiding novice errors by restricting the programming language grammar

Scheme defines several language-constructs where a sequence of expressions are accepted, but only the result of the last expression is returned. The `cond`-construct is an example of this. Consider the expression:

```
(cond [\#t (+ 1 2) (* 3 4) 5])
```

In this case, the two sub-expressions `(+ 1 2)` and `(* 3 4)` are evaluated, but their respective results are ignored, and only the literal value 5 is returned. For a first-year student this behaviour is not obvious. Other examples of Scheme-constructs implementing this behaviour are `let`, `letrec` and `lambda`, etc. Another consequence of these kind of language-constructs is illustrated in Figure 2.1, where a valid Scheme function is defined. Instead of calling the `inner`-function, it returns `x` directly. This behaviour is even more incomprehensible for the first-year students. It is valid Scheme syntax, because the body of a `letrec` accepts multiple expressions, but only the result of the latter is returned. Using the subset of Scheme with the `bnf-mode`, a `letrec` with multiple expressions in the body will not be accepted as valid syntax, because the `letrec` defined in the subset of Scheme only accepts one expression in the body of a `letrec`.

In general, the subset of Scheme avoids constructs that evaluates several expressions and only returns the value of the last one. These kinds of constructs seems to have only one purpose: to impose side-effects, e.g., printing. The subset of Scheme only defines one construct where several expressions are accepted, but only the result of the latter is returned, namely `begin`, which is typically used for printing. Writing Scheme code like this, makes side-effects explicit and not accidental. Novice students are much less likely to use a construct such as `begin`, unless they really intend to introduce a side-effect such as a printout.

In Scheme it is possible to redefine built-in functions such as `if`, `quote` or `define`, etc. Redefining built-in functions by mistake will cause incomprehen-

```
(define (foo x)
  (letrec ([inner (lambda (n)
                            (+ 1 n))])
          inner x))
```

Figure 2.1: Example of a very likely error. The `letrec` in Scheme accepts several expressions to be evaluated and only the result of the last expression is returned. In this case the application of the inner function was left out, resulting in x being returned.

```
> (cond
       [(if #f 1) 0]
       [else 1])
0
```

Figure 2.2: Evaluating an `if`-statement in Scheme without an `else`-branch and with the condition hard-coded to #f, results in `void`, which is interpreted as true in Scheme in the context of a boolean-value, because everything that is not explicit false is interpreted as true.

sible error messages to be reported, which is why it should be avoided.

2.2.2 Complete function definitions

A function with the signature: `int -> boolean`, is expected to always return a boolean value no matter which input is given. However, in Scheme a function is allowed to return any value. A novice programmer should be able to look at a function definition and reason that the type of the function is always the same no matter what the input value is.

However reasoning about the output type of a Scheme function is not always easy, because some of the built-in Scheme-constructs are hard to comprehend. For example, the Scheme standard accepts the definition of an *if*-statement without an *else*-branch. If the *else*-branch is left out, and the *condition* is false, it is not clear what the result should be. In fact the result is *void*, which in Scheme might be interpreted as true, if the expression is evaluated in a context where a boolean-value is expected, as illustrated in Figure 2.2. Defining a function that returns a `boolean`-value for some inputs and `void` for other inputs is easily done by mistake.

By requiring the programmer to always provide an *else*-branch, it is explicit what is returned when the *condition* of an *if*-statement is false. Likewise an *else*-branch should always be used in a *cond*-expression.

As illustrated here, there are many reasons why novice programmers should only be exposed to well-defined language-constructs. In order to avoid the advanced language-constructs a grammar excluding these constructs should be used in conjunction with the `bnf-mode`. The `bnf-mode` will be able to point out any syntax that is not valid according to the given BNF and even check if BNF-defined constants are being redefined. The next section (Section 2.3)

presents how the `bnf-mode` points out syntax errors.

2.3 Error precision: Grammatical evaluation and constraints

In order to provide precise error syntax errors, it is vital to point exactly at the location of invalid input syntax and at the grammatical rule being violated. In this section, the techniques for precise syntax error feedback are presented.

Before presenting the grammars supported by the `bnf-mode` (Section 2.4), two distinct assumptions about input syntax are discussed and presented in Section 2.3.1. These two assumptions about the input syntax lead to a discussion about how to evaluate the input syntax (Section 2.3.2), and based on the discussion about the evaluation of input syntax, some grammatical constraints are introduced in Section 2.3.3.

2.3.1 Evaluating input syntax against a BNF

Two approaches for evaluating input syntax against a BNF are presented in this section. They differ in their assumptions about the input syntax. The two approaches expect the input syntax to be either:

(A) plain-text without any restrictions, or
(B) fully-parenthesized terms, namely a tree structure, i.e., S-expressions.

An example of (A) could be a text-string of English:

```
I like carbonated water.
```

which is accepted by the grammar of non-parenthesized expressions:

```
⟨sentence⟩   ::= ⟨subject⟩ ⟨verb⟩ ⟨complement⟩.

⟨subject⟩    ::= I
               | You
⟨verb⟩       ::= like
               | dislike
⟨complement⟩ ::= water
               | carbonated water
```

An example of (B) could be an S-expression of binary trees:

```
(node (leaf 1)
      (node (leaf 2)
            (leaf 3)))
```

which is accepted by the grammar of fully-parenthesized expressions for binary trees:

```
⟨tree⟩  ::= (node ⟨tree⟩ ⟨tree⟩)
          | (leaf ⟨number⟩)
```

The two approaches shall be referred to as (A): *the plain-text evaluation approach* and (B): *the S-expression evaluation approach* and they are further presented in the following section (Section 2.3.2).

Input Grammar

```
(a (b 1) c)     ⟨root⟩ ::= (a ⟨b⟩ ⟨c⟩)
                ⟨b⟩    ::= (b ⟨number⟩)
                ⟨c⟩    ::= c
```

- (a (b 1) c) is matched against the <root> production.
- (b 1) c is matched against the reference to <b>, namely the production: (b <number>).
- 1 is matched against the reference to <number>.
- The input syntax that was not matched by the grammatical reference to <b>, namely c, is matched against the reference to <c>.

Figure 2.3: The plain-text evaluation considers the current grammatical rule and the head of remaining input.

2.3.2 The plain-text evaluation vs. the S-expression evaluation

Before presenting the two evaluation approaches, some of the grammatical terms are introduced briefly:

- A grammatical-construct is a constant (including numbers), parenthesis or grammatical-reference, i.e., abc, 42, (abc <a>) or <a>.
- A grammatical-production is a : <root> :: = (abc <a>).
- A grammatical-reference: <a>.
- A choice between several grammatical-constructs: <root> ::= 1 | 2.

The grammatical components are presented further in Section 3.5. A grammatical-reference is "followed" by looking up its grammatical definition in the grammar, which is presented further in Section 2.5.4.

The plain-text evaluation approach searches through the grammar, following grammatical references, left to right, while carrying the entire remaining input syntax along, matching as much of the input as possible against the grammatical rules encountered in the recursive descent. The search terminates either when a syntax error is found in the input syntax or when all of the input syntax has been successfully matched, as illustrated in Figure 2.3. A syntax error is found in the evaluation when the current input syntax does not conform to the current grammatical-construct.

The advantage of the plain-text evaluation approach is that the input does not have to be associated in any way with the grammatical constructs, which makes it possible to evaluate any grammar successfully, as illustrated in Figure 2.3. The disadvantage of the plain-text evaluation approach is that remaining input is transferred from one grammatical-construct to be matched in the next grammatical-construct, as illustrated in Figure 2.4, where the input (b 1 d) contains a superfluous d. In the plain-text evaluation, this extra d will be transferred to be matched against the reference to <c>. This results in an error pointing at the grammatical reference to <c>, stating that the input d did not

Input	Grammar
`(a (b 1 d) c)`	⟨root⟩ ::= (a ⟨b⟩ ⟨c⟩) ⟨b⟩ ::= (b ⟨number⟩) ⟨c⟩ ::= c

- `(a ...)` is matched against the `<root>` production.
- `(b 1 d) c` is matched against the reference to `<b>`.
- `1` is matched against the reference to `<number>`.
- The input that was not matched by `<b>`, namely `d c`, is matched against the reference to `<c>`.
- An error is reported `<c>`, stating that `d` did not match the grammar for `<c>`.

Figure 2.4: The plain-text evaluation in unable to provide precise error feedback because it does not associate input with the grammatical-constructs, but rather tries to match input in a greedy fashion while transferring unmatched input from one grammatical construct to the next.

match the grammar for `<c>`, which is misleading. The error that should be reported, is the superfluous `d` in the input syntax, and the grammatical definition for `<b>`, namely: `(b <number>)`.

The S-expression evaluation approach uses a divide-and-conquer strategy, where the input syntax is directly associated with the corresponding parts of a given grammatical production, the first syntactic construct is matched with the first grammatical construct, the second with the second and so on, as outlined in Figure 2.5.

The S-expression evaluation has an increased error precision as compared with the plain-text evaluation, because it associates input syntax with specific grammatical constructs. Since the input syntax has to be associated with the grammatical rules in a well-defined way, the S-expression evaluation assumes that the input syntax is S-expressions and thus the BNF must describe fully-parenthesized terms, which is further discussed in Section 2.3.3.

In this thesis project, the S-expression evaluation approach is used because it provides the most precise error feedback of the two evaluation techniques, when it is combined with the grammatical constraints presented in Section 2.3.3

2.3.3 Imposed constraints for improved error feedback

To increase the error feedback capabilities of the `bnf-mode`, two constraints has been imposed on the accepted grammars: grammatical-constants are reserved in order to avoid redefinitions of built-in language syntax, and each grammatical production defining more than one construct must be enclosed by parenthesis.

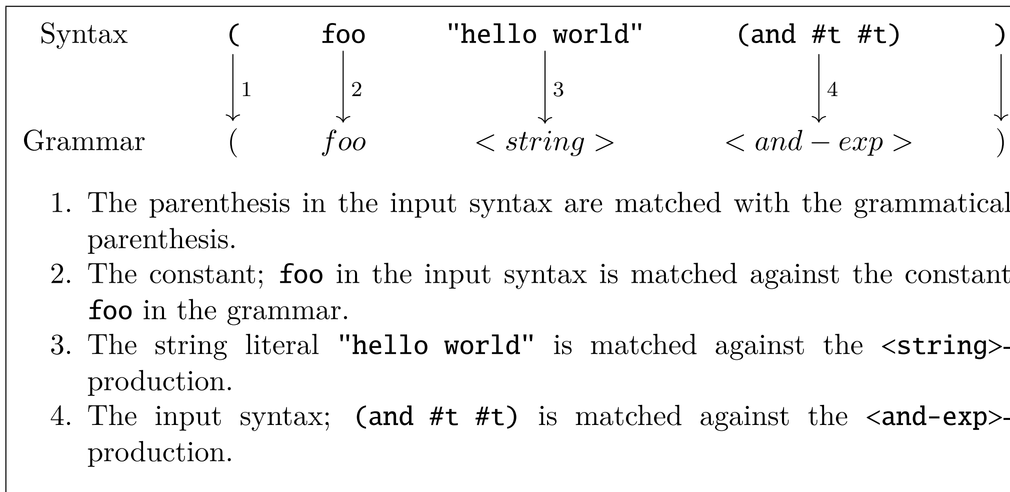

Figure 2.5: The S-expression evaluation approach considers the structure of each grammatical construct and matches the input syntax (on top) with the grammatical components (in the bottom).

Grammatical constants reserved

Scheme allows any built-in construct to be redefined, but as discussed in Section 2.2.1, redefining built-in language constructs should be avoided.

As seen in the subset of Scheme (Section 7.1), the `define`-construct uses the built-in `<variable>`-production for naming functions. Thus to avoid redefinitions of built-in constructs, it was decided that constants defined in the BNF, such as `define`, `if`, `quote`, etc. in the subset of Scheme, are not accepted within the context of a `<variable>`. By disallowing BNF-constants in a `<variable>`-context it is impossible to redefine built-in language constructs by mistake, such as:

```
(define if (lambda (x) x))
```

This is completely generic and applies for any constant in any user-defined grammar, when using the built-in `<variable>`-production. A constant is any sequence of characters inside a grammatical production (except for references and parenthesis).

Imposing parenthesis constraint

As discussed in Section 2.3.2, the S-expression evaluation approach is chosen because of its increased error precision, but it requires the input syntax to be grouped in a well-defined way. Grouping the input syntax is achieved by restricting the accepted BNFs such that a grammatical production defining more than one grammatical construct must be enclosed by parenthesis, i.e., these two productions are accepted:

- `<root> ::= (a b)`
- `<root> ::= <boolean>`

while these two are not:

```
BNF

⟨root⟩ ::=  ⟨a⟩ ⟨b⟩ ⟨c⟩
⟨a⟩         ::=  a
⟨b⟩         ::=  b | b c
⟨c⟩         ::=  c | d

⟨root⟩ :  a    b   c   d
⟨root⟩ : ⟨a⟩   b   c   d     ;; ‘‘a’’ matches ⟨a⟩
⟨root⟩ : ⟨a⟩  ⟨b⟩  c   d     ;; ‘‘b’’ matches ⟨b⟩
⟨root⟩ : ⟨a⟩  ⟨b⟩  ⟨c⟩ d     ;; ‘‘c’’ matches ⟨c⟩
Error: unmatched input ‘‘d’’.
```

Figure 2.6: The S-expression evaluation approach matches the input syntax on the grammatical constructs, at the top-level, in the <root>-production, the input is matched on the grammatical references, such that: <a> <b> <c> is matched with respectively `a`, `b` and `c`. The constant `d` remains unmatched, thus resulting in the unmatched-input error.

- `<root> ::= a b`
- `<root> ::= (foo <bar>) (baz <number>)`

If the input syntax is not grouped using parenthesis, the S-expression evaluation approach is unable to correctly associate the input with the grammatical constructs as illustrated in Figure 2.6.

In Figure 2.6 the evaluation of the input syntax results in an error, because the syntax is separated by the white space and associated with the grammatical-constructs, such that: `a b c` is associated with the references to respectively `<a>`, `<b>` and `<c>`, but last `d` remains unmatched.

By wrapping parenthesis around the input syntax, it is possible to properly associate the input syntax with the grammatical constructs, as illustrated in Figure 2.7.

Requiring parenthesis in the grammar (and thus also in the input syntax) is a simple way to solve the problem of associating input syntax to the grammatical rules. Requiring that the input is fully-parenthesized imposes a constraint on the context-free languages accepted by the `bnf-mode`, but since Scheme is fully-parenthesized this is not a problem, it only limits a bit of the generic ability of the `bnf-mode`.

Using fully-parenthesized input syntax makes the precision of the syntax errors in the input and in the grammar precise.

2.4 Grammars supported by the `bnf-mode`

The grammars supported by the `bnf-mode` are context-free grammars in BNF, describing fully-parenthesized languages, as previously discussed in Section 2.3.3. The accepted BNF-grammars are extended with two extra notations: Kleene star, `*`, and plus, `+`, to facilitate grammatical expressiveness. Star “`*`” and plus “`+`” are referred to as *multipliers*. A constant followed by multiplier is inter-

```
BNF

⟨root⟩ ::=  (⟨a⟩ ⟨b⟩ ⟨c⟩)
⟨a⟩        ::=  a
⟨b⟩        ::=  b | (b c)
⟨c⟩        ::=  c | d
```

Example 1 :Valid input syntax:

```
⟨root⟩ : (a (b c) d)
⟨root⟩ : (⟨a⟩ (b c) d)  ;; ‘‘a’’ matches ⟨a⟩
⟨root⟩ : (⟨a⟩  ⟨b⟩  d)  ;; ‘‘(b c)’’ matches ⟨b⟩
⟨root⟩ : (⟨a⟩  ⟨b⟩  ⟨c⟩)  ;; ‘‘d’’ matches ⟨c⟩
The input was matched by the grammar.
```

Example 2: introducing an extra “a” in the beginning:

```
⟨toplevel⟩ : (a  a (b c) d)
⟨toplevel⟩ : (⟨a⟩  a (b c) d)
Error: missing parenthesis!
```

Figure 2.7: A fully-parenthesized BNF-grammar with a valid input (Example 1) and an invalid input (Example 2). The invalid input is the same as the valid, except that it contains an extra “a” in the beginning. The error is reported at the occurrence of the second “a”.

preted as a constant and not as a multiplier of that constant. An accepted grammar only allows a single multiplier inside a parenthesis. A Multiplier should be used with a curly brace to explicitly state its binding to the grammatical constructs, however multipliers are accepted after grammatical-references and parenthesis.

A set of built-in productions for easing the grammatical definitions are provided by the `bnf-mode` (Section 2.4.2). Comments are not explicitly mentioned in the language for the accepted grammars, but it is allowed anywhere in the grammatical definitions by using “`;`”, which comments out the following content until the end of the line.

The grammars supported by the `bnf-mode` are specified in Figure 2.8. White space is accepted anywhere, except inside production definitions and their references, i.e., `<ro ot>` is not a valid production name. Anything between two double quotes is interpreted as a string literal. Currently there is no escape character for BNF meta characters, such as curly braces, thus it is currently impossible to define a grammar describing a syntax that contains any meta character such as curly braces.

2.4.1 Grammatical root

A grammar has an entry point from which any string accepted by the grammar can be built. The entry point is referred to as the *root* of the grammar. In the `bnf-mode`, the root-production is assumed to be the first production from the top in the grammatical definition (The first production does not have to be named `<root>` in order for the syntax checking to work).

```
⟨root⟩                  ::= ⟨toplevel-syntax⟩
⟨toplevel-syntax⟩       ::= {⟨group-syntax⟩}+
⟨group-syntax⟩          ::= ⟨left-hand-syntax⟩ "::=" ⟨members⟩
⟨left-hand-syntax⟩      ::= {⟨white-space⟩}* ⟨reference⟩ {⟨white-space⟩}*

⟨members⟩               ::= {⟨white-space⟩}* ⟨syntax⟩ {⟨white-space⟩}*
                          | {⟨white-space⟩}* ⟨syntax⟩
                            {⟨white-space⟩}* "|" ⟨members⟩

⟨syntax⟩                ::= ⟨atomic⟩ | ⟨parenthesized-term⟩
⟨atomic⟩                ::= ⟨built-in⟩ | ⟨constant⟩ | ⟨reference⟩
⟨parenthesized-term⟩    ::= ⟨parenthesized⟩ | ⟨meta-parenthesis⟩

⟨parenthesized⟩         ::= "(" ⟨sequence-multiplier⟩ ")"
                          | "[" ⟨sequence-multiplier⟩ "]"
⟨meta-parenthesis⟩      ::= "{" ⟨sequence-multiplier⟩ "}"

⟨sequence-no-multiplier⟩::= ⟨constant-sequence⟩
                          | ⟨reference⟩
                          | ⟨parenthesized-term⟩
                          | ⟨constant-sequence⟩ ⟨reference⟩ {⟨white-space⟩}*
                                                ⟨sequence-no-multiplier⟩
                          | ⟨constant-sequence⟩ ⟨parenthesized-term⟩
                                               {⟨white-space⟩}* ⟨sequence-no-multiplier⟩
                          | ⟨constant-sequence⟩ {⟨white-space⟩}+ ⟨sequence-no-multiplier⟩
                          | ⟨reference⟩ {⟨white-space⟩}* ⟨sequence-no-multiplier⟩
                          | ⟨parenthesized-term⟩ {⟨white-space⟩}*
                                                 ⟨sequence-no-multiplier⟩

⟨sequence-multiplier⟩   ::= ⟨sequence-no-multiplier⟩ ⟨opt-multiplier⟩
                                                     ⟨sequence-no-multiplier⟩
                          | ⟨sequence-no-multiplier⟩

⟨constant-sequence⟩     ::= ⟨constant⟩
                          | ⟨constant⟩ {⟨white-space⟩}+ ⟨constant-sequence⟩

⟨multiplier⟩            ::= ⟨metha-parenthesis⟩"*"
                          | ⟨metha-parenthesis⟩"+"
                          | ⟨parenthesised-term⟩"*"
                          | ⟨parenthesised-term⟩"+"
                          | ⟨reference⟩"*"
                          | ⟨reference⟩"+"
⟨built-in⟩              ::= "⟨boolean⟩" | "⟨integer⟩" | "⟨number⟩"
                          | "⟨natural-number⟩" | "⟨character⟩"
                          | "⟨string⟩" | "⟨symbol⟩" | "⟨identifier⟩"
                          | "⟨nothing⟩" | "⟨variable⟩"
⟨white-space⟩           ::= ⟨space⟩
                          | ⟨tab⟩
                          | ⟨new-line⟩
⟨reference⟩             ::= "⟨" ⟨constant⟩ "⟩"
⟨constant⟩              ::= {⟨char⟩}*
⟨char⟩                  ::= a | ... | Z | 0 | ... | 9 | - | ... | ‘
```

Figure 2.8: A context free grammar that specifies the grammars supported by the `bnf-mode`.

```
⟨boolean⟩          ::= #t | #f
⟨integer⟩          ::= -536870912 to 536870911
⟨number⟩           ::= -536870912 to 536870911
⟨natural-number⟩   ::= 0  to 536870911
⟨character⟩        ::= \#[0-9 | a-z | A-Z |
                       ... ()[]{}@^#] | \#space
⟨string⟩           ::= "[0-9a-zA-Z()[]{}@^#]*"
⟨identifier⟩       ::= [0-9a-zA-Z()[]{}@^#]+
⟨nothing⟩          ::= ;; Matches the empty input
⟨variable⟩         ::= ⟨identifier⟩
                       ;;;  Without constants defined
                       ;;;  in the grammar.
⟨symbol⟩           ::= ⟨identifier⟩
```

Figure 2.9: Built-in productions. The <boolean> and <character> productions accepts the Scheme-values for boolean and character.

2.4.2 Built-in productions

A number of built-in productions are provided to make the definition of a BNF-grammar easier. The built-in productions are outlined in Figure 2.9. A built-in production is overwritten if it is explicitly specified in a user-defined BNF.

The `<boolean>`-production is defined to be the Scheme boolean, but this can be redefined by a custom definition in a BNF. The `<character>`-production is also defined to be the Scheme notation for characters. The production `<variable>` accepts any sequence of characters except for constants defined in the grammar, as described in Section 2.3.3. The `<string>`-production matches any string literals, which is defined to be any characters including white space starting and ending with a double quote. The `<nothing>`-production matches the empty input. The `<nothing>`-production should only be used to specify that the empty input is accepted, it should not be used to specify an option.

2.5 Grammatical validation of input syntax

If invalid input syntax is evaluated against a grammar, the result is an error containing information about which grammatical construct was violated and where in the input syntax the error was found.

To provide precise error feedback the `bnf-mode` must be able to:

1. Perform a series of simple grammatical checks on the input as outlined in Section 2.5.1.
2. Report grammatical violations at an intuitive level in the BNF structure, such that the highlighted position make sense for the end-user as outlined in Section 2.5.2.
3. Pick the best *partial-match*, when several possible partial-matches of the input syntax are available, as outlined in Section 2.5.3.
4. Resolve references in the BNF, as outlined in Section 2.5.4.

Input	Grammar
`(a)`	`⟨root⟩ ::= (a b)`

Figure 2.10: Missing syntax error. The `bnf-mode` points to the outer parenthesis in the input syntax, where some syntax was expected. The violated grammatical rule is pointed out as well.

5. Evaluate Kleene Star and plus in an optimal way, such that syntax errors are pointed out exactly in both the input syntax and in the BNF, as outlined in Section 2.5.5.

2.5.1 General grammatical checks

Due to the generic nature of the grammar checker, there are only a few general constraints that has to be checked.

Basic checks

The check for parenthesis consists of the following checks:

- Basic parenthesis check: check if the input is a parenthesis.
- Parenthesis type check: check that the input parenthesis type (i.e. `[ ... ]` or `( ... )`) is the same as the parenthesis type specified in the grammar.
- Parenthesis type match: check that the open parenthesis type matches the closing parenthesis type.
- Meta parenthesis ignored: if the grammar specifies a meta parenthesis (curly brace), ignore the parenthesis check of the input.

The check for constants simply checks if a given syntactic constant matches the expected constant in the grammar. References are checked by a look up followed by an evaluation of the input against the result of the look up. This is further described in Section 2.5.4.

Missing syntax

When a syntax error is reported due to missing syntax, the analysis is unable to point to something that is not present, but will instead underline the nearest enclosing parenthesis and the missing grammatical construct in the BNF, as outlined in Figure 2.10.

Too much syntax

When too much syntax is provided, the superfluous input syntax and the grammatical production violated (with too much syntax) will be underlined, as outlined in Figure 2.11.

```
Input            Grammar

(a b c)          ⟨root⟩ ::= (a b)
```

Figure 2.11: Too much syntax error. The `bnf-mode` points to the superfluous syntax in the input. The violated grammatical rule is pointed out as well.

```
Input            Grammar

(cba)            ⟨root⟩       ::= <expression>
                 ⟨expression⟩ ::= ⟨number⟩
                                | ⟨boolean⟩
                                | ⟨character⟩
                                | ⟨string⟩
                                | (abc d)
```

Figure 2.12: The input `(cba)` does not partly match any of the productions of the <expression>, thus the error should be reported one level up, namely at the reference to <expression>.

2.5.2 BNF error report level

When a syntax error is found, the grammatical rule being violated should be pointed out at a best effort, where it makes the most sense for the user.

A grammatical production can define a set of *choices* (separated by `|`) between several distinct productions, as illustrated by the `<expression>`-production in Figure 2.12. In Figure 2.12, the invalid input syntax do not match any of the choices defined by the `<expression>`-production, thus the error is reported at the reference to the `<expression>`-production in the `<root>`-production.

In general there are two levels at which a grammatical violation is pointed out:

1. A reference to a grammatical production, as illustrated in Figure 2.12,
2. a choice inside a grammatical production, as illustrated in Figure 2.13.

For the latter case, when deciding which "choice" inside a grammatical production is the best, a metric for comparing matches of the various choices in a production has to be defined. A metric for comparing two such matches is defined next (Section 2.5.3).

2.5.3 Partial-match order

Invalid input syntax is only partly matched against a grammatical production, and the result is a *partial-match*. A partial-match consists of the resulting syntax error, the violated grammatical production and the remaining input syntax.

If a production in the BNF defines a number of choices such as:

```
Input            Grammar

(abc)            ⟨root⟩        ::= ⟨expression⟩
                 ⟨expression⟩  ::= ⟨number⟩
                                 | ⟨boolean⟩
                                 | ⟨character⟩
                                 | ⟨string⟩
                                 | (abc d)
```

Figure 2.13: The input (abc) does not fully match any of the productions of the <expression>. However it does match part of the production (abc d), thus pointing it out a being the violated rule.

```
⟨root⟩ ::= ⟨a⟩ | ⟨b⟩ | ⟨c⟩
```

and invalid input syntax is evaluated against the grammar, none of the choices completely accepts the input syntax. However, one of them might be a *better match* than the other two. Deciding which of the three choices is a better match for the input syntax by order is referred to as partial-match order. The partial-match order is illustrated by the <expression>-production in Figure 2.13, where the invalid syntax (abc) is best matched on the grammatical choice: (abc d), and thus the syntax error points at that BNF location.

A well-defined method for comparing two partial-matches is based on the structure of the syntax:

- Constants,
- parenthesis,
- arity,
- and the combination of these, i.e. [constant and parenthesis was matched].

The assumption is: the more of the structural components that are matched, the more likely the input was meant to match the given grammatical rule. All the possible combinations of two partial matches are enumerated in Table 2.1. The table shows which of two partial matches P_1 and P_2 are selected based on how many of the structural criteria are satisfied. In Table 2.1, "len" indicates that the partial match (P_1 or P_2) with the shortest remaining unmatched syntax is returned. If P_1 and P_2 have the same length of remaining unmatched input syntax, P_1 is returned.

2.5.4 Resolving references

A production is referenced by inlining its name in the BNF. Recursive references are allowed. A reference to a production called <expression> could be written like; (time <expression>). A production does not have to be defined before it is referenced. If a production is referenced without being defined an error will occur, that points at the reference in the BNF that was not found and the input syntax that was suppose to be matched on that reference, as depicted in

$P_2 \backslash P_1$	C	P	A	CP	CA	PA	CPA	none
C	len	P_2	P_2	P_1	P_1	P_2	P_1	P_2
P	P_1	len	len	P_1	P_1	P_1	P_1	P_2
A	P_1	len	len	P_1	P_1	P_1	P_1	P_2
CP	P_2	P_2	P_2	len	P_2	P_2	P_1	P_2
CA	P_2	P_2	P_2	P_1	len	P_2	P_1	P_2
PA	P_1	P_2	P_2	P_1	P_1	len	P_1	P_2
CPA	P_2	P_2	P_2	P_2	P_2	P_2	len	P_2
none	P_1	P_1	P_1	P_1	P_1	P_1	P_1	len

Table 2.1: A table indicating which one of two partial matches are chosen. P1 and P2 represents two partial matches. Len means that the Partial match with the shortest remaining unmatched syntax is returned, if P_1 and P_2 have the same length of unmatched input syntax, P_1 is returned.

Input		Grammar
`Example 1:`	`(abc)`	`⟨a⟩ ::= (abc <b>)`
`Example 2:`	`(abc a)`	`⟨a⟩ ::= (abc <b>)`

Figure 2.14: When an undefined grammatical production is referenced, the location of the reference is underlined in the grammar and in the input syntax.

figure 2.14. A reference is resolved along with some input that is to be matched against the referenced production as illustrated in figure 2.15.

Resolving references are done through the following steps:

1. Check if the reference is defined in the grammar.
2. If the reference is defined in the BNF, retrieve the grammatical definition of the reference and evaluate the input syntax associated with the reference against this definition.
3. If the reference is not defined in the BNF, check if the reference is a built-in production.
4. If the reference is a built-in production, call the built-in function with the associated input syntax.
5. If the referenced production was not found, an error is reported at the location of the input that was to be validated against the reference and at the location of the reference in the grammar.

2.5.5 Kleene star and plus optimized for common case

The Kleene Star operator is introduced as a convenient way to specify "zero or many" of some grammatical pattern. For example:

```
⟨a⟩  ::= {a}*
```

Input	Grammar
`(time 42)`	`⟨time-expression⟩ ::= (time ⟨expression⟩)`

step	syntax	grammar
1)	`time`	`time`
2)	`42`	`<expression>`
3)	`42`	Look up `<expression>`:

```
⟨expression⟩ ::= ⟨number⟩
               | ⟨boolean⟩
               | ⟨character⟩
              ...
               | ⟨application⟩
```

step	syntax	grammar
5)	`42`	Look up `<number>`
6)	`42`	`<number>` is a built-in production.
7)	`42`	Invoke the built-in function: `(bi-is-number? 42) => true`.

Figure 2.15: Resolving references in the grammar.

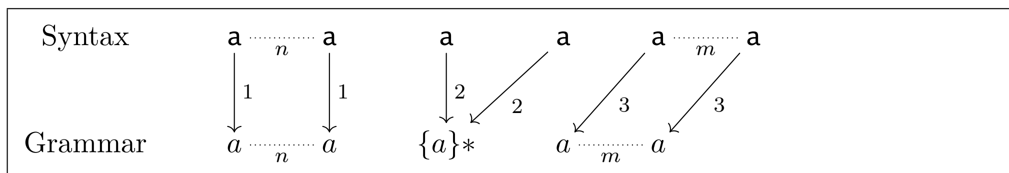

Figure 2.16: Matching input on Kleene star (outer parenthesis are omitted).

means “zero or many a”, which accepts input such as “`a a`” or “`a a a a`”. The plus operator is a shorthand notation to express “one or many”.

The evaluation of Kleene Star or plus is done in an optimal way for the common case. By observing the subset of Scheme, it is clear that nowhere in the grammar two multipliers occur inside the same parenthesis. Because a multiplier is always alone inside a single parenthesis expression, it is possible to have precise error precision for Kleene Star and plus.

The evaluation of Kleene Star and plus is done in 3 steps (as also illustrated in Figure 2.16):

1. Match the first n grammatical constructs on the left side of the multiplier with the first n elements in the input.
2. Ignore the first n and the last m grammatical constructs on both sides of the multiplier, and ignore the first n and last m elements from the input, and evaluate the remaining input on the multiplier.
3. Match the last m grammatical constructs on the right side of the multiplier with the last m elements in the input:

Input syntax 1	Input syntax 2	Grammar 1	Grammar 2
’a	(quote a)	<root> ::= ’a	<root> ::= (quote a)
‘a	(quasiquote a)	<root> ::= ‘a	<root> ::= (quasiquote a)
,a	(unquote a)	<root> ::= ,a	<root> ::= (unquote a)

Table 2.2: The grammars in the column 3 and 4 are equivalent. The syntax in column 1 and 2 matches the grammar in column 3 and column 4.

By evaluating Kleene Star and plus this way, it is possible to precisely determine which grammatical construct is violated if a mismatch between the input and the grammar is found.

2.5.6 Adjustments for Scheme

Three desugaring conversions have been implemented to accommodate Scheme. Scheme does not syntactically distinguish between “`'x`”, “`` `x ``” and “`,x`”, and their desugared counterparts: `(quote x)`, `(quasiquote x)` and `(unquote x)`. It was decided to make the same transformations in the `bnf-mode`, both in the grammar and in the input syntax as outlined in Table 2.2.

These transformations also ensures that the characters are internally represented as their constant counterparts. This again means that any redefinitions of `quote, quasiquote` or `unquote` will not be permitted if the BNF mentions the single character counterparts as discussed in the section about imposed constraints for improved error feedback, Section 2.3.3.

To summarize, a redefinition of, for example `quote`:

```
(define quote 1)
```

is not accepted by the subset of Scheme, even if the provided grammar only mentions the single quote character and not the constant “`quote`”.

2.6 Editorial environment

The editorial environment in Emacs consists of two buffers, one where the input syntax is written (the bnf-mode buffer) and one where the grammar is inserted (the `*bnf-mode-bnf*` buffer) an illustration of this is shown in Figure 2.17.

The *bnf-mode-bnf* buffer is automatically initiated and shown, when the `bnf-mode` is activated (`M-x bnf-mode`). On startup the `bnf-mode` will automatically look for a file in the current working directory postfixed “.bnf”. The content of the “.bnf”-file is loaded into the grammar buffer (*bnf-mode-bnf*), if the buffer is empty. Once a grammar is inserted, the editorial environment is ready for usage.

The `bnf-mode` extends the default scheme-mode and provides one extra key-shortcut for activating the syntax checker, `M-n`. Pressing `M-n`, while having the cursor placed below a definition or expression, makes the syntax-checker analyze the code according to the specified BNF. While processing, the Emacs mini-buffer will show the status: `BNF: Analyzing...`, when the syntax-checking is

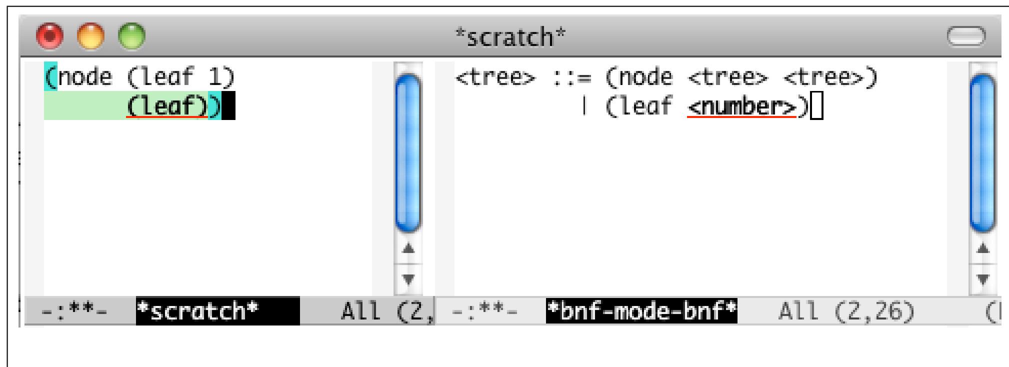

Figure 2.17: The editorial environment in Emacs.

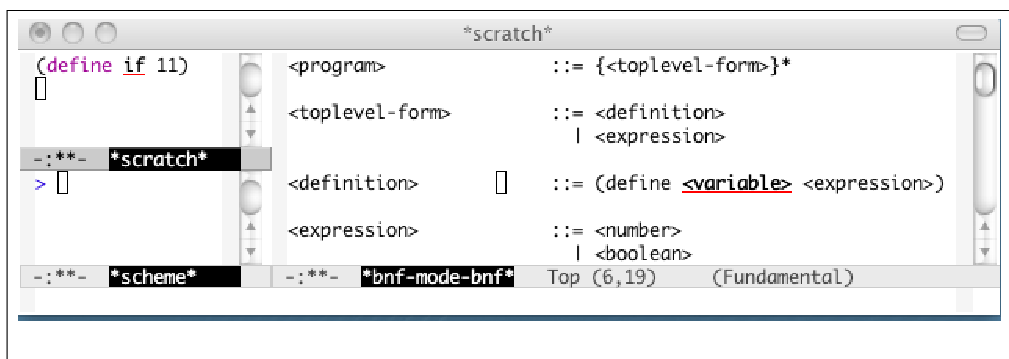

Figure 2.18: Trying to redefine the built-in `if` will result in a syntax error, stating that `if` is not allowed in the place of a variable.

done, the text will either show `BNF: Correct!` or `BNF: Error in BNF at: <grammatical-rule>`. If a syntax error is found, both the invalid syntax and the offended grammar-rule is underlined. Once an error is encountered, the editor is in an *error environment*, where only the part of the code that was analyzed is syntax highlighted, all other code is colored black. To exit the error environment, press `M-n` again.

To install the `bnf-mode`, use the guide in Section 7.2.

2.7 Summary

Section 2.2 presented some examples of which kind of language constructs that will typically cause a novice programmer to introduce errors.

The generic syntax checker accepting fully-parenthesize context-free grammars in BNF was presented. Techniques to ensure precise error feedback was presented and demonstrated by example. The grammar of BNFs accepted by the `bnf-mode` was presented along with a set of built-in grammatical productions. The techniques for pointing at grammatical rules in the user-provided BNF when a syntax error is encountered was presented and demonstrated by example. The handling of references in the grammatical structure in connection with the evaluation of input syntax was explained. A common case optimal evaluation for Kleene star end plus was presented and demonstrated by example.

Desugaring of Scheme syntax was described and its necessity was motivated. The editorial environment of the `bnf-mode` was presented and examples of usage was presented through figures [Figure 2.17 and Figure 2.18].

The next chapter (Chapter 3) presents some of the interesting implementation specific details of the `bnf-mode`.

Chapter 3

Implementation

This chapter presents some of the implementation specific details of the `bnf-mode`. Since the techniques behind the BNF-based syntax checking has already been described in detail in Chapter 2, this chapter presents a discussion of the chosen implementation language (Section 3.1) along with some limitations of the implementation language (Section 3.2). Further a solution for these limitations is presented (Section 3.4) and finally the two parsers are presented in Section 3.5.

3.1 Implementation language

The `bnf-mode` is implemented in Emacs Lisp, known as Elisp [B19]. Elisp was chosen primarily because it is closely bound to Emacs, but also because it is platform independent, and it is convenient for the end user to install an Emacs-mode written entirely in Elisp.

The Elisp environment enables the mode-developer to implement domain specific modes for Emacs, i.e., a mode for syntax highlighting a specific programming language syntax, or extending existing modes for added functionality, which is the case for the `bnf-mode` that appends functionality to the default scheme-mode in Emacs.

To install the `bnf-mode`, the user has to load the file containing the Elisp code definitions for the `bnf-mode` and activate the mode from within Emacs. Typically a mode for Emacs is setup to be automatically loaded (see Section 7.2 for installation).

However, Elisp has some limitations concerning recursive functions which are addressed in the next section (Section 3.2).

3.2 Limitations of Elisp

The Elisp interpreter in Emacs has three drawbacks concerning recursive functions which are presented in this section:

(1) The Elisp interpreter does not optimize tail calls, and it has a fixed limit on the stack height, meaning that a stack height above a predefined limit is considered as a stack-overflow error.

(2) The Elisp interpreter enforces an upper limit on the number of local variables defined, thus if too many recursive calls are made, too many local variables are introduced and the limit is exceeded.
(3) The Elisp interpreter represents closures and lexical binding of free variables in 4 distinct ways depending on the version of Emacs and the setting of the environment. Having distinct closure representations is a potential problem because some settings of Emacs do not bind free variables at all, while other settings do.

There are two ways to come about the stack-overflow problem (1), and the local variable declaration limitation (2):

(A) set the upper bound on the stack height and the upper bound on the local variable table to a very large number, or
(B) avoid stack-overflow by transforming recursive functions into stack neutral functions.

(A) is a hack, but it works for most inputs. (B) enables any input size, but it also requires program transformations. Once the original code is transformed, each bug fix in the original code will potentially have to be transformed. Program transformations imposes a extra amount of work on the development process, which is why the whole program has not been transformed at this point.

The solution to the problem of how Elisp represents closures and lexical binding (3), can be handled in three distinct ways:

(I) by using the built-in `(lexical-let ...)`-constructs that also works on older versions of Emacs,
(II) by using the `lexical-binding`-feature as introduced in Emacs 24, or
(III) by completely avoiding the native closures in Elisp.

This section demonstrates how to avoid stack-overflow (solution (B)), by transforming a recursive function to a stack neutral counterpart using *continuation-passing style* (CPS) transformations (Section 3.4.2) [[A3] [B9]] and *trampoline* transformations (Section 3.4.3) [B14].

To avoid native closure representation and native lexical binding facilities in the various versions of Emacs, solution (III) is implemented by using a *defunctionalizing* transformation [B11], presented in Section 3.4.4.

3.2.1 Overview

Section 3.3 investigates how Elisp represents functions, and how variables are lexically bound in these function representations. Section 3.4 presents the systematic program transformations necessary for transforming a function to a equivalent counterpart with a fixed stack height. Further Section 3.4 presents a technique for representing closures independently of the meta language.

3.3 Elisp: one language 4 distinct function representations

Elisp has 4 distinct ways to represent functions depending on the version of Emacs and which `lexical-binding`-setting Emacs is setup to use. The `lexical-binding`-setting is available for Emacs 24 and later. It ensures that local variables from an outer lexical scope that are referenced inside a function are lexically bound within the closure representing the function. The `lexical-binding`-setting is an optional feature in Elisp, because it potentially causes legacy code to fail. Legacy code fails, because the dynamic scoping rules allows a function to mention undeclared variables, but this is not accepted when the lexical-binding rules are enabled (unless the variable exists in the global scope at run time).

The 4 distinct ways that a function can be represented in Elisp is outlined here. The goal is to better understand the representations, and to find a solution that works independently of the Elisp environment setup and version.

3.3.1 Function representation with dynamic scope

By default Elisp uses dynamic scope for variables, which means that free variables mentioned in functions are not lexically bound in a closure representing the function. The internal representation for a function is a list of the symbol `lambda`, a list of the formal parameters and a list containing the function body. Consider the example in Figure 3.1. The function `consumer` mentions a variable `x`, that is not declared within the `consumer`-function. The `producer`-function provides a definition of the variable `x` and calls the `consumer`-function. When the `consumer`-function is invoked at run time, both `x` and `y` are bound. The variable `x` is bound by the outer `producer`-function and `y` is bound locally by the `consumer`-function.

3.3.2 Function representation using the lexical-let construct with lexical-binding disabled

Until Emacs version 24, the `(lexical-let ...)`-construct was the only native language construct offering proper lexical-binding for functions in Elisp. The internal representation of a function with lexically bound variables through the `(lexical-let ...)`-construct is illustrated in Figure 3.2: the internal representation of the function `foo` is a deep closure consisting of two `lambdas`. The innermost lambda declares three more formal parameters than the original `foo`-function, one for each of the lexically bound variables: `a`, `b` and `c`.

In the `foo`-function, the lexically bound value of the variable `a` is returned. This is represented by retrieving the value of the reference to `a` by a look up in the symbol-table: `(symbol-value G76333)` (the formal parameter `G76333` is bound to the reference to `a`). The variables `a`, `b` and `c` are represented by the respective symbols `'-a- '-b-` and `'-c-`. The outer `lambda` declares a variable number of formal parameters(`&rest -cl-rest-`), which are used to represent the formal parameters of the `foo`-function, in this case only `x`.

```
(defun consumer (y)
  (list x y))

(defun producer (x)
   (consumer x))

(producer 1)


;; Inner function representation for consumer:
(lambda
  (y)
  (list x y))

;; Inner function representation for producer:
(lambda
  (x)
  (consumer x))
```

Figure 3.1: The dynamic scoping rules allows the binding of the variable `x` to be visible inside the `consumer`-function, when the `consumer`-function is called inside the `producer`-function which declares `x`. With `lexical-binding` enabled, the mentioning of `x` in the `consumer`-function results in a `void-variable`-error at run time (unless `x` is present in the global scope).

All variables from any outer `(lexical-let ...)`-construct are lexically bound in the function representation if just one of them is referenced inside the function body.

3.3.3 Function representation using the let construct with lexical-binding enabled

Elisp in Emacs version 24 provides optional lexical binding of variables, by enabling the `lexical-binding` feature (set the global variable `lexical-binding` to true).

When `lexical-binding` is enabled and at least one of the locally declared variables in an outer scope is mentioned inside the function definition, the function is represented as deep closure, that lexically binds all variables explicitly mentioned in a `(let ...)` or `(lexical-let ...)`-construct within the scope of the function.

However, if no local variables are referenced inside the function definition, only the value of the variables bound with a `(let ...)`-construct is bound by the closure, as illustrated in Figure 3.1.

When the `lexical-binding` is enabled, the dynamic scoping rules still partly apply. A variable that is not lexically bound by the closure representing the function is looked up at run time. If the variable is declared in some outer scope at run time, this value is used, otherwise a `void-variable` error

```
;; A sample function
(lexical-let ((a 42) (b 43) (c 55))
  (defun foo (x)
    a))

;; The internal representation of the function foo:
(lambda
  (&rest --cl-rest--)
  (apply
   '(lambda
      (G76331 G76332 G76333 x)
      (symbol-value G76333))
   '--c-- '--b-- '--a-- --cl-rest--))
```

Figure 3.2: The internal representation of a function with the `lexical-binding` disabled, using a `(lexical-let ...)`-construct to lexically bind the values a=42, b=43 and c=55 to the function `foo`.

is reported. An example of the partly dynamic scoping rules is illustrated in Figure 3.1.

The `lexical-binding`-feature is not backward compatible. Consider the reference to the undeclared variable, `x`, in the `consumer`-function in figure Figure 3.1. Referencing the variable `x`, will result in a `void-variable` error when `lexical-binding` is enabled (unless `x` is defined the global scope at run time), but accepted when `lexical-binding` is disabled.

A closure in Elisp version 24 with `lexical-binding` enabled, is a list that consists of: the symbol `closure`, a list representing the lexical environment always including the variable `t`, a list of the formal parameters of the function and a list containing the body of the function.

```
;; A global variable, c
(defvar c 42)

;; With lexical-binding enabled
(let ((a 42) (b 43))
  (lambda (x) (* x a c)))

;; The resulting closure, where the global
;; variable, c, is not lexically bound.
(closure ((b . 43) (a . 42) t) (x) (* x a c))

;; a is bound even though it is not referenced
(let ((a 42))
  (lambda (x) x))

;; The inner representation:
(closure ((a . 42) t) (x) x)
```

3.3.4 Function representation using the lexical-let construct with lexical-binding enabled

A function using the `(lexical-let ...)`-construct with `lexical-binding` enabled is internally represented much like the legacy `(lexical-let ...)` illustrated in Figure 3.2. However the inner most `lambda` is now represented as a `closure`, that indirectly captures the values of the variables `a`, `b` and `c`. The actual values are stored in an internal symbol table, which is looked up when the values are referenced.

```
;; Enabling  the lexical-binding.
(setq lexical-binding t)

;; A sample function
(lexical-let ((a 42) (b 43) (c 55))
  (defun foo (x)
    a))

;; Inspecting the internal function
;; representation of foo
(pp (symbol-function 'foo))

;; The internal function representation:
(lambda
  (&rest --cl-rest--)
  (apply
   '(closure
     ((--cl-c-- . --c--)
      (--cl-b-- . --b--)
      (--cl-a-- . --a--)
      t)
     (G211626 G211627 G211628 x)
     (symbol-value G211628))
   '--c-- '--b-- '--a-- --cl-rest--))
```

3.3.5 Summary

The 4 ways Elisp represents functions were presented. Each function representation defines its own way of binding local variables. The function representations that lexically bind variables do so by including all variables mentioned in a `(lexical-let ...)`-construct or `(let ...)`-construct, but values of variables that are not declared in a local scope are not bound. A function representation with lexical binding uses dynamic scoping rules to resolve the run time values of variables that were not lexically bound. The function representations is thus a mix of deep lexical binding of variables combined with dynamic scope.

3.4 Stack neutral functions in Elisp by function transformations

In order for a function to have a neutral stack, it has to be acknowledged that:

1. Any recursive function can be rewritten to a tail call counterpart by CPS-transforming it.
2. Any function that is in tail call form can be rewritten in trampolined style [B14], which ensures that only one stack frame is required for the execution.

The CPS transformation together with the trampolining transformation introduces a number of anonymous functions, but as seen in Section 3.3, various versions of Emacs, with various setups interpret and represent closures in distinct ways. To ensure that the closures behave consistently independently of the version of Emacs and its setup, the underlying closure mechanism can be completely avoided by defunctionalizing anonymous functions. Three function transformations are presented:

```
Original --> (1) CPS-transformed --> (2) Trampolined --> (3) Defunctionalized
```

The function illustrated as example in the following sections (Section 3.4.1 - 3.4.4) is a function for extracting key-words (constants) from a BNF. The primary goal is to avoid stack-overflow (Section 3.4.1 - 3.4.3), the secondary goal is to ensure proper closure representations and lexical binding independent of the Emacs version in use (Section 3.4.4).

3.4.1 The original function

The function used as a running example in this section, Section 3.4.2 and Section 3.4.3 is a function for extracting constants from a BNF: `key-words-list-from-bnf`, as shown below. An abstract syntax tree (AST) representing the BNF is used as input for the function (see all the nodes defined by the BNF-AST in Section 3.5.1). The node of interest in the AST is the `constant`-node where constants are extracted. Some nodes contains a list of nodes. In order to handle these lists of nodes in a generic fashion, a case `is-value-list?` is introduced. In the `is-value-list?`-case each node in the list is visited by the `key-words-list-from-bnf`-function. In the "original-function" a `fold-left`-function is used to iterate through the list.

```
1 (defun key-words-list-from-bnf (bnf)
2   (cond
3
4    ((is-constant? bnf)
5     (list (constant-1 bnf)))
6
7    ((is-group-ref? bnf)
8     '())
9
10   ((is-bnf? bnf)
11    (key-words-list-from-bnf (bnf-1 bnf)))
12
13    ...
14
15   ((is-value-list? bnf)
16    (fold-left
17     bnf
```

```
18      (lambda (x rest)
19        (set-append
20          rest
21          (key-words-list-from-bnf x)))
22      (lambda () '())))))
```

The first step to achieve a stack neutral counterpart is to CPS-transform the function as described next (Section 3.4.2).

3.4.2 CPS transformation

A CPS transformation is a technique to transform a recursive function into a tail call counterpart, and if the underlying compiler or interpreter supports tail-call optimizations, a function with k formal parameters will be translated to a `goto` with k arguments by the compiler or interpreter [B20]. The translation from a tail-call function to a `goto`, has a profound impact on the size of the stack at run time. Instead of a stack frame for each function call, which yields a stack size of O(n), a single stack frame of size O(1) containing the k arguments will suffice. Further, a CPS-transformed function runs slightly faster when it has been tail-call optimized, because there is a small overhead of making a function call, compared with the assignment and jump required by the `goto` counterpart.

CPS-transforming a function is not only a technique for transforming a recursive function to a tail call counterpart, but also a mean to achieve control over the flow of the computation. Continuations are closures that capture the lexical environment where they are defined, and thus the state of the program at that point in the computation is stored in the closure. Storing the program state at a given point in the computation in a closure, makes it possible to restore that state again by invoking the closure. A function could even use two continuations (2CPS [B9]), one that is applied to continue the computation on a successful basis and one that is applied to continue the computation in case of failure.

The Elisp interpreter does not perform tail-call optimizations as discussed in Section 3.2, problem (1). Missing tail-call optimizations in Elisp means that a CPS-transformed function will not be converted to a `goto` by the compiler or interpreter, and cause the stack to grow and, in the case of Elisp, adds an overhead to the computation caused by excessive invocations of the garbage collector. However, the CPS transformation is not in vain, it ensures that all function calls are in tail position, which enables a trampoline transformation that makes it possible to reduce the stack size to a single stack frame.

The CPS transformation introduces an extra formal parameter to the function, the continuation (`k`):

```
(defun key-words-list-from-bnf-cps (bnf k) ... )
```

The continuation is a function of one formal parameter expressing the next step of the computation. The formal parameter of the continuation represents the result of the rest of the computation. At every return site a tail call is made, either to the continuation (passing the intermediate result of the computation to the continuation),

```
((is-constant? bnf)
 (funcall k
          (list (constant-1 bnf))))
```

or a tail call to the main function itself, passing along the current continuation along:

```
((is-bnf? bnf)
 (key-words-list-from-bnf-cps
  (bnf-1 bnf)
  k))
```

The body of the continuation function defines how the intermediate result should be used to take the computation further. In the `is-value-list?`-case below, each element of the list should be evaluated. First the head of the list is evaluated. The corresponding continuation (Continuation 1), defines what to happen when the head of the list has been evaluated, namely evaluate the tail of the list or apply the current continuation to the intermediate result if the list is empty. Continuation 2 is responsible for combining the intermediate results from the head of the list with the result from the tail of the list:

```
((is-value-list? bnf)
    (lexical-let ((k_ k) (bnf-cdr (cdr bnf)))
      (key-words-list-from-bnf-cps
       (car bnf)
       (lambda (x)                  ;; Continuation 1
         (lexical-let ((x_ x))
           (if bnf-cdr
               (key-words-list-from-bnf-cps
                bnf-cdr
                (lambda (y)         ;; Continuation 2
                  (funcall
                   k_
                   (set-append x_ y))))
             (funcall
              k_
              x)))))))
```

The `(lexical-let ...)`-constructs are used as proper lexical binding of the local variables (which has to be renamed to be properly bound).
To call the `key-words-list-from-bnf-cps`-function, the initial continuation should be the identity function:

```
(key-words-list-from-bnf-cps
   bnf
   (lambda (x) x))
```

The function is now in tail-call form and it is possible to apply the trampoline-transformation as described next (Section 3.4.3).

3.4.3 Trampoline transformation

A trampoline transformation is a technique to avoid stack-overflow for recursive functions in environments that do not perform tail-call optimizations. This technique is for example applied in *Clojure* [B12] programs to avoid stack overflow, because the Clojure environment is based on the Java Virtual Machine, which suffers from missing tail-call optimizations.

Because the Elisp implementation is not tail-call optimized, the CPS transformation introduces a huge overhead of stack frames. The trampoline transformation applied to the CPS-transformed function reduces the stack to a single stack frame.

The trampoline transformation ensures that a recursive function applies each function call in a single step and returns a thunk representing the remaining computation. By returning a thunk that represents the remaining computation, the stack height will be restricted to a single stack frame.

The trampoline transformation introduces two data types to determine if the computation should stop or continue: `trampoline-continue` and `trampoline-stop`. Every function call is wrapped in a thunk and a `trampoline-continue` data constructor (the thunks and continuations has been annotated with comments, for later identification):

```
((is-constant? bnf)
 (lexical-let ((k_ k) (bnf_ bnf))
   (trampoline-continue  ;; The trampoline-continue data constructor
    (lambda ()           ;; T1, a thunk.
      (funcall k_
               (list (constant-1 bnf_)))))))

((is-bnf? bnf)
 (lexical-let ((k_ k) (bnf_ bnf))
   (trampoline-continue  ;; The trampoline-continue data constructor.
    (lambda ()           ;; T3, a thunk.
      (key-words-list-from-bnf-cps-tramp
       (bnf-1 bnf_)
       k_)))))

...

((is-value-list? bnf)
 (lexical-let ((k_ k) (bnf-cdr (cdr bnf)) (bnf-car (car bnf)))
   (trampoline-continue
    (lambda ()           ;; T9, a  thunk
      (key-words-list-from-bnf-cps-tramp
       bnf-car
        (lambda (x)      ;; C8, a continuation.
         (lexical-let ((x_ x))
           (if bnf-cdr
               (trampoline-continue
                (lambda ()    ;; T6, a thunk.
                  (key-words-list-from-bnf-cps-tramp
                   bnf-cdr
                   (lambda (y)  ;; C5, a continuation.
                     (lexical-let ((y_ y) (x__ x_) (k__ k_))
                       (trampoline-continue
                        (lambda ()  ;; T4, a thunk.
                          (funcall
                           k__
                           (set-append x__ y_)))))))))
             (lexical-let ((k__ k_) (x_ x))
               (trampoline-continue
                (lambda () ;; T7, a thunk.
                  (funcall
```

```
                        k__
                        x_)
                       )))))))))))
```

Only the base case of the computation, which returns the final result, is wrapped in the `trampoline-stop` data constructor, in this case the initial continuation:

```
(defun key-words-list-from-bnf-cps-tramp-main (bnf)
  (trampoline
   (key-words-list-from-bnf-cps-tramp
    bnf
    (lambda (x) (trampoline-stop x))))) ;; C0
```

To run the execution through the evaluation of the thunks a `trampoline`-function is introduced. The `trampoline`-function uses a while-loop to iterate over the thunks, invoking one thunk, checking if the resulting value is a `trampoline-stop` or another `trampoline-continue` containing a thunk to be evaluated:

```
(defun trampoline (v)
  (cond

   ((is-trampoline-stop? v)
    (trampoline-stop-1 v))

   ((is-trampoline-continue? v)
    (setq res (funcall (trampoline-continue-1 v)))
    (while (not (is-trampoline-stop? res))
      (setq res (funcall (trampoline-continue-1 res))))
    (trampoline-stop-1 res))))
```

The `key-words-list-from-bnf-cps-tramp-main`-function is stack neutral. To avoid the native closures of Elisp and issues of lexical binding of variables, the function is further defunctionalized as described next (Section 3.4.4).

3.4.4 Defunctionalization

Defunctionalization is a technique originally introduced by Reynolds in 1972 to specify higher-order definitional interpreters using first-order means [[B11] [B18]]. In order to represent higher-order values with first-order means, closures has to be represented by first-order data types and a first-order `apply` function. Defunctionalization is a way to represent closures as first-order values, rather than as functions of the meta language.

Thus by defunctionalizing, the meta language closures can be completely avoided. Being independent of the Elisp meta level closures, makes the transformed function compatible with any version of Elisp. However, it also turns out to have a positive impact on the evaluation time as demonstrated by the benchmarks in Section 3.4.5.

Defunctionalization replaces each occurrence of a `lambda` with a data type that:

1. represents that `lambda` and
2. captures the free variables referenced from within the `lambda`.

The lexical binding of variables in the function representations is made explicit with data constructors. The continuation of the function is now represented as a data type, `f`. The thunks from the trampoline transformation are also represented as data types. Recall the comments annotating the names of the continuations and the thunks in the trampolined-transformation code snippets in Section 3.4.3. These annotations are now used for naming the data types representing the closures:

```
(defun key-words-list-from-bnf-cps-tramp-defunc (bnf f)
   ((is-constant? bnf)
    (trampoline-continue
     (T1 f bnf)))    ;; T1 represents a thunk

   ((is-bnf? bnf)
    (trampoline-continue
     (T3 (bnf-1 bnf) f)))  ;; T3 represents a thunk.

   ...

   ((is-value-list? bnf)
    (trampoline-continue
     (T9 (car bnf) (cdr bnf) f)))  ;; T9 represents a thunk.
```

Invocation of the closures represented by the data types are (in this case) processed by two `apply`-functions that handles data constructors that:

1 represents the thunks from the trampoline transformation (with no formal parameters), and
2 represents the continuations (with one formal parameter).

Data types representing thunks from the trampoline transformation are applied by invoking the `apply_trampoline_thunk`-function giving the data type as argument, `v`:

```
(defun apply_trampoline_thunk (v)
  (cond

   ((T1? v)
    (apply_cont
     (T1-1 v)
     (list (constant-1 (T1-2 v)))))

   ...

   ((T3? v)
    (key-words-list-from-bnf-cps-tramp-defunc
     (T3-1 v)
     (T3-2 v)))

    ...

   ((T9? v)
    (key-words-list-from-bnf-cps-tramp-defunc
     (T9-1 v)
     (C8 (T9-2 v) (T9-3 v))))))
```

and data types representing continuations are applied by invoking the `apply_cont`-function, providing the data type representing the closure `v` as argument, and the intermediate result of the computation `x`.

```
(defun apply_cont (v x)
  (cond

   ((C0? v)  ;; C0 represents the initial continuation.
    (trampoline-stop x))

   ((C5? v)
    (trampoline-continue
     (T4 (C5-1 v) (C5-2 v) x)))

   ((C8? v)
    (if (C8-1 v)
        (trampoline-continue
         (T6 (C8-1 v) (C8-2 v) x))
        (trampoline-continue
         (T7 (C8-2 v) x))))))
```

The `trampoline`-function, driving the computation forward with a while-loop, has to call the `apply_trampoline_thunk`-functions instead of using the built-in `funcall` to invoke the thunks:

```
(defun trampoline (v)
  (cond

   ((is-trampoline-stop? v)
    (trampoline-stop-1 v))

   ((is-trampoline-continue? v)
    (setq res (apply_trampoline_thunk (trampoline-continue-1 v)))
    (while (not (is-trampoline-stop? res))
      (setq res (apply_trampoline_thunk (trampoline-continue-1 res))))
    (trampoline-stop-1 res))))
```

This way, both the closure representation and the closure invocation are made explicit.

The initial continuation is represented by the data type `C0`:

```
(defun key-words-list-from-bnf-cps-tramp-defunc-main (bnf)
  (trampoline
   (key-words-list-from-bnf-cps-tramp-defunc
    bnf
    (C0))))
```

The function is now stack neutral, no longer uses native closures of Elisp, and lexical binding of variables are explicitly bound by data constructors. In order to investigate the impact on the evaluation time by these function transformations, each intermediate transformation has been benchmarked as described next (Section 3.4.5).

3.4.5 Benchmarks

The function transformations have different impacts on the performance, which is investigated in this section. Each of the program transformed functions as

presented in the previous sections (Section 3.4.1 to 3.4.4) was invoked 100 times with the subset of Scheme as input. Each benchmark is evaluated completely independent. The benchmarks were evaluated by the Emacs 24.3 Elisp interpreter on a MacBook Pro with a 2.33 GHz Intel Core 2 Duo processor and 2 GB 667 MHz DDR2 SDRAM. The result of the benchmarks are depicted in Figure 3.3, and the actual measurements are outlined in Table 3.1. The two CPS-transformed benchmarks has been cut off because their respective evaluation times exceeded 100 seconds, and their maximum stack height was 591 stack frames, but the measured data for the two functions are outlined in Table 3.1. From the benchmarks (depicted in Figure 3.3), the following observations are made (left to right):

(1), (2) The original function (1 and 2) with a stack height of 25 stack frames is clearly the fastest implementation, but suffers from the stack-overflow problem.

(3), (4) CPS-transformed functions (3 and 4) are clearly not optimized by the Elisp interpreter. The garbage collector used 78 pct. of the total evaluation time and the stack height reached 591 stack frames, compared with only 23 stack frames in the original function.

(5) The `lexical-binding` mode introduced in Emacs 24, has a profound impact on the execution time. The CPS-transformed function using the `(let ...)`-constructs (5) was 31 times faster (total time of 3.82 seconds) than the counterparts using the `(lexical-let ...)`-constructs (total time of 119.7 seconds and 121.15 seconds respectively,Table 3.1), and the CPS-transformed version using the `(let ...)`-construct with the `lexical-binding` enabled, only used 22 pct. of the total evaluation time for garbage collection.

(6), (7) Applying the trampoline transformation (using the `(lexical-let ...)`-construct) to the CPS-transformed function (6 and 7) reduces the total evaluation time from respectively 119.7 seconds and 121.15 seconds to 2.6 seconds and the total stack height is reduced to a single stack frame. In both (6) and (7), the garbage collector runs for 28.5 pct. of the evaluation time.

(8) The trampolined-function using the `(let ...)`-construct with the `lexical-binding` enabled (8), performs worse than its two counterpart using the `(lexical-let ...)`-construct, with `lexical-binding` enabled or disabled respectively. The garbage collector runs for 22.8 pct. of the total execution time.

(9), (10) Defunctionalizing the trampolined-function (9 - 10) to avoid the native closures of Elisp, results in a improved evaluation time from 2.6 seconds to 2.3 seconds which is 11 pct. faster. The garbage collector runs for 7.6 pct. of the total evaluation time.

(11), (12) Optimizing the data types used in the defunctionalization to represent closures from lists to vectors, results in another improvement from 2.3 seconds to 2.1 seconds which is an improvement of the evaluation time of 8.7 pct.

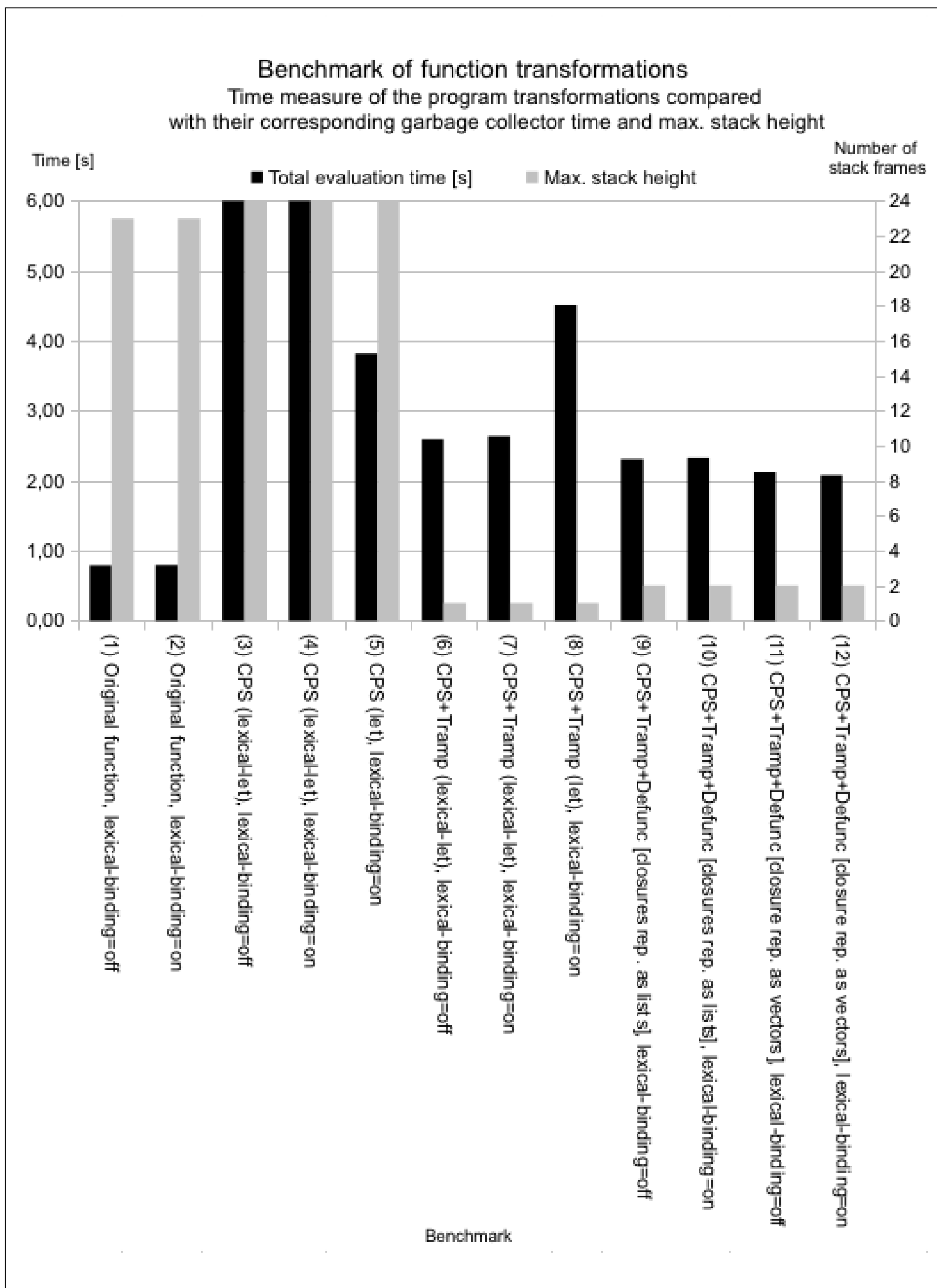

Figure 3.3: Benchmark results for each of the program transformations. The benchmark runs a function that extracts the key-words from the subset of Scheme 100 times. The diagram shows (1) The total evaluation time, (2) The time the garbage collector was running and (3) the maximum stack height resulting from running the function with the subset of Scheme as the input BNF. The y-axis on the left is the time in seconds, the y-axis on the right indicates the maximum stack height measured for the given input.

#	Function	lb	Time [s]	#GC	ms	GC %
1	Original	Off	0.79	1	23	1.24
2	Original	On	0.79	1	23	1.43
3	CPS (lexical-let)	On	119.77	4601	591	78.00
4	CPS (lexical-let)	Off	121.15	4601	591	78.00
5	CPS (let)	On	3.82	16	591	22.82
6	CPS+Tramp (lexical-let)	Off	2.59	76	1	28.52
7	CPS+Tramp (lexical-let)	On	2.64	76	1	28.55
8	CPS+Tramp (let)	On	4.51	22	1	21.81
9	CPS+Tramp+Defunc [closures rep. as lists]	Off	2.31	18	2	7.67
10	CPS+Tramp+Defunc [closures rep. as lists]	On	2.33	18	2	7.48
11	CPS+Tramp+Defunc [closure rep. as vectors]	Off	2.12	16	2	7.21
12	CPS+Tramp+Defunc [closure rep. as vectors]	On	2.08	16	2	7.19

Table 3.1: The benchmark data for the function transformations. lb, indicates if the lexical-binding is on or off. Time [s], indicates the total running time in seconds. #GC, is the total amount of garbage collector invocations doing the benchmark. ms, indicates the maximum stack height. GC %, is the percentage of the total time, that the garbage collector was running.

Summary

A systematical solution to problem 1 - 3 presented in Section 3.2 was demonstrated by applying three well established program transformation techniques: CPS transformation, trampoline transformation and defunctionalization.

A function with a variable stack height was transformed to a semantically equivalent function with a neutral stack height, and it was demonstrated that defunctionalizing that particular function improved the evaluation time, partly because of the reduced garbage collection time.

3.5 Parsing

In this section the parser for input syntax and grammars are presented. Since the input syntax consists of parenthesis and constants, input syntax is referred to as *S-expressions*, likewise the grammar parser is referred to as the BNF parser. The main purpose of the BNF parser and the S-expression parser is to carry text location information about the parsed text. When an error is encountered, this location information enables the analysis to underline the exact grammatical rule that has been violated, along with the concrete syntax violating the rule.

First the AST data structure for representing the parsed BNF is presented (Section 3.5.1), then the S-expression-AST is presented (Section 3.5.2).

3.5.1 BNF-AST

A parsed BNF is represented by an AST with the following nodes:

- (bnf location content), represents the entire BNF.
- (groups location list-of-groups), represents the list of productions (groups) i a grammar, i.e.,

```
⟨a⟩ ::= ...
⟨b⟩ ::= ...
⟨c⟩ ::= ...
```

- (group location name content), represents a single production (group), i.e., <a> ::= (abc <number>).
- (members location list-of-members), represents a list of members inside a production, i.e., ... a | b | c
- (member location content), represents a single member.
- (paren location paren-type content), represents a parenthesis.
- (constant location literal), represents a constant.
- (group-ref location reference), represents a production-reference.
- (star location content), represents a Kleene star, i.e., {a}*.
- (plus location content), represents a plus, i.e., {a}+.

Where location is the start character-position and end character-position in the BNF-source-text where the BNF-term is located. paren-type is one of:

- c, representing round parenthesis "(".
- sqr, representing squared parenthesis "[".

Note that several of the nodes in the AST contains a list of AST-nodes.

3.5.2 S-expression-AST

The S-expression-AST is represented with the following nodes:

- (syntax-paren location paren-type context paren-type)
- (constant location literal)

Note that the S-expression-AST keeps track of both the start and the end type of a parenthesis.

3.5.3 Parsing: common cases

The following text is parsed almost the same way by both the BNF parser and the S-expression parser:

- White space (space, tab or newline), interpreted as token separators and is completely ignored in the output.
- Comments: ; , starting at the semicolon and lasts until the end of the line. Comments are ignored as white space.
- Parenthesis: () []. A parenthesis and its entire content is parsed as a single construct. Information about the begin parenthesis type and end parenthesis type are also stored. It is assumed that all open parenthesis are matched with a closing parenthesis (not necessarily of the same type).

Example 1: accepts `(a . (b . (c . ())))`.

```
⟨quotation⟩ ::=
              ...
              | ()
              | (⟨quotation⟩ . ⟨quotation⟩)
```

Example 2: accepts `(a . (b . (c . ())))` and `(a b c)` - captured by the last rule.

```
⟨quotation⟩ ::=
              ...
              | ()
              | (⟨quotation⟩ . ⟨quotation⟩)
              | (⟨quotation⟩ {⟨quotation⟩}*)
```

Figure 3.4: Desugaring of pairs into lists are not automatically supported, but has to be defined explicit by a grammatical rule as in Example 2.

- Constants, single characters or a sequence of characters, not including white space, single quote, comma, backward quote, string literals (i.e., `"a b c"`) or parenthesis.
- String literals, the input string `“a b c”` parsed as a single constant token. Implications of this is outlined in Section 3.5.5.

Some syntax is treated as syntactic sugar and is automatically converted to its desugared counterpart, this is outlined in Section 3.5.4. Parsing of string literals are outlined in Section 3.5.5. The BNF parser is presented in Section 3.5.6 and the S-expression-parser is presented in Section 3.5.7.

3.5.4 Handling Scheme's syntactic sugar

As discussed in Section 2.5.6, some special cases has been implemented to accommodate some of Scheme's syntactic sugar, namely:

- Single quote: `’` . `’x` is desugared to `(quote x)`, then parsed as: `(paren c ((constant quote) (constant x)) c)`.
- Comma: `,`. `,x` is desugared to `(unquote x)`, then parsed as `(paren c ((constant unquote) (constant x)) c)`.
- Backward quote: `‘`. `‘x` is desugared to `(quasiquote x)`, then parsed as `(paren c ((constant quasiquote) (constant x)) c)`.

Syntactic desugaring of lists as nested pairs is left out, i.e., `’(a b c)` is not desugared into `’(a . (b . (c . ())))`. If the notation `(a b c)` is to be supported on the same terms as `(a . (b . (c . ())))`, a grammatical rule expressing this has to be added, this is illustrated in Figure 3.4.

3.5.5 Handling string literals

A *string literal* is a sequence of characters including white space, starting and ending with a double quote. In the parsing process, white space characters are

Input	Grammar
`(abc "a b c")`	`⟨a⟩ ::= (abc ⟨string⟩)`

Figure 3.5: In order for the input syntax `"a b c"` to be accepted by the grammar, the string literal has to be parsed as a single token, making it possible to match `"a b c"` against the production `<string>`.

Example A

Input	Grammar
`("b")`	`⟨a⟩ ::= ("<b>")` `⟨b⟩ ::= a | b | c`

The input syntax `"b"` is not accepted by the grammar since any grammatical construction surrounded by double quotes " are interpreted as literals. The only accepted input syntax in this example is `"<b>"`.

Example B

Input	Grammar
`("b")`	`⟨a⟩ ::= (⟨string⟩)`

The input syntax is accepted by the grammar. In fact any sequence of characters starting and ending with a double quote will be accepted by this grammar.

Example C

Input	Grammar
`("b")`	`⟨a⟩ ::= (⟨b⟩)` `⟨b⟩ ::= "a" | "b" | "c"`

The input syntax is accepted by the grammar. In fact only `("a")`, `("b")` or `("c")` is accepted by the grammar.

Figure 3.6: Example A, is an example of a grammar that does not accept the syntax as expected due to the way string literals are interpreted by the parsers. Two suggestions to fix the problem is shown in Example B and Example C.

used as token separators, but this should not apply for string literals. In order for the string literals to be correctly matched against the grammar, any input syntax starting and ending with a double quote is parsed as a single token. This is illustrated in Figure 3.5.

Both the S-expression parser and the BNF parser interprets any constant starting and ending with a double quote as a single token. This implies that grammatical constructs surrounded by double quote, such as `<root> ::= "<a>"` will literally match the input syntax: `"<a>"`. In order to define a grammar that accepts input such as `"a"`, `"b"` or `"c"`, the grammar can be specified either by using the built-in `<string>` production as illustrated in Figure 3.6 Example B, or by defining the grammar to only accept `"a"`, `"b"` or `"c"`, using a helper production, as illustrated in Figure 3.6 Example C.

```
⟨root⟩     ::= ⟨ws⟩* ⟨paren⟩+
             | ⟨ws⟩* ⟨constant⟩ ⟨ws⟩*

⟨sequence⟩ ::= ⟨ws⟩* ⟨constant⟩ ⟨ws⟩*
             | ⟨ws⟩* ⟨paren⟩ ⟨ws⟩*
             | ⟨ws⟩* ⟨constant⟩ ⟨ws⟩* ⟨sequence⟩
             | ⟨ws⟩* ⟨paren⟩ ⟨ws⟩* ⟨sequence⟩

⟨paren⟩    ::= ⟨ws⟩* "(" ⟨ws⟩* ⟨sequence⟩ ⟨ws⟩* ")" ⟨ws⟩*
             | ⟨ws⟩* "[" ⟨ws⟩* ⟨sequence⟩ ⟨ws⟩* "]" ⟨ws⟩*
             | ⟨ws⟩* "(" ⟨ws⟩* ")" ⟨ws⟩*
             | ⟨ws⟩* "[" ⟨ws⟩* "]" ⟨ws⟩*

⟨constant⟩ ::= ⟨character⟩+

⟨ws⟩       ::= ⟨space⟩ | ⟨tab⟩ | ⟨newline⟩
```

Figure 3.7: The grammar for the accepted S-Expressions.

3.5.6 The BNF parser

The BNF parser accepts any input as defined by the language of supported grammars in Section 2.4.

Star and plus are interpreted as being part of a constant when they appear next to a constant such as a*, where a* is interpreted as a constant. If a sequence of zero or more a's is wanted, the meta parenthesis should be used: {a}*.

As an addition to the common cases already presented, the grammar parser has to consider the following cases:

1. Production definitions, i.e., <root> ::= x, which is parsed as: (group <root> ((constant x))).
2. Production member separators | .
3. Star * .
4. Plus + .
5. Production references <a>, which is parsed as: (group-ref <a>).
6. Meta parenthesis {}, used to explicitly define the binding of star and plus, and parsed as: (paren curl (..)).

3.5.7 The S-expression parser

The S-expression parser accepts any text as defined by the grammar in Figure 3.7 and returns an AST, representing the input syntax, with text location information attached to each node.

The S-expression parser avoids error checks, leaving as many checks as possible to the syntax checker, rather than reporting errors on its own. Some examples of the parsed input is illustrated in Figure 3.8.

```
[a] b   ==> ((syntax-paren
                (syntax-location 0 3)
                  sqr  ;; type of start parenthesis
                  ((constant (syntax-location 1 2) a))
                  sqr) ;; type of end parenthesis
                  (constant (syntax-location 4 5) b))

'abc    ==> ((syntax-paren
                (syntax-location 0 4)
                  c   ;; type of start parenthesis
                  ((constant (syntax-location 0 4) quote)
                   (constant (syntax-location 1 4) abc))
                  c)) ;; type of end parenthesis

"a b c" ==> ((constant (syntax-location 0 7) "a b c"))
```

Figure 3.8: Examples of parsing of input syntax. The syntax location is carried with the constants.

3.5.8 Limitations of the parsers

The BNF parser and the S-expression parser, assumes that the user provided text is correct according to the supported input languages as outlined in Figure 2.8 and Figure 3.7 respectively.

The parsers do not provide any useful error messages if the input does not conform to the accepted input, i.e., if an invalid BNF is given.

3.5.9 Summary

The limitations concerning function representations in Elisp and the 4 distinct ways that Elisp represents functions depending on version and environment setup were presented. Techniques to completely avoid native closures were presented, but more importantly a technique for avoiding stack-overflow was illustrated. The two techniques for avoiding stack-overflow and native closures are evidence that even if a programming language is missing some of its most basic facilities, it is possible to simulate these facilities manually at the cost of some extra work (both by the evaluator at run time but also by the programmer when implementing the program).

Finally some of the specific details of the two parsers used by the `bnf-mode` were presented.

Chapter 4

User evaluation

The `bnf-mode` was presented to the students following the programming languages course at Aarhus University in the spring of 2013 [B10]. The `bnf-mode` was presented as a help for the students to find and understand syntax errors in Scheme source code. The generic aspect of the `bnf-mode` was also presented, namely that any full-parenthesized BNF can be provided and the syntax checking will be performed based on that BNF. One of the intentions of the `bnf-mode` is to help the students to ensure that they only use syntax that conforms to the subset of Scheme as presented in the course, which will prevent them from introducing a range of typical novice errors.

I attended the weekly study cafe, while the students were working with their assignments. Typically when a student had problems with a function not working as intended, the student and I would go through the Scheme code together following the chain of function calls until the location of the error was found. Once I had spotted the error, I would ask the student to install the `bnf-mode` in Emacs (without telling that I had actually located the error, because in the spirit of the course, I wanted to teach them how to fish, not to catch a fish for them). I would then advise the student on how to use the `bnf-mode` to check for syntax errors, and finally I would ask the student to use the `bnf-mode` to locate the error and try to fix the problem without me interfering.

The user evaluation revealed typical novice errors made by the first-year students when implementing functions in Scheme, but it also revealed which kind of novice errors the `bnf-mode` did not report.

This chapter presents examples of typical errors caught by the `bnf-mode`, and the reactions of the students using the `bnf-mode`.

4.1 Errors caught by the `bnf-mode`

Writing a program in Scheme for the first time can be quite a challenge, and when loading a Scheme program into the Scheme interpreter, it accepts syntax that is not a valid Scheme program as illustrated by several of the errors presented in the following sections (Section 4.1.2 to 4.1.7).

Input

```
(define check-let-expression
  (trace-lambda bar (vs v)
    (and (check-expression v)
    (letrec ([visit
                (trace-lambda foo (purported-clause rest)
                ... )])
      visit (car vs) (cdr vs)))))
```

Grammar

```
 ...
⟨letrec-expression⟩ ::= (letrec ({[<variable>
                                  <lambda-abstraction>]}*)
                                  <expression>)
 ...
```

Figure 4.1: The code in this example is accepted by the Scheme interpreter. The function will always return the result of evaluating the expression `(cdr vs)`. The error is caught by the `bnf-mode`, because the subset of Scheme only accepts one expression in the body of a `letrec`. The student who wrote this code had almost given up on finding the error.

4.1.1 Many students, many versions of Emacs

One of the immediate results of the user evaluations was that the various platforms with various versions of Emacs caused some initial problems with the `bnf-mode`. These problems were quickly resolved, but it was also evidence that the Elisp implementation is version and platform specific as already discussed in Section 3.2.

4.1.2 Missing function application

Forgetting the body of a `letrec`, or forgetting to apply an inner function in the body of a `letrec` are examples of errors that are typically hard to find for a novice programmer. Especially a missing function application is hard to spot, because it is difficult for a novice programmer to see if there is a missing parenthesis. The error is especially hard to find because the Scheme interpreter does not report any errors, because the `letrec`-construct accepts multiple expressions in its body. An example of a missing function invocation in the body of a `letrec` is illustrated in Figure 4.1.

4.1.3 Missing `else` in a `cond`-statement

Missing an `else`-branch in a `cond`-statement often yields an incomplete function definition, resulting in a function that returns `void` for some inputs. In scheme `void` can be interpreted as `true`, since everything that is not explicitly `false` is interpreted as `true`. One student had a predicate function that returned `void` on some inputs, which was then interpreted as `true`, and resulted in an incorrect branch in the `cond`, as illustrated in Figure 4.2.

Input

```
(define is-lambda-abstraction?
  (lambda (x)
    (cond
      [(and (pair? x) (equal? (car x) 'lambda)) #t]
      [(and (pair?  x) ;; Missing else
           (equal?  (car x) 'trace-lambda)) #t])))

(define check-expression
  (lambda (x)
    (cond
      ...
      [(is-lambda-abstraction? x)    ;; is-lambda-abstraction?
       (check-lambda-abstraction x)] ;; returns ⟨void⟩ for
      ...)))                         ;; some inputs.
```

Grammar

```
⟨cond-expression⟩ ::= (cond
                        {⟨cond-clause⟩}*
                        [$\lstul{else}$ ⟨expression⟩])
```

Figure 4.2: A missing `else` in a `cond`-statement might cause a function to return `<void>` for some inputs. A missing `else` in a `cond`-statement is reported as a syntax error by the `bnf-mode` when using the subset of Scheme as input BNF.

4.1.4 Missing else-branch in an `if`-statement

An `if`-statement with a missing `else`-branch might also result in a function that returns `<void>` on some inputs as was also the case in Section 4.1.3. An `if`-statement with a missing `else`-branch will be reported as a syntax error by the `bnf-mode` when using the subset of Scheme as input BNF.

4.1.5 Discovering unexpected errors

In some cases the students found errors in their own code which they first did not believe was actual errors - Scheme did after all accept the function they implemented, and when they invoked the function it gave a result that looked right. Consider the examples in Figure 4.4.

4.1.6 Overwriting built-in functions

Overwriting built-in Scheme function by mistake will lead to errors that are almost impossible to find. The errors that are returned mention nothing about what is really the cause of the problem, namely that a built-in function was redefined. Consider the example in figure 4.5, where a predicate function for `quote` mistakenly overwrites the definition of quote. The student who made this code mistakenly named the predicate-function `quote` instead of `is-quote?` which caused huge problems for him.

Input

```
(define check-and
  (lambda (x)
    (if (equal?  (car x) 'and)(check-expression-list (cdr x)))))
```

Grammar

```
 ...
⟨if-expression⟩ ::= (if ⟨expression⟩ ⟨expression⟩ <expression>)
 ...
```

Figure 4.3: An `if`-statement missing an else-branch is reported as a syntax error when using the subset of Scheme as input BNF.

```
  (cond
    [(pair? v)
      (check-expression (car v))
      (visit (cdr v))]
      ...
      [else ...])
```

The immediate reaction from the student was: "ouh, but this function does not have an error!". But it is actually an error. When using the subset of Scheme with the `bnf-mode`, it results in a syntax error (because the syntax used by the student does not conform to the subset of Scheme), but it is also a semantic error (the code written by the student does not do what the student expected). The student want to use the `and`-function to combine the results of the function call to `check-expression` and the function call to `visit`.

Figure 4.4: This syntax error is caught by the `bnf-mode`, because the syntax did not conform to the subset of Scheme, but the code was accepted by the Scheme interpreter.

4.1.7 Undetected errors

Some syntax might seem like it is obviously wrong, but is not captured by either the `bnf-mode` or the Scheme interpreter. An example of this was discovered by a student, as illustrated in figure 4.6.

4.1.8 Summary

Some of the students who were struggling with their exercises did not use the `bnf-mode`, and as I went to help them find the cause of their problem, I asked why they did not use the `bnf-mode`. They said that it looked like a great tool, but they were too busy solving the exercises to figure out how to install it and use it.

The students who installed and used the `bnf-mode`, all said that the `bnf-mode` helped them to find errors, especially missing or misplaced parenthesis. Some were using the `bnf-mode` to check each function definition one at a time and corrected syntax errors as they were reported by the `bnf-mode`. Other students

Consider the redefinition of quote in Scheme:

```
(define quote
  (lambda (x)
    (equal? (car x) 'quote)))
```

If the quote is used in another context such as:

```
'(1 2 3)
```

The following error will occur: `Exception: attempt to apply non-procedure 1 Type (debug) to enter the debugger.`

Figure 4.5: Example of the built-in function `quote` being overwritten, causing incomprehensible errors to be reported..

The BNF:

```
...
⟨definition⟩ ::= (define ⟨variable⟩ ⟨exp⟩)
...
```

Accepts syntax like:

```
(define foo
  (lamba (v)
    ...))
```

One student was wondering why such an obvious error was not found by the `bnf-mode` and reported it as an error.

Figure 4.6: Example of syntax that looks like an error. The syntax is not recognized as a syntax error by the `bnf-mode` because of the application rule in the grammar for the subset of Scheme. The syntax describes an application of a function called “lamba”.

used the `bnf-mode` just once to verify that all their functions did not contain any syntax errors and conformed to the subset of Scheme.

In general the students using the `bnf-mode` were excited about the help they got from accurate syntax error pinpointing.

Chapter 5

Related work

The system presented in this thesis project is one of many tools made to facilitate teaching and learning of programming languages. The related work covers several aspects:

1. The curriculum for first-year students who are learning a programming language for the first time and the pedagogical approach to teaching programming languages,
2. editors targeted students learning a programming language,
3. investigation of how students respond to error messages, and
4. systems using the same techniques as the `bnf-mode`, in terms of syntax checking based on a subset of some programming language.

5.1 Form over function

M. Sperber et al. [A7] argues why the form of a program is more important than its function, and why students should only be introduced to a small well-defined programming language.

5.1.1 Summary

M. Sperber et al. argues that teaching programming languages to students should not emphasize functionality, i.e., working programs, but rather promote the process of converting concrete problems into program code in a structured way. In their introductory programming course, they introduce the students to a design recipe, that outlines the structure of a program.

They explain why the requirements of a programming language for novice students differ from that of a production languages. Production languages provides features that are easily misunderstood by novice programmers, and when used wrong these features will lead to errors. Working within a subset of a language that do not contain features that are too advanced for the novice programmer, many potential pitfalls are avoided.

They found that focusing on the correctness of a function often results in obscure implementations that might not even work as required. Further, they

found that shifting from *function-oriented* to *form-oriented* has improved the effectiveness of the teaching.

Experienced programmers tend to solve problems using their intuition and experience without being conscious of which methodology they are using. This also applies for teachers. For optimal teaching it is essential to consider what to teach and in which order. They found that continuous repetition of the same *form* of programming at the lectures made the students conscious about the *recipe* for converting problems into code (the thinking of students are: if it is important the teacher will say it again). Using these design recipes, the students avoided the "curse of the blank page" because they had a well-defined starting point when they were to write their first programs from a blank page. The recipe is outline as:

- Write a short description of the procedure.
- Analyze the data involved and determine the type of the data.
- Pick a name for the procedure and write the signature of the procedure.
- Write a number of test cases for the procedure.
- From the signature, make a code skeleton for the procedure.
- Apply a design recipe (defined in their course) based on the signature and data analysis.
- Complete the procedure body.
- Make sure that all the test cases run successfully.

They found that the redundancy helped the students to practice the difference between information and the data that represents it.

When entering the course, some students has a fixed mindset and think they cannot possible learn the programming language. In order to avoid that these students quit or are tempted to plagiarize, early success is important.

They argue that Eclipse and Emacs have poor support for Scheme and thus should not be used. They suggest Dr. Racket as the ideal tool for learning Scheme, since it is possible to restrict Scheme to a subset in the Dr. Racket editor.

5.1.2 Analysis

M. Sperber et al. argues that in the process of learning a programming language, the emphasis should be on the process of converting problems into program code. The language of discourse should be one with a minimal syntax and features. They favor the Dr. Racket editorial system for teaching first-year students programming languages, because it provides three restricted levels of Scheme subsets. Using a subset of Scheme prevents the students from introducing some of the potential errors that are typically caused by unintentional use of language features.

5.2 Measuring the effectiveness of error messages designed for novice programmers

Measuring the quality of error messages based on observing students is investigated by G. Marceau et al. [A6].

5.2.1 Summary

To successfully correct an error in an incorrect program, one must be able to read and understand the error, then formulate a valid correction. To understand an error and come up with a correction is a challenge for novice programmers. Often the cause of an error is indirectly pointed out because an error often triggers a second error. By recording the editorial actions made by students while writing code, G. Marceau et al. are able to identify which actions the students make in order to correct errors encountered in the development process. Based on the editorial history of the students, they found five general error-response groups:

- DEL students who delete the code causing the error rather than correcting it.
- UNR students who changed something unrelated to the error.
- DIFF students who made a useful edit, but not related to the reported error.
- PART students who understood or partly understood the problem and was trying to make appropriate actions to solve it.
- FIX students who fixed the error.

The purpose of the experiment was not to survey or judge the students, but to investigate the quality of the error messages provided by the DrRacket/Scheme environment. The results from the survey was also used to find out which exercises caused problems for the students.

5.2.2 Analysis

The survey had its emphasis on the quality of the error messages reported to the students. They found that some were able to read, understand and correct the errors, while others were clueless as to what was causing the error. Their survey shows that the ability to decode error messages are an important aspect of learning a programming language. The novice programmers are faced with error messages that mentions unfamiliar terms which are incomprehensible to them. The fact that some students tried to correct errors by changing unrelated code or deleted the code causing the problem is evidence that the error messages did not manage to properly explain to the student what is the cause of the problem.

5.3 The structure and interpretation of the computer science curriculum

A structured curriculum for first-year students in programming-languages is presented by M. Felleisen et al. [A4].

5.3.1 Summary

The time in academia is typically the only time where students have the opportunity to focus on general principles of programming languages. M. Felleisen et al. argue that *industrial programming-languages* should not be the main focus of academia, but should be taught on a need to know basis such that the students are familiar with the industrial programming-languages for their summer jobs.

They suggest that the main emphasis of a first-year curriculum should be:

- Learn to read a problem statement carefully and extract useful information.
- Organize programs; match problem analysis and program organization, learn that organization of programs matters for maintenance.
- learn to test their programs before they write them. The students should be critical about the output of their functions. Output that "looks like" the correct output should not be accepted.

They argue that these principles do not favor any language in particular, but could be applied to any language, especially languages where it is convenient to describe a set of values with little effort.

No matter which language is chosen as a introduction language the students will use much time getting use to syntax. Thus a language with a minimal syntax should be favored.

They suggest Scheme as an ideal programming language, in particular they suggest the DrScheme environment for applying their principles. Scheme is a lightweight language with a minimal syntax, the DrScheme environment provides an interactive interpreter for quick evaluation of expressions. Programs can be evaluated without having to understand the underlying concepts of the computer or compiler, such as registers, program pointers, interpreting or compiling code, etc. Pinpointing the source of run-time exceptions are critical elements of the learning process. The DrScheme environment provides such features for Scheme programmers. Scheme is dynamically typed which means that the novice students do not have to worry about understanding static type annotations or fight against a type checker. According to their guide line, they ask the students to annotate their functions in Scheme with a type signature as a comment, thus preparing the students for static types in future courses.

In their book they present a "design recipe" in six steps on how to built programs in a structural way. The book present a minimal subset of Scheme, making the students focus on the design principles rather than the use of language features that they do not understand.

They suggest to teach novice students a small language such as Scheme and once the students have a basic understanding of programming languages, it is easy for them to learn the languages used in the industry.

5.3.2 Analysis

Presenting a "design recipe" for programming in a structural fashion serves as a good starting point for novice students. As a novice student one is asked to

write a program to solve a problem, but writing code from scratch on a blank screen for the first time is hard.

They found that the students are better off from start, if they are given a "cook book" with some basic guidelines explaining how to proceed when solving a problem. Using a recipe for solving problems when writing code is in alignment with M. Sperber et al. [A7] as discussed in Section 5.1.

Compared with the industry, academia has the advantage that students often work alone or together on a project for a relative short period of time, where a company might have the same software base being developed over many years by many different programmers (with very different level of skills). This obviously compromises the programming principles taught at university to some extend.

They suggest that the students are introduced to Scheme because of its minimal syntax, and once the basic concepts are understood the students are able to learn any industrial language with less effort.

5.4 We Scheme

Teaching high school students Scheme in a browser environment is investigated by D. Yoo et al. [B21]. They present the system WeScheme (wescheme.org), which is an easy-accessible online platform for Scheme programming.

5.4.1 Summary

The goal of their project is to show high school students the connection between mathematical functions and programs. They built a web environment for Scheme primarily to avoid installing software on high school computers, but a web platform that requires no installation also enables Scheme programming to reach a broad audience.

Their system provides a user authentication system enabling the students to store and load programs on the web-server. All computation is done on the server thus allowing older computers to run the environment without any problems. The WeScheme environment is executed inside a browser, which enables additional values, such as images, sound and graphical animations.

The system provides syntax highlighting, parenthesis matching and proper indentation. Errors are presented both as highlighted source code, but also as clickable links. When a link is clicked the region of code where the error is found is shown.

5.4.2 Analysis

Placing a Scheme interpreter on a web-page is a way to reach a huge group of people and to demonstrate how Scheme works without having to spend time on installation. Presenting middle- and high school students with a programming language such as Scheme is a way to inspire them to learn about programming languages and how they relate to mathematical problems.

The paper mainly focuses on look and feel of the system, and does not mention how the students experienced the system or how they managed to write their first program.

5.5 DrScheme: a programming environment for Scheme

R. B. Findler et al. [B13] presents a pedagogic programming environment for Scheme, DrScheme (now known as Racket). It is a all-in-one programming environment built to teach Scheme.

5.5.1 Summary

DrScheme was originally developed to help novice students when programming in Scheme. The motivation behind the development was the poor error feedback provided by the Scheme implementations. Simple run-time errors would not be pointed out, and simple syntax errors would cause so much confusion for the students that they went completely off the track.

They present the introductory computing course from Rice university where programs are viewed as consuming data and producing data. They teach the students how to break down problems into input data and output data.

They provide a five step recipe on how to derive a program from a given data structure. This includes writing a set of tests before completing a function definition.

When working with Scheme, there is a clear mapping from algebraic calculations to program execution and the underlying structure of the computer is completely abstracted away.

DrScheme provides a set of editorial tools such as automatic indentation, and a "launch" tool for creating programs the can run independently of DrScheme. Four subsets of Scheme grammar can be selected in the editor. Syntax checking is them performed on the basis of the selected Scheme grammar. Each subset of Scheme extends the previous. This way Scheme syntax can be introduced incrementally by the teachers. Additional checks on the syntax can be imposed, such as errors on unmatched `cond`-cases, and reporting undefined variables where they are referenced.

To avoid old function definitions, they made it easy to restart the REPL such that students are not confused with "ghost code". DrScheme even issues a warning about old function definitions living in the REPL before evaluating an expression or function definition.

Depending on the mode, DrScheme uses an output syntax for values called *constructor syntax*, that looks like the input syntax. Thus the code: `(map add1 (cons 2 (cons 3 empty)))` is output as `(cons 3 (cons 4 empty))` instead of `(3 4)` which is more consistent and is less confusing for the students, since `(3 4)` could be confused with a function application.

DrScheme is capable of highlighting the part of the code where an error was found. A correlation between the original program text and its macro-expanded version is maintained, which allows the system to report the source location of run-time errors.

DrScheme evaluates the Scheme program using reduction semantics, which makes it possible to step through the program execution to follow and understand how a Scheme program is evaluated.

It is possible to jump to function definitions and variable declarations from their references and DrScheme provides many of the shortcuts known from Emacs.

5.5.2 Analysis

DrScheme was developed as stand-alone system for teaching Scheme to novice students. It was made to improve the user friendliness of the Scheme environment by providing tools for better finding and reporting errors such as syntax errors.

DrScheme is easy to install and an installer can be found for the most used platforms. The various levels of predefined subsets of Scheme is a good way for novice students to be introduced to the syntax in an escalating fashion.

It is not possible to provide a custom definition of a Scheme subset in DrScheme, but the latest version of the editor provides a few extra variants of Scheme subsets.

A lot of the Emacs-editing shortcuts has been adopted by DrScheme. The DrScheme editor ships with a set of editorial support tools, such as a static analysis tool for type inference of recursive functions. DrScheme even provides a tool for wrapping a Scheme program into an executable binary file for standalone execution.

5.6 JazzScheme: evolution of a Lisp-based development system

JassScheme is an editorial support system for development of large-scale Scheme or Lisp projects. The system is presented by G. Gartier et al. [B8]and motivates for the development of JazzScheme.

5.6.1 Summary

The author had been using other IDEs and was impressed how efficient a good IDE is, when writing large-scale enterprise software. The author wanted to built a solid IDE for Scheme, meeting three requirements:

- being open-source,
- being implemented in JazzScheme and
- being able to handle large scale enterprise development.

JazzScheme is shipped with an extended version of Scheme and Gambit-C. Gambit-C is a high-performance R5RS-compliant Scheme implementation conforming to the IEEE standard for Scheme and offers a rich standard library. The extended version of Scheme includes a module system, hygienic macros and an object-oriented structure. The JazzScheme environment further offers tools

such as a cross-platform application framework and a built system to make executable binary files for Windows, Mac and Linux.

Functions can be encapsulated to model object-oriented programming. Invoking a function `foo` on an instance `x` is done using a special syntax " ", such as: `(foo  x)`. JazzScheme also supports inheritance with multiple interfaces, as known from Java, along with generic multi-dispatch functions known from Common Lisp. The declaration of functions is close to Schemes define-construct, however it allows for accessor parameters such as private or public. To avoid unknown references at run-time, JazzScheme continuously check references against their definitions.

The JazzScheme system ships with its own IDE environment called Jedi, which is built entirely in JazzScheme and supports among others: Scheme, Common Lisp, C/C++, Java, JavaScript, TEX. The IDE manages project layout, navigation to declarations or references, code evaluation in a REPL like fashion, text manipulation such as a clipboard ring, as known from Emacs, and many of the shortcut key-mappings known from Emacs. The IDE also ships with a debugger, an object inspector and a profiler.

Because Jedi is built entirely in JazzScheme it is possible to fix bugs or add new features to the program at run time, resulting in very short development cycles.

5.6.2 Analysis

JazzScheme was not targeted novice programmers, but is a serious tool for developing large scale systems in Scheme or Lisp, in a manageable fashion. The ability to modify Scheme programs at run time makes experiments and development convenient. As some of the other editors that has been reviewed here, the system offers a set of great editorial tools for easing the development.

5.7 Identifying and correcting Java programming errors for introductory computer science students

A system for identifying a set of typical error patterns in Java is presented by M. Hristova et al. [B15]. The system is targeted students attending an introductory programming course where Java is the target language. They present a preprocessor for Java, called Expresso (misspelled intentionally), which analyzes the input code for typical Java error patterns and return helpful suggestions to the students.

5.7.1 Summary

Expresso is able to identify a set of typical mistakes made by novice programmers and provide more pedagogical error descriptions than the default Java compiler. The program is capable of identifying 20 typical errors made by novice students, and offers help to guide the students to correct the errors on their own.

The 20 error patterns was extracted by contacting 58 schools teaching introductory programming using Java as target language. They collected a list containing 62 different types of errors. From this list they identified the 20 most essential errors to focus on.

They looked at other tools for providing support for learning Java - i.e. BlueJ, but did not find these tools to provide much help on the error feedback side.

5.7.2 Analysis

Identifying common errors made by novice students is a way to help many of the students who become stuck with complicated compiler error messages.

For example the default Java compiler would not give an error when comparing two strings with "==" or when assignment (single equal "=") is mistaken for reference equality (double equal "=="). In Java the `.equals`-method should be used when comparing strings character by character, which is typically the comparison that one wants.

Expresso is a command-line tool that takes any Java file as input and possibly returns easy to understand warnings or errors, or even suggestions on how to fix the problems found.

5.8 BlueJ

M. Kolling [B16] presents BlueJ, an interactive environment built for teaching Java at an introductory level for first-year students.

5.8.1 Summary

BlueJ is a sandbox environment for Java, providing quick access to writing Java programs, without having to worry about compilation or main methods, while giving an easy to overview UML-diagram of the Java classes. The environment is built to facilitate UML design of programs by showing the classes as boxes with the respective UML arrows between them.

It is easy to start programming Java when using BlueJ, because it is easy to setup BlueJ and because the editor provides template code that is useful when one forgets the Java syntax, i.e., in the beginning it is hard for the novice programmer to remember in which order the syntax for declaring a method should be.

BlueJ provides an interactive work-bench where classes can be instantiate and methods on instances can be invoked. This gives the user a feeling of how a class has to be instantiated before its methods can be called, and that the constructors are always called first when instantiating a class.

5.8.2 Analysis

BlueJ is a convenient tool for writing Java programs for the first time and it provide features for the programmer to explore object-instances at run time and

invoke methods manually in a graphical environment. BlueJ is a pedagogical tool that clearly shows the connection between the UML-abstraction and the concrete Java code. BlueJ successfully abstracts away some of the initial hurdles when of learning a programming language, i.e., setup of editor and compiling code.

BlueJ ships with a set of Java examples that the user can use to see what the Java syntax looks like.

5.9 Debugging: the good, the bad, and the quirky - a qualitative analysis of novices' strategies

L. Murphy et al. [B17] presents a survey of novice students debugging capabilities and investigates how teaching can be adjusted to accommodate better teaching of debugging techniques.

5.9.1 Summary

Their goal is to improve teaching of debugging techniques to novice students. They argue that introductory programming texts pays little or no attention to debugging or how to teach it. Further they argue that novice students seems to practice debugging techniques in a unstructured way or that the students have no idea what to do when they encounter an error. They investigate novice strategies for debugging, because currently no best practice for teaching debugging techniques exists.

They gave a group of students a set of exercises, asking them to find and correct semantic errors in Java programs. They observed the students reaction to the exercises and how they worked with the debugging of the programs. Based on their observations, they divided the students in three categories based on their performance: the good, the bad and the quirky.

The students in "the good" group was characterized by: the ability to gain domain knowledge about the problem at hand, i.e., read and understand the specification, they were able to trace the control flow with meaningful printouts. They used test cases to verify the inputs and outputs and some were able to isolate the problem by out-commenting code. Students in "the bad" group was characterized by: poor printouts, poor testing or not testing at all, unable to understand the control flow of the program, did not compile the program before debugging, out-commenting parts of the program unrelated to the error and random edits. The students in "the quirky" group was characterized by: added hacks to the final results to make the overall output seem right, random edits such as correcting spelling mistakes in the comments.

They argue that students should learn debugging techniques in a structured way, including: tracing and testing, heuristics and patterns for applying debugging techniques effectively. This could be achieved by doing traces on paper, using well-placed printout statements and print meaningful information and by learning about debugger tools.

Students should be encouraged to be open to other ways of solving problems:

"what else could I try?", "is this too much to keep track of in my head?" and "what are other possible sources of the bug?".

5.9.2 Analysis

Their work reviles that teaching debugging is actually a good idea, some of the students simply do not know what to do when they experience a program error. The good students are characterized by the ability to read and understand a problem before engaging it, while the bad students are characterized by seemingly random edits. They motivate for introducing printouts to follow the execution control flow, but they also emphasize using a debugger tool which most serious editors have.

It is a great idea to teach debugging, but it is also important for the students to understand why debugging is necessary in the first place, namely that the code did something different than was expected. Errors in a learning context are often a symptom of the students not understanding the language syntax.

5.10 Use of a syntax checker to improve student access to computing

C. Fox et al. [A5] presents a syntax checker for a subset of the programming language WATFIV, a variant of Fortran.

5.10.1 Summary

The motivation behind the syntax checker was to have a tool small enough to be installed on a terminal machine, that novice students can use to check if their programming code is syntactically correct before dispatching it to the mainframe. They found that 67 pct. of the jobs submitted to the mainframe was returned with a syntax error. If these errors can be detected before being dispatched to the mainframe, this would free a lot of computation on the mainframe.

Syntax errors can be incomprehensible for novice students, i.e., a common WATFIV error is a line longer than 72 characters, which is not valid syntax (except for comments). The syntax checker will issue a warning if a line contains any characters after column 72.

Due to the limited capacity of the terminal computers at the time, they were forced to build a syntax checker for a subset of the WATFIV language that accepted a minimal syntax, for example, the `GOTO` statement is not accepted as valid syntax. If a student writes invalid syntax, a syntax error is issued directly on the terminal machine, rather than the on the mainframe, which reduces the time for implementing a program greatly. The WATFIV syntax checker accepts some syntax that the WATFIV compiler does not.

5.10.2 Analysis

The initial motivation for the syntax checker was to save time on the mainframe, but it turned out that the syntax checker also prevented the students from introducing common errors. Common errors was avoided because the syntax checker only accepts a subset without some of the advanced language features of the WATFIV language. This has two advantages:

1. the students are introduced to a minimal syntax and
2. many common novice errors are avoided.

5.11 Summary of related work

As discussed in Section 1.1 and Section 1.2, programming languages is a huge subject which the students find difficult, and they are easily taken off track by compile errors, language syntax, exercises required by the course or stress about the exam.

A common denominator for some of the related articles is the use of *design recipes* [[A7] [A4] [B13]]. A design recipe is a list of steps explaining how to proceed in order to solve a problem in terms of a program, i.e., how to convert a problem into program code.

M. Felleisen et al. [A4] mentions that no language in particular is favored, but it should be one with a minimal syntax. Minimal syntax is favored other authors as well [M. Sperber et al. [A7] C. Fox et al. [A5]] because this prevents the students from being confused about syntax that they have not yet heard about. Syntax checking based on a subset of a language is an other common approach used to restrict the novice students from using syntax advanced features of a language.

Improving the quality of error messages is another common subject [[A6] [B13] [B15]]. Errors provided by a compiler or interpreter often uses terms that are incomprehensible for the students. The errors are incomprehensible because the compilers or interpreters are made for expert programmers. Incomprehensible error messages are handled in one case, by preprocessors to spot typical errors patterns and report easy-to-understand error reports or even suggestions on how to correct the errors. In another case [A6] incomprehensible errors are identified by observing students and how they respond to to various errors.

A Stand-alone-system providing a pedagogical graphical user interface for teaching purposes is another aspect of learning a programming language. The user interface makes the run time state of programs visible to the user, i.e., a system such as BlueJ presented by M. Kolling [B16] shows object-instances at run time, and the user can inspect the object-instances by clicking on the graphical object-instance representations. Other systems aims at improving the error feedback provided by the default compiler [[B15] [A6]].

The stand-alone-systems made for teaching purposes typically provide tools for debugging and editorial support. Some of the systems aims at accessibly, i.e., the WeScheme system [[B21]], that presents a Scheme interpreter for the browser, but in general the systems are capable of abstracting away some

of the less important details of learning about programming languages, such as installing a compiler, setting up build dependencies, invoking the compiler and defining mandatory `main`-methods. The intentions of the these "built-for-learning" systems are good, but it seems like some of them are reinventing something that is already there, i.e., several of them mention that they employ the same key-board shortcuts as Emacs.

One of the common conclusions from the authors is that teaching industry languages should not be the main emphasis, because the students will be capable of easily learning these languages once they understand the basic concepts of programming languages. Academia is the only time where the students have time to experiment with other languages, and this should be embraced.

Chapter 6

Conclusion

An editorial support system built for syntax checking of Scheme code in Emacs was presented. The system is built as a mode for Emacs called the `bnf-mode`. The `bnf-mode` was used by the students attending the programming-languages course and helped them:

- to write Scheme code that conforms to a predefined subset of Scheme (or other grammars, as specified by the user),
- to report precise location of syntax errors, and
- to better understand the correspondence between the syntax of programs and the grammar defining this syntax.

The techniques behind the `bnf-mode` were presented and it was demonstrated how the `bnf-mode` points precisely at syntax errors in the source code and in the BNF. The `bnf-mode` uses a grammatical evaluation that accepts S-expressions as input syntax and BNFs describing fully-parenthesized languages as grammar. The `bnf-mode` uses a metric for comparing partly matched grammar rules, and an optimal approach for evaluating Kleene star and plus, which results in precise error locations both in the input syntax and in the BNF.

It was demonstrated how to overcome stack-overflow problems and to improve closure representations in Elisp by applying systematic program transformations.

The beta version of the `bnf-mode` was installed and used by about half of the students following the course. The students were happy to get a tool to help them to locate errors. They were also surprised as to which kind of errors the tool could actually find. Some students would first argue that a given syntax error was not actually an error, but a bug in the `bnf-mode`, or in the Scheme mode, or in Emacs, or in Petite Chez Scheme, or in Scheme, or in the operating system, etc. Subsequently they would escape this epistemological regression, realize the error of their syntactic ways, and understand how to put the `bnf-mode` to effective use. Nevertheless the students did also point out some of the errors that the `bnf-mode` was unable to detect, which indicates that they put the tool to significant use. The students filed bug reports, but they also suggested changes and features which were implemented on a day-to-day basis.

Programming languages can be taught in many distinct ways, as was reviewed in the chapter about related work (Chapter 5). Several of the authors

agreed on common approaches for teaching novice students programming languages. For example, they presented a “design recipe” which they have applied to their own courses on programming languages. A design recipe is a list of instructions that the students follow in order to learn how to properly structure their programs. Several of the authors suggested that a programming language with a minimal syntax should be favored for novice programmers, which is in direct alignment with the approach of restricted language grammar as discussed in this dissertation.

The goal was to build a generic syntax checker primarily for Scheme, and evaluate it with students, in a way that would be useful in the continuation of their studies, i.e., by using Emacs. This evaluation has concretely demonstrated the usefulness of the `bnf-mode` for first-year students in programming languages at Aarhus University.

Primary Bibliography

[A1] Olivier Danvy. Lecture notes, first week of the programming languages course, spring 2013. 2013.

[A2] Olivier Danvy. Lecture notes, the programming languages course, spring 2013. 2013.

[A3] Olivier Danvy, Jacob Johannsen, and Ian Zerny. A walk in the semantic park. In Siau-Cheng Khoo and Jeremy Siek, editors, *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2011)*, pages 1–12, New York, New York, January 2011. ACM Press. Invited talk.

[A4] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The structure and interpretation of the computer science curriculum. *J. Funct. Program.*, 14(4):365–378, July 2004.

[A5] Christopher Fox and Ronald L. Lancaster. Use of a syntax checker to improve student access to computing. *SIGCSE Bull.*, 16(1):65–68, January 1984.

[A6] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In Thomas J. Cortina, Ellen Lowenfeld Walker, Laurie A. Smith King, and David R. Musicant, editors, *SIGCSE*, pages 499–504. ACM, 2011.

[A7] Michael Sperber and Marcus Crestani. Form over function: Teaching beginners how to construct programs. In Olivier Danvy, editor, *Proceedings of the 2012 ACM SIGPLAN Workshop on Scheme and Functional Programming*, ACM Digital Library, Copenhagen, Denmark, September 2012. Distilled tutorial.

Secondary Bibliography

[B8] Guillaume Cartier and Louis-Julien Guillemette. JazzScheme: Evolution of a Lisp-based development system. 2010.

[B9] Olivier Danvy. *An Analytical Approach to Programs as Data Objects*. DSc thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, October 2006.

[B10] Olivier Danvy. The programming languages course, spring 2013. 2013.

[B11] Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Sci. Comput. Program.*, 74(8):534–549, June 2009.

[B12] Chas Emerick, Brian Carper, and Christophe Grand. *Clojure programming*. 2012.

[B13] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *In Proc. International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 369–388, 1997.

[B14] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In Peter Lee, editor, *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 34, No. 9, pages 18–27, Paris, France, September 1999. ACM Press.

[B15] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. Identifying and correcting Java programming errors for introductory computer science students. *SIGCSE Bull.*, 35(1):153–156, January 2003.

[B16] Michael Kolling. The BlueJ tutorial. January 2002.

[B17] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: the good, the bad, and the quirky – a qualitative analysis of novices' strategies. In J. D. Dougherty, Susan H. Rodger, Sue Fitzgerald, and Mark Guzdial, editors, *SIGCSE*, pages 163–167. ACM, 2008.

[B18] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages

717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363-397, 1998, with a foreword.

[B19] Richard Stallman. The GNU Emacs Lisp Reference Manual corresponding to Emacs version 24.3. 1977-2013.

[B20] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: Compiling Standard ML to C. *LOPLAS*, 1(2):161–177, 1992.

[B21] Danny Yoo, Emmanuel Schanzer, Shriram Krishnamurthi, and Kathi Fisler. WeScheme: the browser is your programming environment. In Guido Rössling, Thomas L. Naps, and Christian Spannagel, editors, *Proceedings of the 16th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2011, Darmstadt, Germany, June 27-29, 2011*, pages 163–167. ACM, 2011.

Chapter 7

Appendix

7.1 Subset of Scheme

```
⟨program⟩               ::= {⟨toplevel-form⟩}*

⟨toplevel-form⟩         ::= ⟨definition⟩
                          | ⟨expression⟩

⟨definition⟩            ::= (define ⟨variable⟩ ⟨expression⟩)

⟨expression⟩            ::= ⟨number⟩
                          | ⟨boolean⟩
                          | ⟨character⟩
                          | ⟨string⟩
                          | ⟨variable⟩
                          | ⟨time-expression⟩
                          | ⟨if-expression⟩
                          | ⟨cond-expression⟩
                          | ⟨case-expression⟩
                          | ⟨and-expression⟩
                          | ⟨or-expression⟩
                          | ⟨let-expression⟩
                          | ⟨letstar-expression⟩
                          | ⟨letrec-expression⟩
                          | ⟨begin-expression⟩
                          | ⟨quote-expression⟩
                          | ⟨quasiquote-expression⟩
                          | ⟨lambda-abstraction⟩
                          | ⟨application⟩

⟨time-expression⟩       ::= (time ⟨expression⟩)

⟨if-expression⟩         ::= (if ⟨expression⟩ ⟨expression⟩ ⟨expression⟩)

⟨cond-expression⟩       ::= (cond
                               {⟨cond-clause⟩}*
                               [else ⟨expression⟩])

⟨cond-clause⟩           ::= [⟨expression⟩]
                          | [⟨expression⟩ ⟨expression⟩]
                          | [⟨expression⟩ =⟩ ⟨expression⟩]
```

```
⟨case-expression⟩        ::= (case ⟨expression⟩
                                {[({⟨quotation⟩}*) ⟨expression⟩]}*
                                [else ⟨expression⟩])

⟨and-expression⟩         ::= (and {⟨expression⟩}*)

⟨or-expression⟩          ::= (or {⟨expression⟩}*)

⟨let-expression⟩         ::= (let ({[⟨variable⟩ ⟨expression⟩]}*) ⟨expression⟩)
                             ;;; where all the variables are distinct

⟨letstar-expression⟩     ::= (let* ({[⟨variable⟩ ⟨expression⟩]}*) ⟨expression⟩)

⟨letrec-expression⟩      ::= (letrec ({[⟨variable⟩
                                        ⟨lambda-abstraction⟩]}*)
                                        ⟨expression⟩)
                             ;;; where all the variables are distinct

⟨begin-expression⟩       ::= (begin {⟨expression⟩}* ⟨expression⟩)

⟨quote-expression⟩       ::= (quote ⟨quotation⟩)

⟨quotation⟩              ::= ⟨number⟩
                           | ⟨boolean⟩
                           | ⟨character⟩
                           | ⟨string⟩
                           | ⟨symbol⟩
                           | ()
                           | (⟨quotation⟩ . ⟨quotation⟩)

⟨quasiquote-expression⟩  ::= (quasiquote ⟨quasiquotation_0⟩)

⟨quasiquotation_0⟩       ::= ⟨number⟩
                           | ⟨boolean⟩
                           | ⟨character⟩
                           | ⟨string⟩
                           | ⟨symbol⟩
                           | ()
                           | (quasiquote ⟨quasiquotation_1⟩)
                           | (unquote ⟨expression⟩)
                           | (unquote-splicing ⟨expression⟩)
                           | (⟨quasiquotation_0⟩ . ⟨quasiquotation_0⟩)

⟨quasiquotation_j⟩       ::= ⟨number⟩
                           | ⟨boolean⟩
                           | ⟨character⟩
                           | ⟨string⟩
                           | ⟨symbol⟩
                           | ()
                           | (quasiquote ⟨quasiquotation_k⟩)
                             ;;; where k = j + 1
                           | (unquote ⟨quasiquotation_i⟩)
                             ;;; where j = i + 1
                           | (unquote-splicing ⟨quasiquotation_i⟩)
                             ;;; where j = i + 1
                           | (⟨quasiquotation_j⟩ . ⟨quasiquotation_j⟩)
```

```
⟨lambda-abstraction⟩    ::= (lambda ⟨lambda-formals⟩ ⟨expression⟩)
                          | (trace-lambda ⟨symbol⟩ ⟨lambda-formals⟩ ⟨expression⟩)

⟨lambda-formals⟩        ::= ⟨variable⟩
                          | ({⟨variable⟩}*)
                            ;;; where all the variables are distinct
                          | ({⟨variable⟩}+ . ⟨variable⟩)
                            ;;; where all the variables are distinct

⟨application⟩           ::= (⟨expression⟩ {⟨expression⟩}*)
```

7.2 Installing the **bnf-mode**

- Obtain the `bnf-mode.el` file from the project website:
  `http://users-cs.au.dk/u061245/bnf-mode/`
- Save the `bnf-mode.el` file to your system.
- Open Emacs, type `C-x C-f`, locate and open the file ~`/.emacs`
- Add the line: `(load-file "c:/path/to/bnf-mode.el")`
- Move the cursor to the end of the line that was just inserted, press `C-x C-e` to load the BNF mode.
- Save the `.emacs` file `C-x C-s`, and close the `.emacs` file `C-x k`.