FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	SECD Virtual Machine Debugger
Student:	Vojtěch Rozhoň
Supervisor:	Ing. Petr Máj
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2022/2023

Instructions

Familiarize yourself with the tinyLisp language and the SECD virtual machine. Implement a tinyLisp to SECD instructions compiler and an SECD virtual machine, both as a module into the Lambdulus system already used in BI-PPA course. The implementation of both must be in TypeScript language and with clean and well documented code that can be used for educational purposes.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 28 December 2021 in Prague.

Bachelor's thesis

SECD VIRTUAL MACHINE DEBUGGER

Vojtěch Rozhoň

Faculty of Information Technology
Katedra teoretické informatiky
Supervisor: Ing. Petr Máj
May 9, 2022

Czech Technical University in Prague
Faculty of Information Technology
© 2022 Vojtěch Rozhoň. Citation of this thesis.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Rozhoň Vojtěch. *SECD Virtual Machine Debugger*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Contents

Acknowledgments vi

Declaration vii

Abstract viii

Acronyms ix

1 Introduction 1
- 1.1 Goals of the thesis 2

2 Tiny-lisp 3
- 2.1 Data Types 3
- 2.2 Grammar 4
- 2.3 Expressions 4
- 2.4 Macros 8
  - 2.4.1 Hygienic macros 8

3 SECD 11
- 3.1 State of the machine 11
- 3.2 Closures 12
- 3.3 Instruction set 13

4 Design 15
- 4.1 Extensions to the SECD 15
- 4.2 Debugger 16
  - 4.2.1 Core module 16
  - 4.2.2 Front-end module 17

5 Realization 19
- 5.1 Core module 19
  - 5.1.1 SECD Compiler 20
  - 5.1.2 Macro expander 26
  - 5.1.3 SECD VM 27
- 5.2 Front-end module 27
  - 5.2.1 Registers 29
  - 5.2.2 AST 30
  - 5.2.3 Source code 33

6 Evaluation 37

7 Conclusion 39
- 7.1 Future work 39

Contents of the enclosed flash disk 43

List of Figures

2.1 An example of cons cells representing expression *(* (+ 1 2) 2)* 3

3.1 An example of an evaluation of an instruction 12

4.1 Lambdulus architecture 16
4.2 A comparison between compilation of a function call and compilation of a macro call 18

5.1 An example of invalid tiny-lisp code with too few arguments to a function and an error message shown by the front-end module 21
5.2 An example of use of an undeclared identifier in tiny-lisp and an error message shown by the front-end module 21
5.3 The submit screen on front-end 28
5.4 The debug screen on front-end 28
5.5 An example of an interpreter error message shown by the debugger 28
5.6 An example of a placeholder called fact in the environment register 29
5.7 An example of nodes representing expression *((lambda (x y) (+ x y)) 10 20)* . . . 30
5.8 An example of a creation of a ReduceNode 31
5.9 An example of a creation of the *removeReduction* method called when returning from a function 32
5.10 An example of a load of value 5 to variable x in the AST 33
5.11 An example of AST of a body of a factorial function before a recursive call to factorial and after returning from the factorial 34
5.12 An example of colouring of the source code and registers when applying a function 35
5.13 An example of mouse interacting with number 10 in source code 36
5.14 An example of hovering with mouse over the variable y, after the value 20 was loaded into it 36

List of Tables

6.1 A comparison of evaluation steps of a program computing factorial 38
6.2 A comparison of evaluation steps of a program computing fibonacci numbers . . 38

List of code listings

1 The grammar of the tiny-lisp language in EBNF format 5
2 An example of invalid tiny-lisp code with two functions calling each other 7
3 An example of valid tiny-lisp code with two functions calling each other 8
4 An example of unhygienic declaration of a macro 9
5 An example of the gensym function used in a macro 10
6 An example of a constant compilation 22
7 An example of the load of a variable compilation 22
8 An example of operators compilation 22
9 An example of the quote operator compilation, resulting in a creation of list containing numbers 4, 5 and 6 23
10 An example of the comma operator inside of a backquoted list 23
11 An example of the if statement compilation 23
12 An example of the lambda function compilation 23
13 An example of compilation of the function call 24
14 An example of the let statement compilation 24
15 An example of the letrec statement compilation 25
16 An example of the letrec statement compilation 25
17 An example of the global function compilation 25
18 An example of a macro expression compilation 26
19 An example of code genereted from a macro reduction 26
20 An example of fibonacci program in tiny-lisp, where user interactions with the code might not work properly 34
21 An example of a program computing factorial of x in lambda calculus 37
22 An example of a program computing factorial of x in tiny-lisp 37
23 An example of a program computing fibonacci number of x in the lambda calculus 38
24 An example of a program computing fibonacci number of x in tiny-lisp 38

I would like to thank my supervisor Ing. Petr Máj for his valuable help, guidance and patience and Bc. Jan Sliacký for his help with the integration of the debugger to the Lambdulus system.

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

In Prague on May 9, 2022 ....

Abstract

This thesis describes an implementation of the debugger of the SECD machine, consisting of the SECD virtual machine and a frontend part. The tiny-lisp programming language is designed to serve as a language that will be evaluated by the virtual machine. The thesis discusses how to compile tiny-lisp expressions, including macros, to the SECD bytecode. The implementation of the debugger focuses on helping students understand key concepts of the SECD machine by interactively showing connections between the source code and the SECD bytecode. The thesis includes a design and implementation of individual parts of the virtual machine and the frontend module.

Klíčová slova debugger, tiny-lisp, interactive interpreter, teaching of functional programming languages

Abstrakt

Práce popisuje implementaci debuggeru SECD stroje, sestávajícího se z SECD virtualního stroje a frontendové části. Programovací jazyk tiny-lisp je navržen jako jazyk, jehož programy budou interpretovány SECD virtuálním strojem, Práce diskutuje jak přeložit výrazy z jazyka tiny-lisp, včetně maker, do SECD bajtkódu. Implementace debuggeru se zaměřuje na pomoc studentům s pochopením klíčových konceptů SECD stroje zdůrazněním souvislostí mezi zdrojovým kódem a SECD bajtkódem. Práce zahrnuje návrh a implementaci jednotlivým částí virtálního stroje a frontendového modulu.

Keywords debugger, tiny-lisp, interaktivní interpreter, výuka funkcionálních programovacích jazyků

Acronyms

API	Application Programming Interface
AST	Abstract Syntax Tree
CSS	Cascading Style Sheets
EBNF	Extended Backus–Naur form
FIT CTU	Faculty of Information Technology at Czech Technical University in Prague
LISP	List Processing
PPA	Programming Paradigms
VM	Virtual Machine
REPL	Read–Eval–Print Loop
SECD	Stack Environment Code Dump

Chapter 1

Introduction

Although functional programming is an essential subfield of computer science, and it is nowadays a must-known skill for all programmers, it is still not very widely known how functional languages are implemented. The knowledge of how the functional programming languages are implemented helps programmers understand the whole functional paradigm.

The Lambdulus system [1] was developed for the BI-PPA (Programming Paradigms) course to help students understand the lambda calculus. The lambda calculus is an abstract concept, which makes it harder to learn it. The way how Lambdulus helps is by presenting the lambda calculus as a real programming language. In the PPA course, functional programming is also thought. The goal of this thesis is to create a debugger of a functional programming language that would show an evaluation of programs written in this language and make it seem less abstract to the students. The tiny-lisp programming language, designed in this thesis, was chosen to be the language to be evaluated by the debugger.

One of the possible implementations is the so-called SECD virtual machine. It is an influential virtual machine created in the 1960s by P. J. Landin and focused on the effective implementation of functional languages. The design of it is fairly abstract and a lot of implementation details remain undescribed. On the other hand, it is remarkably simple. Commonly used languages by programmers are usually compiled directly into the machine code that is then evaluated by the computer. Generated machine code is usually quite big and hard to read. Fortunately, while the presence of multiple stack registers in the SECD VM used for evaluation might be confusing, concepts of the SECD machine are often not so hard to understand and generated SECD bytecode is quite simple in comparison to the common machine code. Hence, it is easier and more effective to create a visualizer of the SECD bytecode evaluation. Although there are some SECD evaluators to be found on the internet, they are all very simple and none of them helps their users to see the connection between the code of the programming language compiled to the SECD bytecode and the SECD bytecode itself. One of the reasons why SECD is not widely understood amongst students and the programming community, in general, is the non-existence of an easy-to-understand visualizer. With such a tool, students could learn key concepts of the SECD bytecode and improve their functional programming skills fairly quickly.

The tiny-lisp programming language is designed to be quite small. This is possible because the language supports macros, and many of the language features can be implemented as a macro.

Chapter 2 of this thesis is devoted to features of the tiny-lisp language. Chapter 3 describes the SECD virtual machine. Chapter 4 considers possible changes to the standard SECD and provides an overall design of the debugger. Chapter 5 talks about my implementation of the SECD debugger, and chapter 6 compares the performance of the implementation to an implementation of a lambda calculus evaluator. The last chapter 7 evaluates the whole thesis.

1.1 Goals of the thesis

The thesis aims to design the tiny-lisp language based on a functional programming paradigm and create a visualized SECD virtual machine to evaluate programs written in the tiny-lisp language.

Tiny-lisp should be designed to be similar to the racket language that students of the BI-PPA course are expected to know. It should support macros. The SECD machine should be discussed and its concepts explained.

Both the tiny-lisp language compiler and the SECD virtual machine should be implemented in the typescript language as a part of the existing Lambdulus system used in the BI-PPA course at FIT CTU, and the solution should be integrated into this system with a proper user interface. The solution should also actively help students understand important concepts of the machine.

It is necessary for the implementation to be clean and well documented, so students could view it to learn more about the implementation of functional programming languages and the implementation of programming languages in general.

Chapter 2
Tiny-lisp

Tiny-lisp is a simple LISP-like [2] programming language. Since students of the BI-PPA course are expected to be more familiar with the racket language [3], tiny-lisp expressions tend to be similar to their racket counterparts.

2.1 Data Types

Tiny-lisp supports just few data types.

Integer An integer value is a signed 64-bit sized number.

Boolean A boolean has two possible values: *false* declared by #f and *true* declared by #t.

String A string is a sequence of arbitrary characters.

Cons cells The cons cell is a data structure containing two members. The first is called *car* and the second *cdr*. Both values are either a primitive value(Integer, Boolean, String) or another cons cell.

List Lists in tiny-lisp work as linked lists. An empty list is called nil. A list is either a nil or a chain of cons cells, where the cdr of one cons cell points to the next cons cell until the last cons cell points to nil. The empty list can be created by the *null* keyword. The figure 2.1 shows an example of a list representing expression *(* (+ 1 2) 2)*

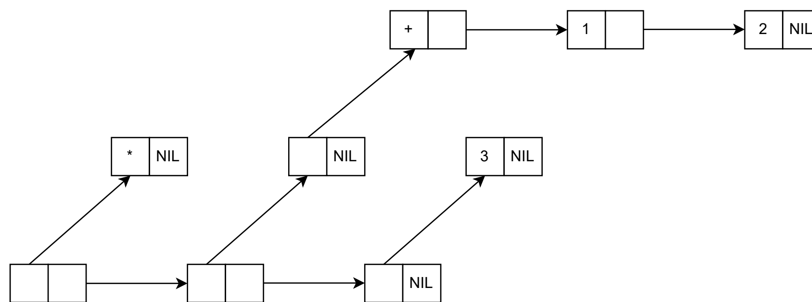


Figure 2.1 An example of cons cells representing expression *(* (+ 1 2) 2)*

S-expressions A symbolic expression (S-expression) is recursively defined as follows: It can either be an atom primitive value (integer, boolean, or string) or a list.

2.2 Grammar

The tiny-lisp language does not distinguish between statements and expressions, so I will be using these terms in the following text as synonyms. Moreover, all data and code in tiny-lisp are S-expressions. Expressions in tiny-lisp are written as lists, where the elements of the list are separated by whitespace. The evaluation of an expression uses the prefix notation. Thus, the first element in the list determines the type of the expression and how the expression will be evaluated. The evaluation of an expression results in a value that can be used as a part of another expression.

The presented grammar 1 is defined in the so-called Extended Backus–Naur Form (EBNF) [4]. Tiny-lisp contains two top-level expressions – global functions and macros. They are defined by using *define* and *define-macro* keywords, respectively. Top-level expressions cannot be nested in other expressions. After their declaration, the names of these functions or macros can be used anywhere in the code. It is also possible to recursively use the name in the body of the top-level expression itself.

Other non-top level expressions can be used, both nested in other expressions and not nested.

The fact that data and code are represented in the same way is a very good precondition for metaprogramming [5] and tiny-lisp macros described below utilize it.

2.3 Expressions

The tiny-lisp program consists of S-expressions that are evaluated from the top of the source code to the bottom. Here I give a list of expressions that are supported by the tiny-lisp language. After the name of each expression, there is an example of it and a result of the example separated by an arrow.

Arithmetical operators

```
(+ 1 3)                    ->                    4
```

Tiny lisp supports operators for addition, subtraction, multiplication, and division and provides multiple options for comparison. An operator expression contains the operator that is followed by its integer arguments and results in an integer. The right argument of the operator is evaluated first.

Quote and backquote operators

```
'(1 (2 3) 4)               ->                    '(1 (2 3) 4)
```

The expression following the quote is prevented from being evaluated. A list of the values in the expression is created instead. The backquote operator works the same way as the quote operator, except that it allows the comma operator to be used inside the quoted list.

Comma operators

```
`(1 ,(+ 1 1) 3)            ->                    '(1 2 3)
```

The comma used inside a backquoted list signals that the expression after the comma will be evaluated.

Cons expressions

```
(cons 1 2)                 ->                    (1 . 2)
```

The cons expression creates a cons cell. The first element in the cons expression list is the *cons* keyword, the second element is the car part of the newly created cons cell and the second element is the cdr part. The dot in the example above signals that the preceding expression of the dot is the car part of the cons cell, while the subsequent expression is the cdr part of the cons cell.

```
⟨top-level⟩ ::= ( , define , ( , ⟨iden⟩ , ⟨args⟩ , ) , ⟨expr⟩ , )
  | ( , define-macro , ( , ⟨iden⟩ , ⟨args⟩ , ) , ⟨expr⟩ , )
  | ⟨expr⟩

⟨args⟩ ::= { ⟨val⟩ }

⟨expr⟩ ::= ( , let , ( , { ( , ⟨iden⟩ , ⟨expr⟩ , ) } , ) , ⟨expr⟩ , )
  | ( , letrec , ( , { ( , ⟨iden , expr⟩ , ) } , ) , ⟨expr⟩ , )
  | ( , lambda , ( , ⟨args⟩ , ) , ⟨expr⟩ , )
  | ( , if , ⟨expr⟩ , ⟨expr⟩ , ⟨expr⟩ , )
  | ( , begin , { ⟨expr⟩ } , )
  | ( , ⟨unary-operator⟩ , ⟨expr⟩ , )
  | ( , ⟨binary-operator⟩ , ⟨expr⟩ , ⟨expr⟩ , )
  | ( , ` , ⟨expr⟩ , )
  | ( , , , ⟨expr⟩ , )
  | ( , { ⟨val⟩ } , )
  | ⟨val⟩

⟨val⟩ ::= ⟨bool⟩
  | ⟨number⟩
  | ⟨iden⟩
  | ' , ( , { ⟨expr⟩ } , )
  | null

⟨unary-operator⟩ ::= car
  | cdr
  | consp

⟨binary-operator⟩ ::= +
  | -
  | *
  | /
  | <
  | <=
  | =
  | =>
  | >
  | or
  | and
  | cons

⟨bool⟩ ::= #t | #f

⟨number⟩ ::= A number is 64-bit valid integer

⟨string⟩ ::= " , sequence of arbitrary characters , "

⟨iden⟩ ::= An identifier starts with a letter. Inside an identifier can be used letters, numbers or
      characters '_' and '-'
```

1 The grammar of the tiny-lisp language in EBNF format

Car expressions

```
(car '(1 2 3))          ->          1
```

The car expression consists of the *car* keyword and one argument that is supposed to be a cons cell. It returns the car member of the cons cell.

Cdr expressions

```
(cdr '(1 2 3))          ->          '(2 3)
```

The cdr expression consists of the *cdr* keyword and one argument that is supposed to be a cons cell. It returns the cdr member of the cons cell.

Consp expressions

```
(consp '(1 2 3))        ->          #t
```

The consp expression consists of the *consp* keyword and one argument. It checks whether the argument is a cons cell. If so, the consp expression returns true. If it is an atomic value, the consp expression returns false.

If expressions

```
(if 0 4 5)              ->          5
```

The if expression is the main option for branching in the tiny-lisp language. It is a list of four elements, where the first one is the *if* keyword, followed by a condition and the true and false branches. First, the condition of the expression is evaluated. Depending on the value returned by this evaluation, one of the branches is chosen to be executed.

Lambda functions

```
((lambda (x y) (+ x y)) 10 20)       ->          30
```

The lambda function statement creates a local function. After the *lambda* keyword, there is a list of parameters of the function. After that, there is an S-expression that contains the body of the lambda that will be evaluated when the lambda function is called. In the example above, the lambda function is the first element in a list. That means that the lambda function is evaluated with other elements of this list as its arguments. Values of arguments replace variables in the body of the lambda. Then the body of the lambda is evaluated, and the result is returned from the function.

Let expressions

```
(let  (
    (x 1)
    (y 2))                               ->          9
(* (+ x y) (+ x y))
```

The let expression is used for binding a variable to an expression. It is possible to bind several variable-expression pairs in a single let expression. The first element of the let expression is the *let* keyword, followed by a list of bindings. Each binding is a list of two elements, where the first element is a variable that is bound to the expression on the second position. The last element in the top-level list is the body of the let statement. It is an S-expression and when a variable from the binding is used here, the expression bound to it is loaded.

Letrec expressions

```
(letrec((fact
    (lambda(n)
        (if (= n 0)
            1                            ->          120
            (* n (fact (- n 1)))))))
(fact 5))
```

The letrec expression is syntactically used the same way as the let expression, with the difference of using the *letrec* keyword instead of the *let* keyword. The letrec expression has the same properties as the let expression, but it also allows using the bound variables in the bound expressions. Hence, when a variable is bound to a function and the variable is also used in this function, a recursive function is created. The use of another let or letrec statement in bindings results in an invalid code.

Begin expressions

```
(begin (+ 1 2) (+ 3 4) (+ 5 6))     ->                  11
```

This is an equivalent of the common-lisp progn expression [2]. In contrast to previously defined expressions, begin expression starts with the *begin* keyword followed by an arbitrary number of arguments. It evaluates all of its arguments and returns the result of the last one. Thus, it enables sequential evaluation.

Global Functions

```
(define (cadr lst)
    (car (cdr lst)))                ->                '(2 1)

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

Global functions are lists with the *define* keyword as their first element. The next element is a non-empty list, where the first element is the name of the function and the other elements are the names of parameters of the function. The last element of the global function is the body of the function, which is an S-expression.

Global functions can be used in the code by creating a list with the function name as the first element and function argument as other elements.

The function can be used anywhere in the code after its declaration and in the global function itself. This means that it is impossible to write a valid code like in the listing 2 in tiny-lisp, where the function *foo* does not know the function *bar* used inside it.

```
(define (foo x)
    (if (= 0 x)
        0
        (+ 1 (bar (- x 1)))))

(define (bar x)
    (if (= 0 x)
        0
        (+ 1 (foo (- x 1)))))

(foo 10)
```

2 An example of invalid tiny-lisp code with two functions calling each other

A solution to this is to not declare functions as global, but rather use a letrec expression with both *foo* and *bar* defined in the bindings. The variables declared in the letrec statement can be used in any of the bound expressions. The figure 3 shows this situation. The code is valid, and its result is 55.

```
(letrec((foo
    (lambda(x)
        (if (= x 0)
            0
            (+ x (bar (- x 1))))))
        (bar
    (lambda(x)
        (if (= x 0)
            0
            (+ x (foo (- x 1))))))
    )
(foo 10))
```

3 An example of valid tiny-lisp code with two functions calling each other

2.4 Macros

Tiny-lisp macros are inspired by LISP [6] and racket [3] macros. Most of the features of the language can be implemented as macros, which allows the language to be quite small in scope. Here is an example of a tiny-lisp macro and its call.

```
(define-macro (if-consp val tb fb)
    `(if (consp ,val)
        ,tb
        ,fb))                   ->      (+ 1 (if (consp 4) (+ 1 2) (+ 2 3)))

(+ 1 (if-consp 4 (+ 1 2) (+ 2 3)))
```

Macros are declared as lists starting with the *define-macro* keyword, followed by a list containing the name of the macro and names of parameters of the macro. This is followed by the body of the macro which is an S-expression. Similar to global functions, macros can be used anywhere in the code, after the macro declaration and in the macro itself, by creating a list, where the first element is the macro name and other elements are arguments of the macro. Similarly, all non-top level tiny-lisp expressions can be used inside the macro.

A big difference between macros and functions is that, while functions are evaluated at runtime, macros are evaluated already at compile time. Moreover, arguments of macros are not evaluated, but they are passed quoted instead.

The process of evaluation of a macro during compile time is called a macro expansion. An S-expression is created as a result of the macro expansion and replaces the macro call in the source code. The macro call may expand into another call of a macro. This call then needs to be also expanded.

Very often, programmers use the backquote immediately at the beginning of the body of the macro. The other common pattern is to use a let statement at the beginning of the body and then use the backquote at the beginning of the body of the let statement.

2.4.1 Hygienic macros

Hygienic macros are macros whose expansion is guaranteed to expand into a code that uses unique identifiers that do not interfere with other identifiers in the code.

Working with unhygienic macros brings potential problems. Let's take a look at them in the code listing 4.

```
(define-macro (cmp a b if-lt if-eq if-ht)
    `(let (
        (lhs ,a)
        (rhs ,b)
    )
    (if (< lhs rhs)
        ,if-lt
        (if (= lhs rhs)
            ,if-eq
            ,if-ht))))

(define (foo x y lhs rhs)
    (cmp x y
        (+ lhs rhs)
        (- lhs rhs)
        (* lhs rhs)))

(foo 6 4 100 255)                       ->                  24
```

4 An example of unhygienic declaration of a macro

Macro *cmp* compares two numbers and based on the result of the comparison evaluates one of the branches passed as arguments. Since the variable *a* is 6 and thus bigger than the variable *b* of value 4, we would expect that the last branch evaluates. The last argument of the macro call is ($*$ *lhs rhs*), so the result should be *lhs* and *rhs* multiplied, thus, 25500. This expectation is wrong, the real result is 24. To understand what happened, we need to take a look at the function *foo* after the macro reduction was performed.

```
(define (foo x y lhs rhs)
    (let ((lhs x)
          (rhs y))
          (if (< lhs rhs)
            (+ lhs rhs)
            (if (= lhs rhs)
                (- lhs rhs)
                (* lhs rhs))))
```

As we can see, variables *lhs* and *rhs* are declared as arguments of the function *foo*, but the result of the macro expansion also use variables with the same names in its body. That is the reason, why all instances of variables *lhs* and *rhs* refer just to the latest declaration of them. Therefore, values 6 and 4 are passed into all occurrences of *lhs* and *rhs* respectively.

There are several possible solutions to this problem. It is possible to use complicated names as variable names inside the macro, but it just makes the problem less likely.

Now assume a function *gensym* that takes no arguments and returns a string that does not meet requirements for a variable name as defined in the tiny-lisp grammar 1. Moreover, this string cannot be already used as a variable name anywhere in the code. With this function, we can change the code to look like listing 5.

Now, the variable names used in the macro are not directly specified by the programmer that wrote the macro, but they are generated by the *gensym* function instead. We are also guaranteed

```
(define-macro (cmp a b if-lt if-eq if-ht)
    (let (
        (tmp1 (gensym))
        (tmp2 (gensym))
        )
    `(let (
        (,tmp1 ,a)
        (,tmp2 ,b)
        )
    (if (< ,tmp1 ,tmp2)
        ,if-lt
        (if (= ,tmp1 ,tmp2)
            ,if-eq
            ,if-ht)))))


(define (foo a b lhs rhs)
    (cmp x y
        (+ lhs rhs)
        (- lhs rhs)
        (* lhs rhs)))

(foo 6 4 100 255)                    ->                  25 500
```

5 An example of the gensym function used in a macro

that the string that is returned from the *gensym* function is not used as a variable name so far. Thanks to this guarantee, macros can be created in a way that the programmer can rely on them that they will not declare any variables that the programmer is already using, so no such variables will arise in the expanded code. Thus, these macros are hygienic.

In the following example, we can see how the expanded code looks like, if the *gensym* function generated strings *"!ff"* and *"!gg"*.

```
(define (foo x y lhs rhs)
    (let ((!ff x)
          (!gg y))
          (if (< !ff !gg)
            (+ lhs rhs)
            (if (= !ff !gg)
                (- lhs rhs)
                (* lhs rhs))))
```

Chapter 3

SECD

In this section, I describe how the SECD virtual machine works. I also list its registers and instructions. SECD is an abstract computer architecture specialized for interpreting functional languages. It was developed in 1964 by Peter J. Landin.

3.1 State of the machine

While running, SECD uses four stack registers implemented as lists. List representation of registers lends itself nicely to the stack API. The top of the register gets the leftmost element of the list, pop removes the leftmost element of the list from the list, and push creates a new cons cell with the pushed element as the car part and the original list as the cdr part. [7]

Next I will list registers used by the SECD VM. [7], [8]

Control It is oftentimes also called the code register. The control register is holding a list of instructions to be executed. They are instructions of the currently evaluated function, if statement branch or a top-level expression. After the evaluation of most of the instruction types, the instruction and its arguments in the code register are popped from the code register.

There are however several instructions (used for branching, exiting the branch, function application, or returning from the function) that construct the content of the new code register differently, as described below.

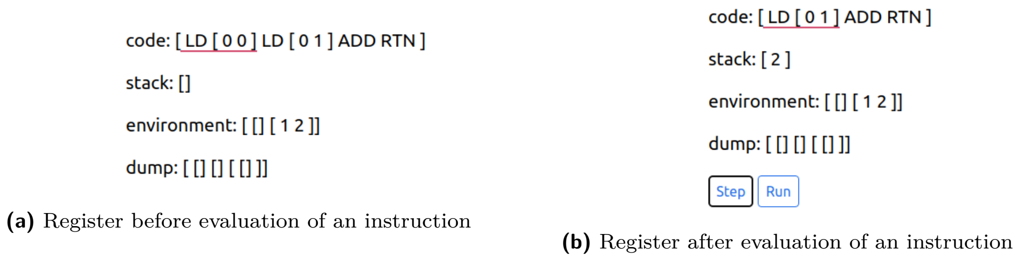
Stack The stack register is used for holding expression evaluation results. It is also used for storing the body of a function and its arguments before its application.

Environment The environment is a list of scopes. A scope is a place in the source code, where a set of variables is defined. In SECD, the scope is implemented as a list of mutually different variable names defined in the scope in the source code. When entering a new scope in the source code, a new scope list is created on the top of the environment register. When leaving the scope, it is popped. For example, when applying the function, a new list is created with values of arguments of the function as values of this list. This list is later popped when we return from the function.

The values are obtained from the environment by using the LD instruction. This instruction takes a list of two numbers as its argument. The first of the numbers is an index of the variable scope where the variable lays. If this index is zero, it refers to the list on the top of the environment register, that is, to the local scope. Similarly, the maximum index refers to the top scope. The second index determines the index of the element in the scope.

Dump The dump register is used for holding backup copies of other registers. There are two cases when registers are saved to dump, namely, evaluating if expressions and function applications. In the former case, only the control register is pushed to dump. In the latter, all control, stack, and environment are pushed to dump. These backups are later popped when we leave the if expression or when we return from the function.

Figure 3.1 shows how evaluation of instruction changes the contents of registers. The left part of the figure shows registers before the evaluation of an LD instruction. The right part shows, how they look after it is performed. After the name of each register, there is a content of it. The squared brackets signal the beginning and end of a list. We can see that the evaluated instruction and its argument are popped from the code register. The result of the LD instruction is pushed to the stack. A value of a variable was searched for in the environment register. The result of the search on indices 0, 0 is a value 2. The pictures in the figure are taken from the implementation created in this thesis.



(a) Register before evaluation of an instruction

(b) Register after evaluation of an instruction

Figure 3.1 An example of an evaluation of an instruction

3.2 Closures

The closure is defined as a list of two elements.

$\langle closure \rangle := [\langle function\text{-}body \rangle, \langle environment \rangle]$

[7]

In the SECD virtual machine, closure is a data structure representing functions. It is built from the function bytecode and the current environment.

The main benefit of closures is that they act as persistent variable scopes. In most programming languages, when we try to load a variable, the language looks for it in the current scope, then in the parent scope, in the parent of the parent scope, etc. until the root scope is reached. However, SECD VM uses closures where variables in the scope are tied directly to the function itself. Thus, any variable present in the scope at the moment of the closure creation can be used in the function.

Closures can also be recursive and contain a reference to themselves in the environment. This allows the SECD VM to successfully interpret recursive functions. Recursive closures are created by RAP and DEFUN instructions which will be described below.

As discussed above, closures are functions with an associated state. Methods in object-oriented programming can be defined in a very similar manner. A class can be implemented in the functional world as a set of closures sharing a common state. Whenever a variable lays in the environment part of the closure but cannot be reached otherwise, we can consider it a private variable of the class. Therefore, the SECD VM can use closures to support an interpretation of objects.

3.3 Instruction set

I will now list the instructions used by the virtual machine.

After each instruction, there is a rule illustrating the effect of the instruction. The definition of the rule is constructed from two parts, divided by the symbol. Both parts describe a list of four elements representing stack, environment, code, and dump registers respectively. The former part of the rule describes a state of registers before the application of the instruction. The letter part describes the state of registers after the application of the instruction. These rules are inspired from [7].

Registers are in these rules represented as lists with the following recursive definition. If the list is represented just by an identifier, then the whole list is tied to this identifier. If the list is represented by brackets that contain a dot and other characters, then the part before the dot describes the car of the list, and the part after the dot describes the cdr of the list. Empty square brackets describe an empty list.

NIL

```
(S E (NIL . C) D) -> (([] . S) E C D)
```

Creates an empty list on the top of the stack register.

LDC

```
(S E (LDC . x . C) D) -> ((x . S) E C D)
```

Loads a constant on the stack.

LD

```
(S E (LD . (i1 . i2) . C) D) ->
(((find value in environment on indices i1 i2) . S) E C D)
```

Loads a variable from the environment on the stack. After the LD instruction, the code should contain a list of two numbers describing the position of the value of the variable in the environment register.

SEL

```
((cond . S) E (SEL . branch1 . branch2 . C) D) ->
    (S E branch1 (C . D)) or (S E branch2 (C . D))
```

SEL is a branching instruction compiled from an if statement. It assumes that there is a boolean value on the top of the stack. After the SEL instruction, there are two lists in the code register representing both branches of the if statement. The code of the branch should always end with a JOIN instruction. Instructions after the SEL instruction are stored on the top of the dump register. The code register is then replaced by one of the branches, depending on the value on the top of the stack.

JOIN

```
(S E (JOIN . C) (C' . D)) -> (S E C' D)
```

The JOIN instruction is used for exiting from the if statement. It simply pops at the top of the dump, where the code following the if statement is stored. Then it pushes it to the code register.

LDF

```
(S E (LDF . f . C) D) -> (((f . E) . S) E C D)
```

The LDF instruction loads a new function on the top of the stack. In SECD, a function is a closure of the body of the function (f) that is located in the code register after the LDF instruction and the current environment. The bytecode of the function should always end with the RTN instruction that is discussed below.

AP

```
(((f . e') . args . S) E (AP . C) D) -> ([] (args . e') f (S . E . C . D))
```

The AP instruction takes care of a non-recursive function application. Namely, it handles entering the function. First, the current states of the code, the stack, and the environment registers are stored on the top of the dump register. The AP instruction also expects a closure of the function code and its environment on the top of the stack and a list of function arguments under them. The function closure is deconstructed, and its code part is used as the new code. The environment part of the closure is the new environment. The stack register is then emptied. Before the execution of the AP instruction, arguments of the function are stored under the top of the stack register. The AP instruction moves these parameters to the top of the environment. Hence, we effectively enter a new function scope. It is expected that the RTN instruction discussed below occurs in the future so that we can return from the function.

RTN

```
((x . S) E (RTN . C) (S' . E' . C' . D')) -> ((x . S') E' C' D')
```

The RTN instruction is the last instruction of a function that takes care of returning from the function. The value on the top of the stack is the returned value. The code and the environment registers are both replaced by their counterparts in the dump register. The stack register is also loaded from the dump register and the returned value is pushed to the top of it. All three lists of these registers are popped from the dump register.

DUM

```
(S E (DUM . C) D) -> (S ([] . E) C D)
```

The DUM instruction creates an empty list on the top of the environment register. This instruction occurs exclusively after compiling a letrec statement. It aims to create a dummy environment that would later become the recursive function itself.

RAP

```
(((f . ([] . E)) . args . S) ([] . E) (RAP . C) D) ->
    ([] (setcar! (([] . E) args) . E) f (S . E . C . D))
```

The RAP instruction stands for recursive function application. Similar to the AP instruction, the RAP instruction firstly saves copies of the code, the stack, and the environment to the dump register. The other similarity is that it takes a closure of the code of the function and an environment of the function from the top of the stack register. However, RAP expects that the value on the top of the environment register is an empty list created by the DUM instruction. The function setcar! is then called. It removes the car of the environment and replaces it with arguments of the function. These arguments are typically the functions that will be recursively applied in the future, while the function closure on top of the stack register is the expression that calls the recursive function for the first time.

Like the AP instruction, the code part of the closure on the top of the stack is used as the new code and the environment part as the new environment. Arguments of the function are under the top of the stack. They are also pushed into the environment and the stack is then emptied.

Operators SECD contains several unary and binary operators. Arguments of the operator are popped from the top of the stack register, and the result of the operator evaluation is pushed to the stack register.

Chapter 4
Design

In this chapter, I discuss extensions to the original SECD machine in my implementation and the overall design of the debugger. The debugger consists of several smaller modules that will be described in detail.

4.1 Extensions to the SECD

In addition to the standard SECD instructions discussed in Chapter 3, there are two more instructions in my implementation. They allow me to compile some expressions of tiny-lisp more practically.

POP

```
((x . S) E (POP . C) D) -> (S E C D)
```

The POP instruction removes the top of the stack register. It is compiled from the begin statement, and it is placed after every compiled expression of the begin statement, except the last one. It serves for removing the result of the expression from the stack.

DEFUN

```
(((f . e'). S) E (DEFUN . C) D) -> (S ((f . e') . E) C D)
```

The DEFUN instruction is compiled from a global function. In my implementation, an empty list is created at the beginning of the evaluation for these global functions. The DEFUN instruction expects a closure on the top of the stack and moves it to the current scope in the environment register. Since the DEFUN instruction is compiled from global functions, the current scope will be always the top scope. Before the DEFUN is executed, the function has to be loaded by the LDF instruction. That means that after LDF evaluates, the environment part of the closure on top of the stack will be the global scope. Then, when the DEFUN instruction is executed, a closure whose environment is the global scope is put to the global scope. Hence, a recursive closure is created, and it is possible to create recursive global functions.

The DEFUN instruction also enables extending the debugger with the Read–eval–print loop (REPL) environment in the future.

4.2 Debugger

The debugger is integrated into the Lambdulus system. Projects in the Lambdulus system consist of two modules. The core module acts as a back-end module and takes care of the processing of the source code and performing evaluation steps. The front-end part takes care of displaying information to the user, taking commands from the user and calling the core module. The 4.1 figure shows how the user, back-end (core) module and front-end module interact with each other in a Lambdulus project.

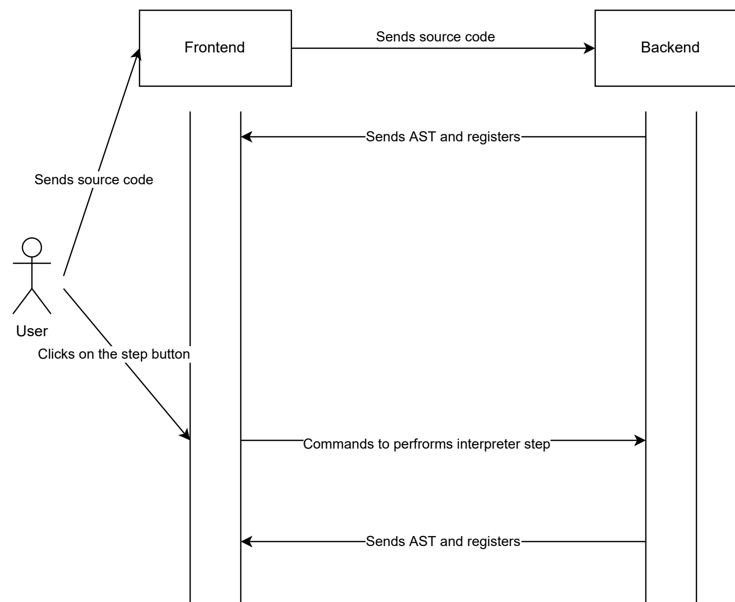


Figure 4.1 Lambdulus architecture

4.2.1 Core module

The core module should have its compiler part that takes care of processing the source code. The compiler uses the lexer part, to transform the source code into syntactic tokens. Then the compiler compiles these tokens to the SECD bytecode. It is possible to first compile the source code to S-expressions and only then compile these S-expressions to SECD bytecode. This approach would be cleaner since tiny-lisp expressions are S-expressions. Implementation of macro expansions would also be more natural. However, it turns out that this intermediate step can be skipped without any trouble, so the compiler in my implementation compiles the source code directly to the SECD bytecode.

While compiling, the core module should be able to ensure the syntax correctness of the source code. When the source code is invalid, the core module should form a helpful error message if possible.

The core module should also construct a representation of the source code and pass it to the front-end module. The representation should be modifiable, while always remembering the

original structure of the source code. There are two possible ways how this representation can be implemented.

The first option is to represent the source code as S-expressions. The first benefit of this approach is easier implementation, particularly if the compiler is implemented to perform an intermediate step to create S-expressions from the source code. The rendering of source code would be also easier on the front-end since the source code is made from S-expressions.

The second option is to represent the source code as an abstract syntax tree (AST). In contrast to S-expressions, the AST consists of nodes of different types. I chose this approach because I want the source code representation to be modifiable by evaluation steps. The AST is more straightforward in this and allows treating every node differently when it comes to modifying the AST or performing operations over the AST. Another benefit is that the AST is probably the most often used solution when it comes to representing the source code.

There should be an AST and compiled SECD code at the end of the compilation process.

The core module should contain an SECD virtual machine part. The SECD VM will contain stack registers and the AST. It should be able to perform an evaluation step based on an instruction and change states of registers and AST in the process. It should offer methods to perform the next instruction evaluation and whole program evaluation.

The core module should also include a macro expander. The macro expander allows the evaluation of macros during the compile time. If the compiler finds a macro call, it should use an instance of the SECD VM to evaluate the macro expansion. The result of this evaluation should be an S-expression, and it should be printed to a string and treated as a replacement for the macro call in the source code. It is possible to prepend this result to the part of the source code that was not yet compiled and continue the evaluation. However, my implementation simply starts a new compiler for the expanded code, and after it is compiled, it continues compiling the source code. This is possible because the result of the expansion should be an S-expression, thus valid tiny-lisp code

The figure 4.2 shows how the compilation of macros and functions differ. Functions are simply compiled to the SECD bytecode, while macros are expanded during compile time by the macro expander and then compiled. It is important to note that the figure shows a compilation of just one macro or function, but the macro expander may produce a call to another macro that also needs to be expanded.

4.2.2 Front-end module

The front-end module should take care of rendering the screens of the debugger to the user. There should be a submit screen, where the user submits their code and a debug screen where they can interact with the state of the debugger and perform new evaluation steps.

The front-end evaluation should start with the submit screen. In the submit screen, the front-end receives a source code from the user and sends it to the core module to compile it and return the AST and registers. Then the front-end should render the debug screen.

After that, the front-end needs to be ready to accept requests from the user to perform evaluation steps. If the request is received, it asks the core module for the result of the step and re-renders the screen. It should be able to highlight parts of the source code and the SECD bytecode that are important for the evaluation of the next expression. The user should also be able to interact with the source code to be able to see connections between the source code and the bytecode when the mouse hovers over the source code.

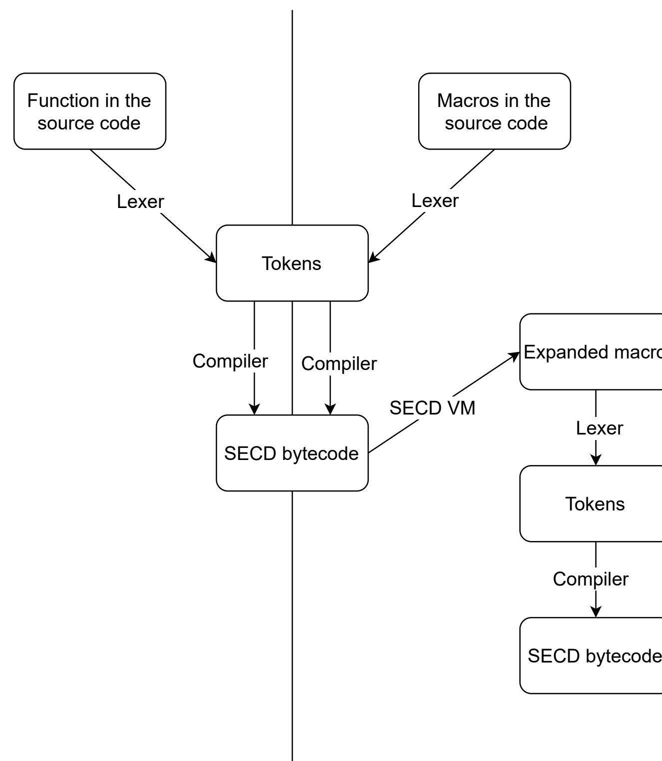


Figure 4.2 A comparison between compilation of a function call and compilation of a macro call

Chapter 5
Realization

The main objective of this thesis is to create a debugger of the SECD machine. This chapter describes, in detail, how individual parts of the machine are implemented. Since the debugger utilizes the Lambdulus system [1] that is written in the typescript programming language [9], both the front-end and the back-end modules of the debugger are also implemented in typescript.

5.1 Core module

As a part of the Lambdulus system, the back-end part of the debugger is implemented as a Lambdulus core module. The first responsibility of the core module is to parse the source code and to create a debugger state consisting of all four stack registers and the AST. The second responsibility is to be able to perform evaluation steps that can modify registers and the AST and pass it back to the front-end.

All data and instructions in SECD registers are implemented via an abstract class named *SECDElement*. The *SECDElement* class is extended by classes *SECDValue*, *SECDArray* and *SECDHidden*. The first one represents primitive data and instructions, while the second one represents lists and the third one is used just to store additional information for function application.

Every *SECDElement* has a variable that stores its colour and a pointer to an AST node. The colour is stored for rendering purposes and will be further described in the 5.2.3.1 section. Multiple elements can point to the same node.

Every *SECDElement* can also be printed by the *print* method. The method *clone* returns a new instance of the element that has the same values of member variables as the original element. There is also a method called *removeReduction* that will be described in the 5.2.2.2 section.

*SECDElement*s are divided into these types:

SECD values An *SECDValue* may be a number, a string or an instruction. All boolean values declared in the source code are converted to either number 0 or number 1.

Numbers and strings are implemented via primitive types provided by typescript. Instructions are implemented via the class *Instruction*. The *Instruction* class contains *instructionShortcut* member variable of enum type *InstructionShortcut* that identifies the type of the instruction. Moreover, the *Instruction* class exports multiple methods that handle the printing of the instruction for debugging and rendering purposes.

SECD arrays SECD lists are implemented via the class *SECDArray*. This class uses composition over a typescript array to store its elements. It supports most of the standard array operations. SECD registers are implemented as *SECDArray*s. The array was chosen as the

container, because it is the default container in the typescript language, and it supports simple syntax for the creation of an array and getting an element on an index. If this implementation proves to be unnecessarily time-consuming, the container can be changed to a linked list.

SECD hidden elements The *SECDHidden* element is used for storing additional information about a function application, and it is not being shown to the user. It is pushed to the dump register when the evaluation of the function begins, and it is popped from there when we return from the function. It contains a pointer to another node beside the one inherited from the *SECDElement* class. This pointer is called *callNode*. The *callNode* represents the part of the AST, from where the function is called. This pointer is stored to update this part of AST when returning from the function with the result of the function.

The second pointer points to the body of the applied function. This pointer is necessary to prepare the AST of the function for another application of this function. It will be further discussed in the 5.2.2 section.

5.1.1 SECD Compiler

The SECD compiler contains several parts. The lexer splits the source code into tokens. The parser ensures the syntax correctness of the program. The main compiler part compiles the source code to the SECD bytecode. The parser and the compiler are implemented in the same class, and the parsing and compiling processes are performed together at the same time.

5.1.1.1 Lexer

The lexer is implemented via the class *Lexer*. It splits the source code into tokens represented via a variable of enum type called *LexerTokens*. Tokens can be keywords, types of values, operators or special characters like brackets. The lexer exports a method called *getNextToken* that returns the next token from the source code. Besides this, the lexer also remembers the last identifier and the last number loaded. The other method that is exported is called *loadExpr*. This method loads the next expression from the source code and returns it as a string. It is needed for a compilation of let or letrec statements.

5.1.1.2 Parser

The parser is implemented in the *Parser* class. It implements a LL(1) syntactic analyser [10] with recursive descent to detect programming errors.

It calls the *getNextToken* method of the lexer to get tokens and expects the tokens to be in a certain order arising from the syntax of the tiny-lisp language defined in the chapter 2. The parser asks for the first token at the beginning of the compilation. Then, each time the type of the token needs to be checked, the parser calls the *compare* method that compares the expected token with the actual one and if they are the same it asks the lexer for a next token. If the expected token differs from the received token, a syntax error is detected, and the parser stops and throws an exception.

The parser contains a member variable *symbTable* that is an instance of the class *SymbTable*. The *SymbTable* class represents a variable scope and contains all variables declared in this scope. It can also have a pointer to another *SymbTable* instance that represents a previous scope. When the use of a variable needs to be compiled, the parser calls the *getPos* method of the *SymbTable* that returns two numbers. The first one is the index of the scope, where the variable was found, and the second is the index of it within the scope.

During parsing, the parser also remembers, for each declared variable, how many arguments it expects. This is the reason why the parser is not only able to detect trivially detectable

errors such as the use of an undeclared identifier or invalid syntax but is also able to spot many other programming errors like a call of an identifier that is not a function, or a wrong number of arguments to a function or an operator. If an error is found, the parser creates an error message and the front-end module shows it to the user. The figure 5.1 shows an example of an error message shown by the front-end module when a function got an insufficient number of arguments. The figure 5.2 shows an example of a front-end error message after the use of an undeclared identifier.

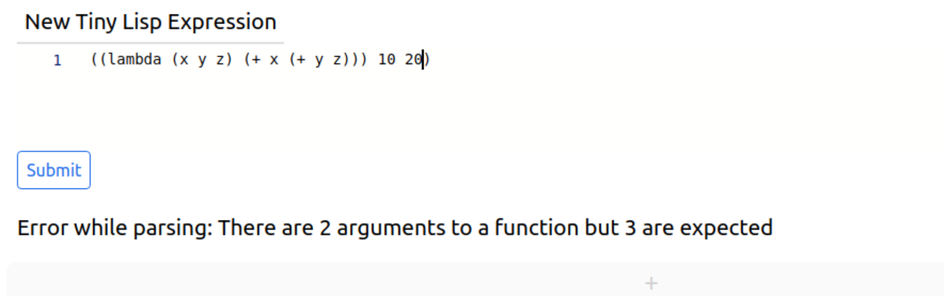


Figure 5.1 An example of invalid tiny-lisp code with too few arguments to a function and an error message shown by the front-end module

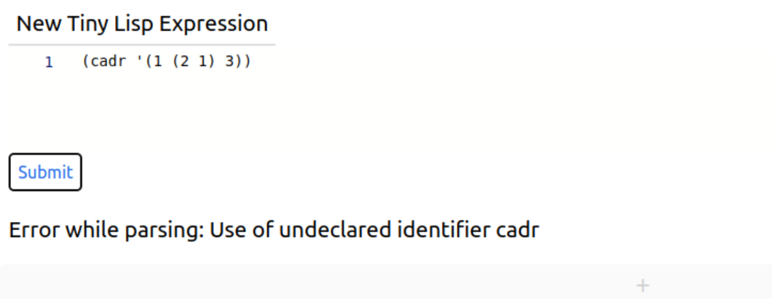


Figure 5.2 An example of use of an undeclared identifier in tiny-lisp and an error message shown by the front-end module

5.1.1.3 Compiler

The compiler is implemented in the *Parser* class. While compiling, the syntax correctness of the program is checked at the same time, as discussed in the section 5.1.1.2. It compiles the source code represented as a string to the SECD bytecode. It also creates an AST representation of the source code. The structure of the AST will be discussed in detail in the 5.2.2 section. The compiler generates the SECD bytecode based on the rules defined in [7]. In the rest of the section, I will show these rules. The left side of each code listing presented is the source code of an expression in the tiny-lisp language, while the right part is the generated bytecode. The square brackets symbolize a list. The rules may contain a function called *compile* that takes an S-expression as its argument and compiles the expression with another compilation rule, depending on the exact type of this expression.

Constants

```
const -> LDC const
```

Constants are compiled into the LDC instruction followed by the constant. The code listing 6 shows an example of a compilation of the number 5.

```
5                                  [ LDC 5 ]
```

6 An example of a constant compilation

Variables

```
identifier ->
LD [<index of the list of values in the environment register>
    <index of the value in the list of values>]
```

Whenever the use of a defined variable is detected, it is compiled to the LD instruction, followed by a list containing two indices. The former index is an index of the list of values in the environment register. The latter index specifies where the value of the variable can be found in the list of values. These indexes are obtained by the method *getPos* in the *SymbTable* class. The code listing 7 shows an example of a compilation of a variable x, assuming it was already defined in the source code. The variables i1 and i2 represent indices of the value of the variable x in the environment register.

```
x                                  [ LD [ i1 i2 ] ]
```

7 An example of the load of a variable compilation

Operators

```
(op arg1 arg2) -> compile(arg1) + compile(arg2) + op
```

All tiny-lisp operators have their SECD instruction counterpart that they are compiled into. The SECD operators are working in the postfix notation. Therefore, the arguments of the operator must precede the operator itself. The code listing 8 shows an example of a compilation of a cons expression.

```
(cons 1 (- (* 2 3) 4))                [ LDC 4 LDC 3 LDC 2 MUL
                                       SUB LDC 1 CONS ]
```

8 An example of operators compilation

Quote and backquote

```
'expr -> LDC expr
```

The expression after the quote operator is not being evaluated, but it is loaded as a constant instead. The backquote operator works similarly, but it also supports the comma operator inside. The code listing 9 shows an example of a compilation of the quote operator that creates a list of values 4, 5 and 6.

Comma

```
,expr -> compile(expr)
```

The expression after the comma is evaluated. The result of the evaluation remains in the same place in the list where was the original expression. The code listing 10 shows an example of the compilation of the comma operator.

```
'(                                      [ NIL
    4                                       LDC 4 CONS
    5                                       LDC 5 CONS
    6                                       LDC 6 CONS
)                                       ]
```

9 An example of the quote operator compilation, resulting in a creation of list containing numbers 4, 5 and 6

```
`(                                      [ NIL
    1                                       LDC 1 CONS
    ,(+ 1 1)                                LDC 1 LDC 1 ADD CONS
    3                                       LDC 3 CONS
)                                       ]
```

10 An example of the comma operator inside of a backquoted list

If statements

```
(if cond branch1 branch2) -> compile(cond) + SEL +
    [compile(branch1) + JOIN] [compile(branch2) + JOIN]
```

The compiled if statement starts with its compiled condition, followed by the branching instruction SEL and both compiled branches. These branches are parts of additional lists that contain the JOIN instruction at the end. The code listing 11 shows an example of the compilation of an if statement.

```
(if 0                                   [ LDC 0 SEL
    1                                       [ LDC 1 JOIN ]
    2)                                      [ LDC 2 JOIN ]]
```

11 An example of the if statement compilation

Lambdas

```
(lambda (args) body) -> LDF + [compile(body) + RTN]
```

Lambdas are compiled into the LDF instruction, followed by a list containing the compiled body of the lambda and the RTN instruction at the end of the list. The code listing 12 shows an example of the compilation of a lambda function.

```
(lambda (x y)                       [ LDF
    (+ x y))                            [ LD [ 0 1 ] LD [ 0 0 ] ADD
                                        RTN ]]
```

12 An example of the lambda function compilation

Function arguments

```
(args) -> NIL + for each in args get (arg -> compile(args) + CONS)
```

Function call

```
(func args) -> compile(args) + compile(func) + AP
```

When calling a function, first its arguments are put into a list on the top of the stack register. That means that the first generated instruction is the NIL instruction to create an empty list. Then each argument is compiled followed by the CONS instruction to move the argument to the list created by the NIL instruction.

Then the function is compiled and at the end, the AP instruction is added. The code listing 13 shows an example of the compilation of a lambda statement with arguments 10 and 20.

```
(                               [ NIL LDC 20 CONS LDC 10 CONS
(lambda (x y)                       LDF
    (+ x y))                        [ LD [ 0 1 ] LD [ 0 0 ] ADD RTN ]
10 20)                              AP ]
```

13 An example of compilation of the function call

Let statements

```
(let (bindings) body) -> NIL + for each in bindings get
    (binding -> compile(binding value) + CONS) +
    [compile (lambda(bindings vars) body) + RTN] + AP
```

The let statement compiles into the NIL instruction, followed by compiled values of bindings in the let statement. Every compiled value is followed by the CONS instruction. Variables of these bindings are then considered arguments of the lambda statement, with the body of the let statement being the body of this lambda. This newly created lambda is then compiled.

The code listing 14 shows an example of a compilation of a let statement declaring variables x and y and using them in its body.

```
(let (                          [ NIL
    (x 1)                           LDC 1 CONS
    (y 2))                          LDC 2 CONS
                                    LDF
(+ x y)                             [ LD [ 0 0 ] LD [ 0 1 ] ADD RTN ]
)                               AP ]
```

14 An example of the let statement compilation

Letrec statements

```
(letrec (bindings) body) -> DUM + NIL + for each in bindings
    get (binding -> compile(binding) + CONS) + LDF +
    [(lambda(bindings vars) body) + RTN] + RAP
```

The letrec statement is compiled almost the same way as the let statement. Since bound variables of the bindings can also be used in the bound expressions, the *Parser* class needs to compile variables first before compiling the expression. This is achieved by utilizing the function *loadExpr* of the lexer. The parser calls it instead of compiling the expressions. Then, when all bindings are loaded and variables are added to the *symbTable*, it starts a new instance of the *Parser* class that compiles the expression.

There are two differences between the generated bytecode of the let expression and the letrec expression. The first difference is that the last instruction is the RAP instead of the AP, and the second one is the additional DUM instruction at the beginning. The combination of DUM

and RAP instructions allows the creation of recursive functions, as discussed in section 3.3. The code listing 15 shows an example of the compilation of a letrec statement that creates a recursive function named *func* that computes factorial numbers.

```
(letrec((fact                         [ DUM NIL
    (lambda(n)                            LDF
        (if (= n 0)                       [ LDC 0 LD [ 0 0 ] EQ SEL
            1                             [ LDC 1 JOIN ]
            (* n (fact (- n 1)))))))      [ NIL LDC 1 LD [ 0 0 ] SUB
                                              CONS LD [ 1 0 ] AP
                                              LD [ 0 0 ] MUL JOIN ]
                                          RTN ]
(fact 2))                             CONS LDF [ NIL LDC 2 CONS
                                          LD [ 0 0 ] AP RTN ] RAP ]
```

15 An example of the letrec statement compilation

Begin statements

```
(begin exprs last) -> for each in exprs
    get (expr -> compile(expr) + POP) + compile(last)
```

Expressions in the begin statement are compiled one by one. There is a POP instruction placed after each compiled expression except the last one. The code listing 16 shows an example of a compilation of a begin statement that consists of three expressions.

```
(begin                              [
    (+ 1 2)                             LDC 2 LDC 1 ADD POP
    (+ 3 4)                             LDC 4 LDC 3 ADD POP
    (+ 5 6)                             LDC 6 LDC 5 ADD
)                                   ]
```

16 An example of the letrec statement compilation

Global functions

```
(define (name args) body) ->
    LDF + [compile(body) + RTN] + DEFUN
```

The global function is compiled the same way as a lambda function, but the DEFUN instruction is added at the end. The code listing 17 shows an example of a compilation of a global function, declaring a function *cadr* that takes one argument and assumes it is a list of at least two elements. It returns all members of the list except the first one and the last one.

```
(define (cadr lst)                  [ LDF
    (car (cdr lst)))                    [ LD [ 0 0 ] CDR CAR RTN ]
                                        DEFUN]
```

17 An example of the global function compilation

5.1.2 Macro expander

The macro expander is also implemented inside the *Parser* class. The parser can detect macro reductions that need to be performed during the compile time. A macro is compiled the same way as a global function. The code listing 18 shows an example of a compilation of a macro called *if-consp*.

```
(define-macro (if-consp val tb fb)      [ NIL
    `(if (consp ,val)                       LDC "if" CONS NIL LDC "consp"
                                            CONS LD [ 0 0 ] CONS CONS
        ,tb                                 LD [ 0 1 ] CONS
        ,fb))                               LD [ 0 2 ] CONS]
```

18 An example of a macro expression compilation

The parser contains a map member variable called *macros*, where keys are macro names and values are codes of these macros. The parser adds a new entry to the map each time a define-macro statement is compiled.

When the parser later finds an identifier that is either a function call or a macro call, the parser searches for the identifier in this map. If it gets a value, it is a macro, otherwise, it is a function.

When a call of a macro is detected, its arguments are simply quoted instead of directly compiled to the SECD bytecode.

The code of the macro is bound to the macro name in the *macros* map. An instance of class *Interpreter* that serves as the SECD VM is then created. The class will be further discussed in the 5.1.3 section. The code of the macro is used as the code register of the SECD VM. Dump and stack registers are empty lists. A list with two elements is used as the environment register. The former element is a list of SECD bytecodes of all global functions and macros that were already parsed, the latter element contains arguments of the macro. The evaluation process is then started by using the run method of the interpreter.

At the end of the evaluation, there should be an S-expression on the top of the stack. This expression is then printed from the *SECDArray* or *SECDValue* representation to a string. The string is then passed to a new parser, so it must represent a valid tiny-lisp code. This is the result of the macro expansion. The left part of the code listing 19 shows the macro *if-consp* called with some arguments. The right part of the code listing, separated from the left part by the arrow, shows the code created by the macro reduction. In the example, the *if-consp* macro is called, and we can see that its second and third arguments are expressions that would get evaluated if this was a call of a function. Then, inside the macro, these arguments replace *fb* and *tb* variable names that are after comma.

```
(if-consp 4 (+ 1 2) (+ 2 3))        ->      (if (consp 4) (+ 1 2) (+ 2 3))
```

19 An example of code genereted from a macro reduction

5.1.2.1 Gensym

As discussed in the section 2.4, the *gensym* function is very important for writing hygienic macros in tiny-lisp. It generates a random string that was not used as an identifier so far. Since tiny-lisp does not support char data type and there are also no utilities for the generation of random strings, the call of the *gensym* function is evaluated at runtime by a special built-in function.

The function is called, by invoking the LD instruction with indexes *-10, -10*. It is impossible to load a variable with negative indices, so we are guaranteed that these indices can not be generated in any other way. Hence, the compiler has to be modified to generate *LD [-10 -10]* bytecode, when it sees the *(gensym)* function in the code. The string generated by the *gensym* function is then pushed on top of the stack.

The string generated by the *gensym* function has one to fifteen characters. The first character is the '!' symbol. The other characters are lower and upper case letters, numbers or characters '-' or '_'. Moreover, the guarantee that the *gensym* function always returns a different string is implemented.

5.1.3 SECD VM

The SECD VM is implemented via the class *Interpreter*. It is a part of the core module responsible for performing evaluation steps. It contains a member variable *interpreterState* of type *InterpreterState*. The *InterpreterState* class is encapsulating four SECD stack registers implemented as *SECDArray*s and the AST. The code register is initialized with the parsed SECD code as described in the section 5.1.1. The environment is initialized with just one empty *SECDArray* inside. It is prepared for storing global functions. The stack and dump registers are empty at the beginning of the evaluation. The variable *lastInstruction* is initialized to DUMMY. The boolean variable *finished* is initialized to false.

The SECD VM exports two different debugging functions, namely *step* and *run*.

The more important of them is the step method. Several things happen each time it is invoked. If *lastIntruction* is not a DUMMY instruction, then *lastIntruction* is evaluated and states of the registers change according to the rules described in the chapter 3. When this is done and the code is not empty, the head of the code register is copied to the *lastIntruction* variable. If the code register is empty, the member variable *finished* is set to true, thus informing about the end of the evaluation.

The DUMMY instruction does not modify any registers or AST so the first trigger of the step method, does not perform any evaluation step. However, the front-end part can use this step to highlight important parts of the next evaluation, as I will further discuss in 5.2.

Besides the step method, the interpreter also exports a run method that runs the step method repeatedly until the variable *finished* is set to true.

There is an *InterpreterUtils* class that exports the *evaluateLoad* method that loads a value from the environment register based on 2 indexes. This method is used in the *Interpreter* call and also in the front-end module in the *Painter* class.

An error may occur during interpretation. An example of this is calling the CDR instruction on an empty list. The *Interpreter* class is in many cases able to provide a helpful error message. This message is thrown in as an exception to the front-end module.

5.2 Front-end module

The front-end module is responsible for rendering the user interface. It is implemented using the typescript language and its React library [11]. The front-end module renders one of two screens.

The first screen of the debugger is the submit screen. The user writes the source code into the field and finishes by clicking the submit button. The figure 5.3 shows an example of a source code prepared to be submitted.

Then the debug screen is shown. On this screen, the user sees the source code and bytecode of all four registers. They can either click on the debug button and perform a debugger step, or they can finish the evaluation by using the run button. The figure 5.4 shows an example of a program being debugged in the debug screen.

New Tiny Lisp Expression

```
1   (define (cadr lst)
2       (car (cdr lst)))
3
4   (define-macro (my-let capture-pair body)
5       `(
6       (lambda (,(car capture-pair)) ,body)
7       ,(cadr capture-pair)
8       )
9   )
10  (my-let (x (+ 1 2)) x)
```

Submit

+

Figure 5.3 The submit screen on front-end

New Tiny Lisp Expression

LISP:
(+ 1 ((lambda (x y)
(+ x y)) 20 10))

code: [NIL LDC 20 CONS LDC 10 CONS LDF [LD [0 1] LD [0 0] ADD RTN] AP LDC 1 ADD]

stack: []

environment: [[]]

dump: []

Step Run

Figure 5.4 The debug screen on front-end

If there is an error during evaluation and an exception is thrown by the core module, the front-end module changes the screen to the submit screen and shows an error message to the user, as can be seen in the figure 5.5.

New Tiny Lisp Expression

```
1   (car '())
```

Submit

The CAR instruction called on an empty array

+

Figure 5.5 An example of an interpreter error message shown by the debugger

5.2.1 Registers

Rendering of registers is the responsibility of the *ReactSECDPrinter* class. The *ReactSECDPrinter* is initialized with an argument of the type *SECDArray* that contains the code of a register and with two callback functions. A React element is then constructed by calling the method *getElements* with the register as its argument. The method *getElements* takes an argument of type *SECDElement* so any subclass of the *SECDElement* class can be passed. The React element might be also assigned a Cascading Style Sheets (CSS) [12] style based on the value of the *colour* variable of the element. The instruction that is currently being evaluated is coloured in red. Arguments of the currently evaluated instruction are coloured in green, blue or yellow. It will be further discussed in the 5.2.3.1 section.

If a *SECDHidden* is passed, an empty React element is created.

If a *SECDValue* is passed, a React element is created from the value.

If a *SECDArray* is passed, the *getElements* is called for each element of the array. A new React element is created from the results of these calls.

5.2.1.1 Placeholders

A *SECDArray* can contain references to itself somewhere inside. The primary cause of it is recursive closures. It means that the rendering of arrays could end up in an infinite cycle. For this reason, *SECDArray*s that were already rendered are replaced by placeholders with a convenient name if they are to be rendered for the second time. They also will not recursively call the *getElements* method on its elements.

What is a convenient name for a placeholder? Luckily, *SECDArray*s containing themselves occur just in two specific cases. The former is a global function or macro, and the name of the placeholder can be taken from its name. The latter is a letrec expression that uses the bound variable name is the bound expression. In this case, the placeholder can be named after the variable name. The *ReactSECDPrinter* class uses the static *getFunctionName* method from the *GeneralUtils* class to deduct the placeholder name.

The *SECDArray* class has a member variable called *printedState* of enum type *PrintedState*. It has 3 possible values – *Not*, *Once* and *More*. The *Not* value is saying that this array has not been rendered yet, the *Once* value means that it was rendered just once so far and the *More* value means that it has already been rendered multiple times.

The first declaration of the array is rendered with additional *'name-of-the-placeholder':* preceding it. Other occurrences of the array are then rendered just as *['name-of-the-placeholder']*. The figure 5.6 shows an example of a placeholder named fact in the environment register.

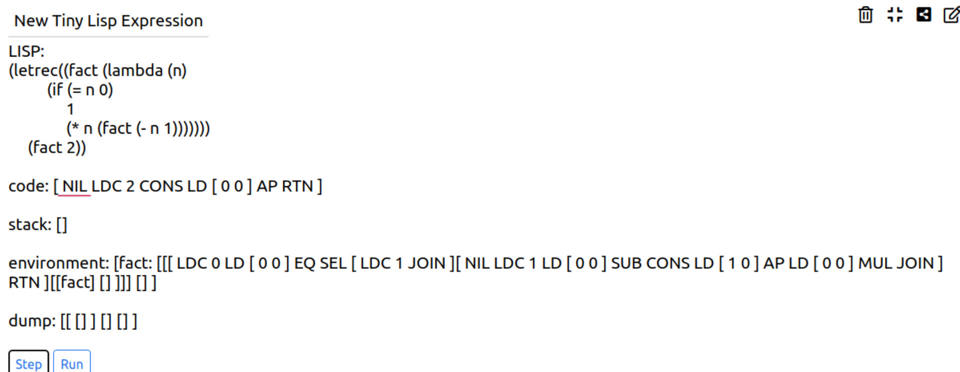


Figure 5.6 An example of a placeholder called fact in the environment register

When the *getElements* method is called with a *SECDArray* as an argument then, before

the recursive calls on members of the array are performed, the *printedState variable* is checked and if the value is not *None*, meaning this array was already rendered, a placeholder is used. Regardless of this check, the *printedInc* method of this array is called. If the *printedState* was *None*, it changes to *Once*, otherwise it is set to *More*.

After the *getElements* method is called on all members of the array, the *printedState* variable is checked again. If the value is *More*, it means that the array contains a reference to itself and therefore a placeholder has to be declared here.

5.2.2 AST

During the compilation of the source code, the abstract syntax tree (AST) is also constructed. AST is a tree-like data structure that remembers the structure of the source code.

Every AST node is implemented as a subclass of the abstract class *Node*. There are two basic types of nodes. The *Topnode* is the root node of the tree and does not directly represent any code. Inner nodes, implemented via the class *InnerNode*, represent a specific part of tiny-lisp language, and there are many types of them. Every inner node has a pointer to its parent and also stores a piece of information about its colour. The figure 5.7 shows an example of an AST representing a lambda function application.

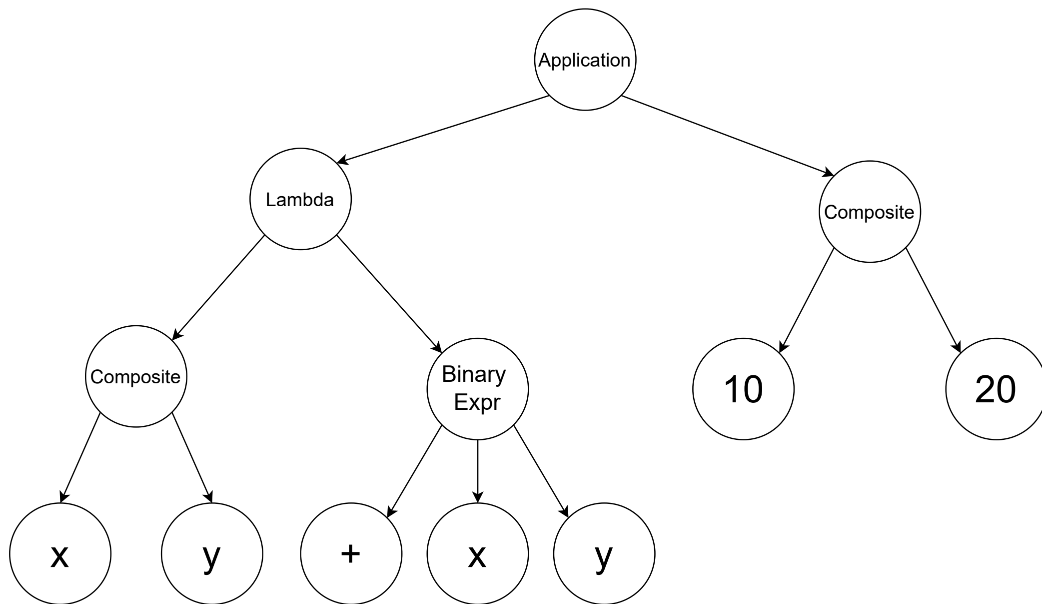


Figure 5.7 An example of nodes representing expression *((lambda (x y) (+ x y)) 10 20)*

5.2.2.1 AST Reduce Nodes

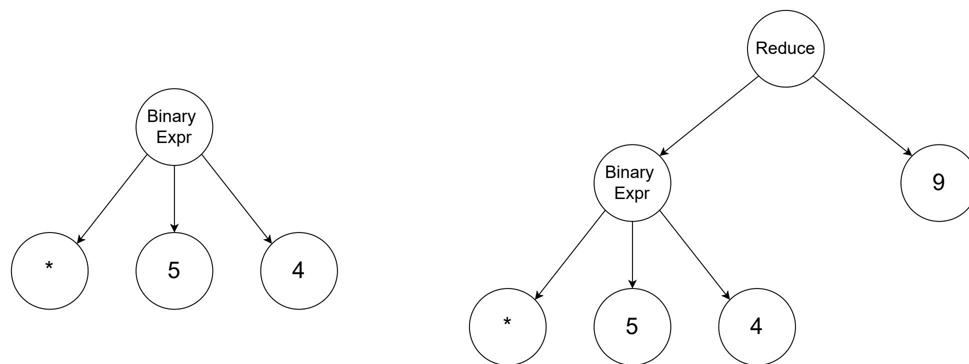
The user of the debugger should be able to see whether a given expression from the source code was already evaluated or not and should be able to check where the potential result of the evaluation currently appears in the bytecode. That means that AST should not only reflect the original structure of the source code but temporary evaluation results as well. In other words, the information about the source code must remain valid while the information about temporary evaluation results is added. This is the idea behind the class *ReduceNode*. Insertion

of a *ReduceNode* to the AST or removal of a *ReduceNode* is the only way how the structure of nodes in the AST can be modified.

The class *ReduceNode* extends the class *InnerNode*. It stores a pointer to a subtree of AST called original and a pointer to a subtree of AST called reduced. The original subtree represents the structure of the source code, and the reduced subtree reflects temporary evaluation results of expressions corresponding to the original subtree.

The reduced subtree corresponds directly to a specific value created either from an evaluation of an operator, the value returned from a function, or a loaded value to a variable. The figure 5.8 shows a *ReduceNode* created from an evaluation of an operator.

The first case, when insertion of a *ReduceNode* occurs, is an evaluation of an operator. The original subtree of the *ReduceNode* is the subtree of the evaluated expression. The reduced subtree is the result of the evaluation. The second case for insertion of a *ReduceNode* is returning from the function. The original subtree is the subtree representing the application of the function, while the reduced subtree is the value returned from the function. The third case, when insertion of a *ReduceNode* occurs, will be discussed in the section 5.2.2.3.



(a) Nodes of the binary expression before the evaluation of the expression

(b) Nodes of the binary expression after the evaluation of the expression

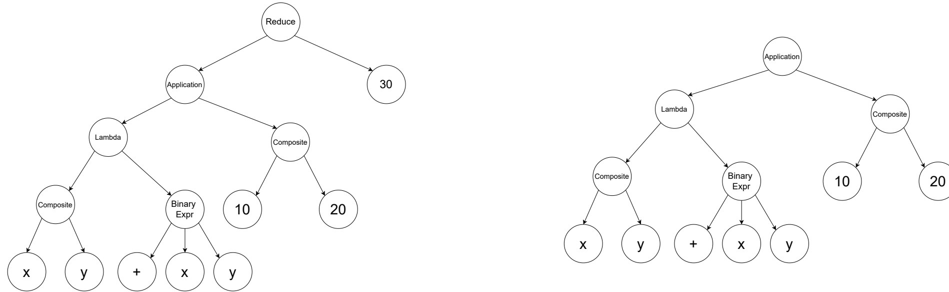
Figure 5.8 An example of a creation of a ReduceNode

5.2.2.2 AST and functions

When returning from a function, a node representing the call of the function should be updated by the node corresponding to the returned value. To remember which node should be updated, the AP or RAP instructions that start the evaluation of the function push an element of *SECDHidden* type to the dump register. The *SECDHidden* stores a node of the place from where the function is called and a node of the function itself. The figure 5.9 shows an example of a *ReduceNode* creation, after returning from the function.

Functions are usually used by programmers in the hope to use the same code multiple times. The AST subtree representing the code of the function should on the other hand occur just once. When the function is applied, some expressions presumably get evaluated in the function and their nodes are updated. When the function is applied for the second time, nodes of the function should not contain these results computed in the previous application of the function. Therefore, the AST should be prepared to remove *ReduceNode*s before the next evaluation of the function. That means that *ReduceNode*s inside AST subtree belonging to the function code have to be removed and replaced by the original subtree of the *ReduceNode* once we return from the function. This is the responsibility of *removeReduction* method of abstract class *Node*.

This method is implemented in the abstract class Node. When it detects that one of the children of the node is a *ReduceNode*, it replaces the *ReduceNode* with the original subtree of the *ReduceNode*. When all children are checked, *removeReduction* is called on every child. Thus, it recursively runs from parent nodes to leaf nodes. The method is called on the node of the function body. This node is taken from the *SECDHidden* element. The figure 5.9 shows, how a call of *removeReduction* method modifies the AST.



(a) An example of nodes representing expression *((lambda (x y) (+ x y)) 10 20)* updated by the result of the expression

(b) An example of nodes representing expression *((lambda (x y) (+ x y)) 10 20)* updated by the result after *removeReduction* is called

Figure 5.9 An example of a creation of the *removeReduction* method called when returning from a function

At this point, some node pointers of *SECDelements* can point to *ReduceNodes* that are removed by the *removeReduction* method. This is the reason why another method with the same name of *removeReduction* is also present in the *SECDElement* class. It changes the pointer from the *ReduceNode* to the original subtree of the *ReduceNode* when called.

5.2.2.3 Loading values of variables to the AST

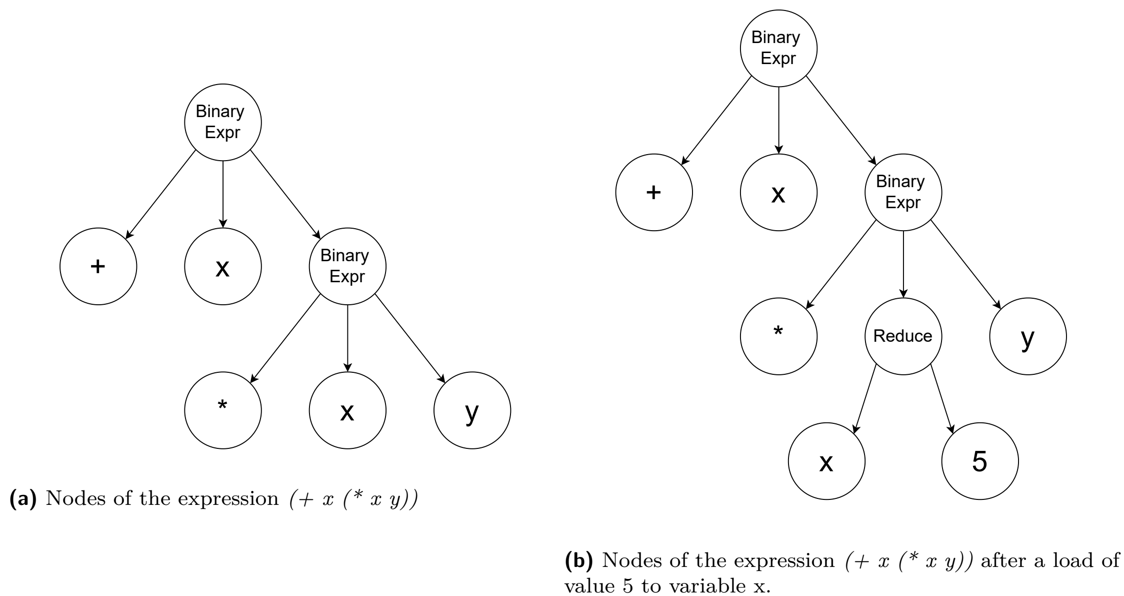
It is important that after a value is loaded to a variable, the user can see what the value of the variable is when hovering with a mouse over the variable.

This is the third case when *ReduceNode* is created. A load of a value of a variable to the AST is triggered by the LD instruction. The *InnerNode* class offers a method called *loadVariable*. The method takes a variable name and a value and recursively searches for the first occurrence of the node of the variable. If it finds such a node, it creates a *ReduceNode* with the found node as an original subtree. The reduced subtree is a node created from the received value.

If the variable is found, the method returns true, otherwise false. The method call subtrees of the current node in the same order in which the expressions they represent in the source code are evaluated. For example, tiny-lisp binary expressions evaluate the second argument first. Similarly, the node of the binary expression calls first the *loadVariable* method of the right argument subtree. Then, if the result was false, it calls the *loadVariable* method of the left argument.

If the *loadVariable* method is called on a *ReduceNode*, it searches for the variable just in the reduced subtree, so nodes cannot be updated multiple times. The figure 5.10 shows a *ReduceNode* created by a load of a value to a variable.

The method is usually not called on the whole AST, but just on a subtree. The root node of this subtree is the node of the current code register. The reason behind it is that this is the node of the function or if statement branch that is currently being evaluated. The values of variables are then loaded to the AST, each time the LD instruction is executed. However, it is important to remember that, when we return from a function, *ReduceNodes* are removed from the function.



(a) Nodes of the expression *(+ x (* x y))*

(b) Nodes of the expression *(+ x (* x y))* after a load of value 5 to variable x.

Figure 5.10 An example of a load of value 5 to variable x in the AST

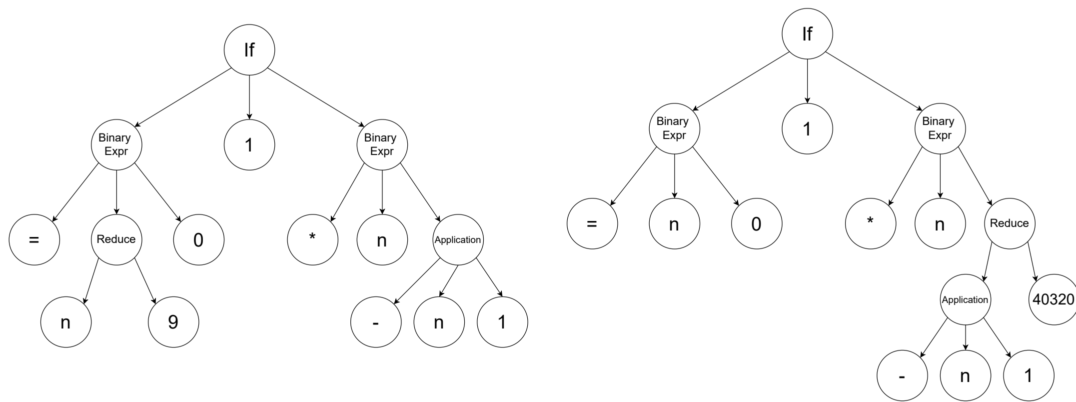
So assume that the called function was the same as the caller function, and now the caller function continues evaluating itself. Now assume that we want to load a value to a variable. It is possible that the variable already occurred in the function and the *ReduceNode* was created. Then the same function was called recursively, and when returning from it, the *ReduceNode* was removed. That means that we should call the *loadVariable* method not on the node of the function, but on the parent node of the place, where the recursive call to the function occurred. This node of the function call is stored in the *SECDHidden* element. From this point, the parent of this node will be used as the root node of the subtree, on which the *loadVariable* method will be called.

Figure 5.11 shows this situation. After returning from the recursive call to a factorial, the *ReduceNode* in the condition of the factorial is removed. The next invocation of the *loadVariable* method should load the value 9 to a node of variable n in the multiplication binary expression node. Thus, it can not be called on the whole body of the function. The parent node of the node representing the recursive call is the multiplication binary expression node. There are now two nodes of the variable n in this subtree. However, one of them is in the original subtree of a *ReduceNode*, so no value will be loaded to it. That means that the value will be correctly loaded to the node of the variable n in the multiplication binary expression.

It is important to note that this solution does not solve all possible situations that can occur. One example is a program computing Fibonacci numbers in the code listing 20. Since it calls itself recursively multiple times in the function, some interactive features of the debugger will not work properly.

5.2.3 Source code

The source code is being rendered from the AST via the visitor class *ReactTreePrinter*. The *ReactTreePrinter* class implements the *ASTVisitor* class and have a visit method for every type of AST node. Nodes are recursively visited from the top node to the leaves in the infix search. Leaf nodes create a React element based on their values, while non-leaf nodes combine React elements created by their subnodes and possibly even add some keywords that were used in the



(a) An example of AST of a body of a factorial function before a recursive call to factorial

(b) Body of the factorial after returning from a recursive factorial call

Figure 5.11 An example of AST of a body of a factorial function before a recursive call to factorial and after returning from the factorial

```
(letrec((fib
    (lambda(n)
        (if (<= n 1)
            1
            (+ (fib (- n 2)) (fib (- n 1)))))))
(fib 2))
```

20 An example of fibonacci program in tiny-lisp, where user interactions with the code might not work properly

source code. The final element is constructed by the top node, and it is rendered for the user.

Because the AST remembers only the structure of the source code and not the detailed syntax, user-defined cosmetic whitespaces are ignored.

5.2.3.1 Colouring

As discussed back in section 5.2.2, nodes may have a colour. In the case that the colour of the node is not *None*, the *ReactTreePrinter* underlines the React element generated by the node with this colour.

Depending on the *_lastIntruction* in the *Interpreter* class, the colouring of the AST and registers is performed. Colouring is a process that is performed by the *Painter* class that is a part of the front-end module. The front-end module calls the painter when it gets the state of the virtual machine from the back-end module after the evaluation step was performed. Both *SECDElement* and *Node* classes have a member variable named *colour*. This variable is of enum type *ColourType*. The variable signals in which colour elements should be either highlighted or underlined. In most situations, elements are underlined. I will be further discussed in the 5.2.3.2. If the *colour* is *None*, the element or node will not be coloured, otherwise, they will be coloured in a colour depending on the exact value of the *colour* variable.

At the start of the colouring process, the colours of all nodes in AST and all *SECDElement*s in all registers are set to *None*. Then depending on the *_lastIntruction*, those *SECDElement*s and AST nodes that would be important in the *_lastIntruction* evaluation will get assigned a colour. The figure 5.12 shows an example of a colouring of the source code and registers when

applying a function. The function is underlined by the red colour, while the arguments are in the green colour.

If the *SECDArray* have different colour than *None*, then the global boolean variable *enclosingArrayColoured* in *ReactSECDPrinter* is set to true. It is done before the recursive calls of *getElements* on its elements, and it will cause that colours of these elements will behave as *None* regardless of the value. After the calls, *enclosingArrayColoured* is set back to false. The reason why this functionality is included is to render the whole array in a single colour and prevent unnecessary confusion of the user.

LISP:
(+ 1 ((lambda (x y)
(+ x y)) 20 10))

code: [AP LDC 1 ADD]

stack: [[20 10][[LD [0 1] LD [0 0] ADD RTN][[]]]]

environment: [[]]

dump: []

Step Run

Figure 5.12 An example of colouring of the source code and registers when applying a function

5.2.3.2 Mouse interactions with source code

When the user does not interact with the code, coloured *SECDElement*s are just underlined with their colour.

The front-end module remembers the global variable called *mouseOver*. It represents the node of the AST, where the mouse currently hovers over. Each time the mouse hovers over another node, the node is assigned to this variable. The whole screen is then re-rendered.

To give the user a better idea about which parts of the bytecode correspond to some parts of the source code, whose *SECDElement*s, whose nodes are either the mouseOver node or the mouseOver node is their predecessor in the tree, are highlighted. If these elements also have a colour assigned, they are highlighted in this colour, otherwise, they are highlighted in the turquoise colour.

The figure 5.13 shows a situation when the user hovers with the mouse over the value 10 in the source code. This action highlights the value 10 in the stack register.

Assume that the user is hovering with the mouse over a function that is underlined. Since this whole function is important in the evaluation of the current instruction, it is expected that the user rather wants to highlight the bytecode of the function and not a bytecode of an expression inside the function. To support this idea, the following rule is added.

In the case that a predecessor of the node that has mouse over itself is coloured, this predecessor is assigned to the *mouseOver* variable instead.

There is yet another reason why instead of the node that the mouse is currently hovering over, a predecessor node can be chosen. It is when the expression of the node was already evaluated into a result. AST should be able to reflect this fact with a predecessor of the node being *ReduceNode*. In this case, the top *ReduceNode* predecessor is chosen instead. If a *ReduceNode* is assigned to the *mouseOver*, the elements, whose nodes point to the reduced subtree of this node, are highlighted. The figure 5.14 shows the situation when the mouse hovers over the variable

LISP:
(+ 1 ((lambda (x y)
(+ x y)) 20 10))

code: [AP LDC 1 ADD]

stack: [[20 10][[LD [0 1] LD [0 0] ADD RTN][[]]]]

environment: [[]]

dump: []

Step Run

Figure 5.13 An example of mouse interacting with number 10 in source code

y, whose value is 20. These occurrences of the value 20 are highlighted in registers. They are highlighted in the turquoise colour since they are not important in the evaluation of the current instruction.

LISP:
(+ 1 ((lambda (x y)
(+ x y)) 10 20))

code: [LD [0 0] ADD RTN]

stack: [20]

environment: [[10 20] []]

dump: [[[]][LDC 1 ADD] []]

Step Run

Figure 5.14 An example of hovering with mouse over the variable y, after the value 20 was loaded into it

Chapter 6
Evaluation

The main objective of the thesis was to create a debugger of the SECD machine in the Lambdulus system. In this section, the implemented solution will be compared to a lambda calculus evaluator.

Table 6.1 shows the number of instructions that are executed when the factorial of x is computed. The SECD debugger is compared with the already existing lambda calculus evaluator from the Lambdulus website [1]. The numbers in the first line of the table represent the argument of the factorial in the experiment. The values in the second line show a number of steps of the SECD debugger for the program computing factorial of the number in the same column in the first line. The values in the third line represent the number of steps of the lambda calculus evaluator for a program computing factorial of the number in the same column in the first line. The simplified evaluation of the lambda calculus is disabled.

The code in figure 21 was used as lambda calculus source code at the Lambdulus website.

```
R := (\ f n . ZERO n 1 (* n (f (- n 1))));
Y R x
```

21 An example of a program computing factorial of x in lambda calculus

The code in figure 22 was used for computing factorial in tiny-lisp.

```
(letrec((fact
    (lambda(n)
        (if (= n 0)
            1
            (* n (fact (- n 1)))))))
(fact x))
```

22 An example of a program computing factorial of x in tiny-lisp

We can see from the table 6.1 that the number of instructions SECD VM executes is *20 + x * 15*. As we can see, the SECD machine is a big improvement over the lambda calculus in terms of program speed, and in contrast to the lambda calculus, the amount of generated instructions depends linearly on the variable x. The lambda calculus wins for the factorial of zero, while the SECD VM wins for all other arguments of the factorial. This is probably because

the tiny-lisp letrec expression compiles into quite a lot of instructions that have to be executed, even though they are not needed, since there is no recursive call to the factorial in the end.

Table 6.1 Number of evaluation steps of (fact x)

evaluator / x	0	1	2	3	4
SECD VM	20	35	50	65	80
lambda calculus	14	47	207	1057	6247

Table 6.2 shows the number of instructions that are executed when computing the Fibonacci number of x.

The code in figure 23 was used as lambda calculus source code at the Lambdulus website.

```
R := ( f n . <= n 1 1 (+ (f (- n 2)) (f (- n 1))));
Y R x
```

23 An example of a program computing fibonacci number of x in the lambda calculus

The code in figure 24 was used for computing factorial in tiny-lisp.

```
(letrec((fib
    (lambda(n)
        (if (<= n 1)
            1
            (+ (fib (- n 2)) (fib (- n 1)))))))
(fib x))
```

24 An example of a program computing fibonacci number of x in tiny-lisp

We can see from the table that, while the Fibonacci of zero and one takes a different amount of instructions in the lambda calculus, the amount is the same in the SECD VM. This is probably due to the fact, that the speed of loading of a number in the lambda calculus depends on the size of the number. This is not a case in the SECD VM, where the number of instructions generated for an evaluation of an operator is always the same and does not depend on the size of the numbers.

Table 6.2 Number of evaluation steps of (fib x)

evaluator / x	0	1	2	3	4
SECD VM	20	20	48	76	132
lambda calculus	28	31	134	281	640

We can also see from the tables that the Fibonacci in lambda calculus is surprisingly significantly faster than the factorial in lambda calculus, while the factorial is faster in the SECD VM. This is probably because factorial produces bigger numbers than Fibonacci and that increases the number of steps for the lambda calculus as stated earlier. On the other hand, for the SECD VM, the size of numbers does not influence the number of generated instructions. The computation of Fibonacci numbers has more instructions than the computation of factorial in the SECD VM since the Fibonacci program is creating more recursive calls. For the same reason, the number of instructions generated for the Fibonacci program rises more quickly than for the factorial in the SECD VM.

Chapter 7
Conclusion

’ The thesis provides an overview of the tiny-lisp language and its features. The language is designed to be similar to the racket language, thus it is easy to understand for students of the BI-PPA course. The SECD machine is presented and its concepts and instructions are described. The thesis describes, in detail, how to compile a tiny-lisp program to the SECD bytecode. The design and implementation of the debugger are discussed.

The implementation provided in the thesis can parse the source code to the SECD bytecode and evaluate the bytecode step-by-step. It is also able to compile and evaluate macros in the compile time. Macros can use the *gensym* function that can ensure that the macro is hygienic.

The frontend part of the debugger is also provided. It shows the user both the source code and the state of the SECD machine, and actively highlight for the user important parts of the source code and the bytecode in the evaluation of the current instruction. The user can also interact with the source code, to see connections between it and the bytecode.

The thesis also offers a comparison of the performance of the SECD machine and the lambda calculus.

The debugger is easy to understand and people that want to experiment with the SECD machine can already do so. The codebase can be found on GitHub at the address https://github.com/lambdulus.

7.1 Future work

Extending tiny-lisp More primitive types such as char or double can be added to the tiny-lisp. The string data type can be changed to be a list of chars. The tiny-lisp can also be extended to support record types or structs to allow the creation of more complex data types.

Some already existing features can be improved. For example, global functions can be changed to support the use of a global function before its declaration. The *gensym* function can be implemented properly in tiny-lisp.

More elegant compiler The compiler can be redesigned to compile the source code to S-expressions first before compiling them to the SECD bytecode.

Addition of more SECD instructions It is also possible to implement some SECD optimization, such as adding the TRAP instruction that handles the interpretation of tail-recursive functions. [7] Supporting this instruction, the debugger can clearly show how tail-recursive functions can be interpreted without the overhead from a lot of calls to a function.

The debugger can support IO operations, which would require adding another frontend field for the console. It would also require extending tiny-lisp and implementing SECD IO instructions that were part of the original SECD VM but were not included in my implementation.

Improvements to the frontend As already stated in the 5.2.2.3 section, the design of AST proved to be insufficient for recursive functions. The feature should be at least partly redesigned.

The visualization of the source code and registers in the debug screen can be improved.

Another very interesting extension would be a creation of a frontend REPL evaluation environment. The user would be able to submit functions or expressions one by one and not as a whole program, which would greatly improve the user experience.

Currently, the debugger shows just the result of macro expansions. The user might want to use the debugger to ensure that their macro is correct. If it is not, they can see it, but can not use the debugger to see where the mistake is. That means that the debugger would be much more useful if the user could see the evaluation of the macro expansion step by step.

Moreover, the front-end module can be extended by providing more options to control the evaluation. There can be for example new buttons for restarting the evaluation or cancelling the evaluation. Breakpoints can also be added.

There can also be more evaluation buttons. For example, a button to evaluate a current expression or current function.

Bibliography

1. SLIACKÝ, Jan. *Implementation of lambda expressions evaluator*. Prague, CZ, 2019. Bachelor's thesis. CTU in Prague, Faculty of information techonology.
2. STEELE, Guy. *Common LISP: the language*. Elsevier, 1990. ISBN 0131524143.
3. FLATT, Matthew; PLT. *Reference: Racket* [online]. 2010. Available also from: `https://docs.racket-lang.org/`.
4. SCOWEN, Roger S. Extended BNF — A generic base standard. In: [online]. 1998. Available also from: `https://www.semanticscholar.org/paper/Extended-BNF-%E2%80%94-A-generic-base-standard-Scowen/ed89e6f749768cc4fc585e6ef406afeace436a19?p2df`.
5. ZARATE, Martin. *Meta Programming* [online]. 2012. Available also from: `http://wiki.c2.com/?MetaProgramming`. [cit. 3-May-2022].
6. MCCARTHY, John; ABRAHAMS, Paul W; EDWARDS, Daniel J; HART, Timothy P; LEVIN, Michael I. *LISP 1.5 programmer's manual*. MIT press, 1962. ISBN 9780262130110.
7. KOGGE, Peter M. *The Architecture of Symbolic Computers*. USA: McGraw-Hill, Inc., 1990. ISBN 0070355967.
8. LANDIN, P. J. The Mechanical Evaluation of Expressions. *The Computer Journal*. 1964, vol. 6, no. 4, pp. 308–320. ISSN 0010-4620. Available from DOI: `10.1093/comjnl/6.4.308`.
9. CORPORATION, MICROSOFT. *TypeScript* [online]. 2012. Available also from: `https://www.typescriptlang.org/`. [cit. 18-April-2022].
10. SIPPU, Seppo; SOISALON-SOININEN, Eljas. On LL (k) parsing. *Information and Control*. 1982, vol. 53, no. 3, pp. 141–164.
11. INC, FACEBOOK. *React.js* [online]. 2013. Available also from: `https://reactjs.org/`. [cit. 18-April-2022].
12. LIE, Håkon Wium; BOS, Bert. *Cascading style sheets, level 1* [online]. Recommendation RECCSS1, World Wide Web Consortium, 1996. Available also from: `http://lia.deis.unibo.it/courses/2004-2005/LABTEC-LA-CE/downloads/REC-CSS1-19990111.pdf`. [cit. 16-April-2022].

Contents of the enclosed flash disk

```
readme.md ........................ the file with flash disk contents description
src ............................................ the directory of source codes
  core ..................... the directory of source codes of core module
    dist ............... the directory of compiled JavaScript source code
  frontend ............. the directory of source codes of front-end module
    build .......................... the directory of built static web-site
  thesis .................. the directory of LaTeX source codes of the thesis
text .............................................. the thesis text directory
  thesis.pdf ................................ the thesis text in PDF format
```