KTH Computer Science
and Communication

A Lisp compiler for the JVM

or
How to implement dynamic programming languages on top of the JVM
or
Lack of JVM TCO considered annoying

ANTON KINDESTAM
<ANTONKI@KTH.SE>

Bachelor's Thesis at NADA
Supervisor: Mads Dam
Examiner: Mårten Björkman

Sammanfattning

Att implementera dynamiska och mestadels funktionella programmeringsspråk på miljöer som JVM är allt mer i tiden. Språk såsom Clojure, Scala eller Python. För att åstadkomma duglig prestanda och Java interoperation bör ett sådant språk helst kompileras. Denna essä handlar om tekniker som kan användas för att implementera dynamiska, funktionella programmeringsspråk på JVM:en med speciallt focus på Lisp och Scheme. En implementation av en liten Lisp-kompilator har genomförts för att illustrera några av dessa tekniker.

Abstract

Implementing dynamic, mostly functional, languages on top of an environment such as the JVM is getting ever more popular. Languages such as Clojure, Scala, or Python. To achieve reasonable performance and Java interoperability such a language usually needs to be compiled. This essay features techniques for implementing dynamic and functional languages on the JVM with a focus on Lisp and Scheme, as well as an implementation of a small Lisp compiler demonstrating some of these techniques.

Contents

Part I

Report

Chapter 1

Introduction

A compiler is a program that transforms code written in a source programming language to a target programming language.

The Java Virtual Machine, or JVM, is an ever more popular target for language designers. Languages like Groovy, Scala and Clojure all implemented on the JVM, have recently been gaining attention in particular due to their interoperability with Java. There have even been ports of popular dynamic languages such as Python and Ruby, Jython and JRuby respectively, to the JVM. Interestingly the above-mentioned Clojure is an implementation of modern Lisp dialect.

This thesis aims to investigate what goes into creating a compiler for a dynamic language that compiles to the JVM using a small and simple Lisp dialect as the source language in a hands-on approach.

1.1 Why Lisp?

Despite, or perhaps because of, its age Lisp shares a lot of common ground, and thus implementation issues, with more recent and popular dynamic programming languages such as Python, Ruby or Clojure. The latter is in fact a modern dialect of Lisp operating on top of the JVM.

Lisp is well suited for a project like this in particular due to its ease of implementation. The inherent ability of the language to do a lot given very little is going to make possible to compile interesting programs without having the compiler support the entire language (which is fairly small anyway). There is no need to spend time implementing a parser since one is already available from the LJSP interpreted environment. Writing the compiler in and for Lisp, and in this case even in LJSP itself, becomes very efficient since Lisp code is represented using Lisp data structures so the compiler can easily be built as a dispatch-on-type set of recursive functions.

1.2 Why JVM?

"Attracted by a generally available, machine-independent platform, implementors of other languages are turning to the Java virtual machine as a delivery vehicle for their languages." [JVMSpec] (§1.2).

Other advantages include that the JVM includes native garbage collection giving more time for actually implementing the language and not a garbage collector, which is a big investment in development time.

Disadvantages include inefficiencies and having to deal with how the JVM is closely built around Java, with no inherent support for first-class functions nor the call-stack manipulations typically used to implement tail-call optimization.

Chapter 2

Background

This section explains the choices of source language its feature set as well as some of the vocabulary used in this thesis. The benefits, as well as the drawbacks, of targeting a virtual machine such as the JVM are explored.

2.1 Definitions

Term	Definition
Lisp	***LIS**t **P**rocessing* A family of dynamic programming languages commonly programmed using a functional programming style, but also capable of imperative programming for side-effect.
Functional Programming (FP)	A programing style focusing chiefly on function application and side-effect free computing.
Imperative Programming	A programming style where computation is expressed in terms of statements that change program state.
Virtual Machine (VM)	A computer model implemented in software.
JVM	*Java Virtual Machine* A VM originally implemented for the Java programming language. Java (and more recently a whole flock of different JVM-based languages such as Clojure) compiles to Java Byte Code which the JVM then executes. Since there are implementations of the JVM for different processor architectures and environments the same code runs on portably across many architectures and operating systems without the need for recompiling.
Java Byte Code	The virtual instruction set supported by the JVM.
Jasmin	A program capable of converting a simple text representation of Java Byte Code instructions to actual Java Byte Code. The same role an *Assembler* performs for a regular (usually implemented in hardware) processor architecture.
Source Language	The language a compiler reads as its input.
Implementation Language	The language the compiler is implemented in.
Target Language	The language a compiler outputs.
REPL	Read-Eval-Print-Loop: Traditional name for the Lisp interactive command line
Bootstrapping	The art of pulling oneself up by ones own bootstraps. In the context of compilers this usually refers to the act of writing a compiler capable of compiling itself.

Fixed Arity	Pertaining to a function; a function that accepts only a fixed number of arguments.
Variable Arity	Pertaining to a function; a function that accepts a variable amount of arguments.

2.2 Prior Work

Before starting this thesis the author had implemented a small interpreter and Lisp system for Java called LJSP, for silly reasons[1]. This system will be used as a base, as well as implementation language, for the compiler and classes implemented for the interpreter will be able to be conveniently reused for implementing the compiler, with only minimal changes to them neccessary.

Tricky issues, like mixed-type arithmetic, is already handled in these classes giving more time to work on the core parts of the compiler.

The interpreter features an interface to Java (currently somewhat quirky and limited but still useful) using Javas reflection features. This can, among other things, be used to load generated class files into the runtime after compilation.

2.3 Preliminary Issues

2.3.1 Scoping

This section explains the different variable scoping terms used in this thesis.

Useful terms when speaking about variable scoping [CLtL2] (§3):

Scope The textual portion of a program during which a variable may be referenced.

Extent The interval of time during which references may occur.

Lexical Scoping

A *lexically scoped* binding can only be referenced within the body of some construct enclosing part of the program. The scope extends to the bodies of enclosing constructs within the outer body, allowing for instance nested functions to access, and mutate, bindings introduced by the function that created them.

The bindings are said to be of *indefinite extent*, that is they can be kept as long as something is using them. If a function closing over a binding is returned that binding will be kept so long as there is an active reference to that function, or *closure*.

Example (pseudo-code):

[1] Anything that has something to do with Java ought to have a "J" in the name, and the interchangeability of the letters "i" and "j" in old alphabets made this silly, and unpronouncable using modern orthographical rules, substitution obvious.

```
function foo(x):-
    function bar(y):-
        return y + x
    return bar(x) + x
```

The free variable `x` in `bar` is resolved to the `x` introduced by `foo`. Running `foo(t)` will thus yield `t+t+t`.

Example with mutation:

```
function make-incrementer(x):-
    function inc(y):-
        x = x + y
        return x
    return inc
...

>> a = make-incrementer(2)
<closure inc 1>
>> a(2)
4
>> a(1)
5
>> b = make-incrementer(123)
<closure inc 2>
>> b(5)
128
>> a(6)
11
```

Erratic example:

```
function foobar(a):-
    function baz(b):-
        return b + a
    return baz(a) + b   ; b not defined in this scope
```

This is an error for lexically scoped `a` and `b`. Since `b`s scope only extends throughout the body of `baz`, however the variable `a` is reachable in both `foobar` and `baz`.

Static Scoping

While often used synonymously with lexical scoping *static scoping*, as used in this essay, will refer to the subset of lexically scoped variable bindings that are never captured by any function other than the function in which the bindings were defined. That is the variables scope exists only in the body of the function that established the variable binding, and not in the bodies of any nested functions. This is somewhat similar to the C model of variable scope, disregarding for a while that C typically lacks nested functions.

Example:

```
function foo(x):-
    function bar(x):-
        return x*3
    return bar(x) + 2
```

is valid for a statically scoped `x`, since all `x`:s are local to their defining functions.

```
function foo(w):-
    function bar():-
        return w*3
    return w + bar()
```

Would however result in an error since the free variable `w` is not in scope in `bar`s environment, where as it would be with true lexical scoping.

This is the only scoping supported by the example LJSP compiler built for this thesis (but further extension of the compiler is planned, see section 4.3 The future? on page 39).

Dynamic Scoping

Dynamically scoped variables are said to have *indefinite scope*, that is they can be referenced anywhere in the code, and *dynamic extent*. The latter means that they are referenceable between establishment and explicit disestablishment, at runtime, thus mirroring the actual runtime call stack.

In fact one convenient way of thinking of dynamically bound variables are as stacks with the most recent binding at the top.

Example (all variables are dynamically bound):

```
  function bar(b):-
      print(a) ; can access a here if called from foo
      print(b) ; the b here will however be 12 when
               ; called from foo, and not 18, since that
               ; b has been shadowed by the b in the arguments to bar

  function foo(a, b):-
      bar(12)

...

  >> foo(123, 18)
  123
  12
  <void>
  >> bar(23) ; this will fail since a is not defined
  <somefail>
```

Some implementations of dynamic scoping, such as the one used by the LJSP interpreter, will default to `nil` when accessing a non-defined variable thus failing in a much more subtle way for the last call to `bar`.

This is the only kind of scoping available in the LJSP interpreter.

Interestingly this kind of semantic dichotomy, which the LJSP interpreter and compiler currently displays, between the compiler implementation (static scoping only/by default) and interpreter implementation (dynamic scoping only/by default) is typical of old Lisp implementations. This is usually so since implementation-wise dynamically scoped variables are easier to implement more efficiently in an interpreter, while the statically scoped variables are more easily compiled to efficient code.

2.3.2 About Lisp

This section briefly presents some basic Lisp data types, special operators, common constructs and functions used throughout the essay to aid the reader unfamiliar with Lisp.

A feature that sets Lisp apart from most programming languages is its homoiconicity, the fact that Lisp code is represented by Lisp data structures. Code is data. `(+ a 4)` is a list that contains the symbols `+` and `a` as well as the number 4. Put that list in a different context however, that is evaluate it, and it means: call the function `+` with the arguments variable `a` and literal 4.

This also makes writing a compiler or interpreter for Lisp in Lisp itself relatively simple since it is merely the matter of writing a program working on the normal Lisp data structures, interpreting them as code.

Macros

This homoiconicity also allows for something relatively unusual outside of Lisp programming which is macro programming. That is most Lisp environments allow for the inclusion of Lisp programs that generate Lisp code. This is feasable since Lisp code is Lisp data. These receive Lisp data as their input, are usually run at compile-time and the code at the call-site of the macro is replaced with the code that the macro generated. Macros can be used to define many constructs of the language in the language itself without special compiler support. It naturally also allows definition of new constructs specialized for a certain specific problem.

To read more about Lisp macros in the context of Common Lisp [Graham] (§7) is recommended.

Data types

We now move on to present some Lisp data types.

Lists The single most important data structure in Lisp is the list. Lisp is after all an acronym of *list processing*. A list in a traditional Lisp is a regular singly-linked

list. The nodes in a Lisp linked list are traditionally known as *cons* cells or conses. The two-argument function used to construct them is consequently known as `cons`.

```
(cons 1 2) ⇒ (1 . 2)
```

cons cells typically contain two pointers to Lisp objects. However many Lisp implementations store some objects such as integer values directly in the pointer fields to be more efficient. Interestingly the next-pointer, or second field, is not required to point to another cons cell but can point to any Lisp object. The end of list-marker is usually written `()` and is in most traditional Lisp environments synonymous with `nil` which is the Lisp equivalent of the **`null`** pointer or **`null`** reference. Some more modern variants of Lisp such as Scheme instead distinguish between `()` and `nil` [R^{5}RS].

The value-field of a cons cell, or the head of the list, is accessed using the `car` function. The next-pointer-field, effectively the tail of the list, is accessed using the `cdr` function. Thus these are also known as the *car* and *cdr* fields. These seemingly strange names are a holdover from some low-level details of the machine on which the first Lisp implementation was made. `car` stands for contents of the address part of register and `cdr` stands for contents of the decrement part of register [McCarthy60] (§4) (p. 28).

In most Lisp dialects the car and cdr of a cons cell can be set using the procedure `rplaca`, for replace car, and `rplacd`, for replace cdr, respectively. In some Lisp variants they might be known by other names or, in the case of some side-effect-free, dialect not exist at all.

Building some lists using `cons`:

```
(cons 1 (cons 2 ())) ⇒ (1 . (2 . ())) ⇔ (1 2)
(cons 4 (cons 3 4))  ⇒ (4 . (3 . 4))  ⇔ (4 3 . 4)
```

Note the more convenient textual representation of lists furthest to the right. Note in the latter example how Lisp traditionally allows for what is called improper lists, that is lists that don't end in with the list terminator `()`. This is due to how the second field of the cons cell can point to any lisp object.

Symbols Lisp as a language was originally developed for symbolic computation. A symbol is a uniquely named object written as a string. Symbols are used either as data items in symbolic processing or as a variable name when evaluated as code.

Symbols are typically represented using interned strings. A table of all symbols is kept. Whenever a new symbol is created, either by the parser or using the function `intern` which accepts one string as its argument, the table is first searched for an existing symbol by that name and returns it if found, otherwise a new symbol object is created, inserted in the table and then returned. This allows, among other things, for efficient equality checks between symbols amounting to a simple pointer check since all references to a symbol with the same name will point to the same object.

Boolean values Most traditional Lisp implementations have no specific boolean type and represent falsehood as `nil` and truth as everything that isn't `nil`. LJSP behaves accordingly. By tradition functions that return a boolean value return the symbol `t` for truth.

A notable exception is Scheme which has a special boolean type [R^{5}RS] (§6.3.1).

Numbers It can be argued that lists together with symbols is all that is needed for a language to be called Lisp, however numeric computations becomes very inconvenient and slow without some specialized data types for numbers.

LJSP supports *fixnums*, for integers between -2^{63} and $2^{63}-1$ inclusive, *bignums* for arbitrarily large integers and *flonums* for floating point values. LJSP automatically promotes fixnums to bignums on integer overflow. Thus the more efficient fixnums can be used as long as the value is in range without loosing precision.

Some Lisp languages such as Common Lisp even come with fully-integrated support for complex numbers as well as rationals [CLtL2] (§12).

Notable is how Lisp is without special infix operators to operate on numbers instead using regular functions.

Some LJSP functions that operate on numbers:

`(+ a b)` $a + b$

`(- a b)` $a - b$

`(* a b)` $a * b$

`(/ a b)` a/b

`(= a b)` Returns the symbol `t` if `a` and `b` compare equal and returns `nil` otherwise.

`(> a b)` As above but if `a` is larger than `b`.

Example: `(* (+ a (/ w 2)) 3)` can be more conventionally written as:

$$\left(a + \frac{w}{2}\right) \cdot 3$$

Arrays Many Lisp dialects also provide arrays as an alternative container to linked lists when constant access time and other properties of arrays are of greater importance than the flexibility of linked lists. In LJSP array elements can be accessed using the function `aref` and they can be set using the function `aset`.

Example:

```
(aref #(5 4 3) 1) ⇒ 4
(aset array 3 1337) ; array[3] = 1337
```

Strings While it is possible to implement strings as linked lists of symbols or numbers most modern Lisp implementations include some sort of string data type for convenience and optimization. Strings are usually represented as the subset of arrays which consist only of, or are constrained to, containing only objects of character type.

In LJSP strings are written as `"florp"` and characters, of which strings are composed, are written `#\e`. Strings can be handled using `aref` and `aset` since they are a special form of an array.

Special forms and other common constructs

This subsection explores some Lisp semantics, some built-in functions used in this essay but not yet mentioned, and in particular some constructs with special semantic importance called *special forms*; so called since they are essentially special cases in a Lisp compiler or interpreter of what otherwise would semantically be a function call.

Worth to note at this point is that Lisp doesn't have statements in the usual sense. Everything is an expression and has a return value when evaluated [AIM443] (p. 2) [CLtL2] [R^{5}RS]. For instance the closest approximation of a Lisp if expression in C or Java is the ternary operator.

Function call When a list is evaluated the first element is interpreted as a function, unless it is a symbol whose name corresponds to a special form, and the rest of the list is interpreted as the arguments. The arguments are evaluated and the values gotten from evaluating the arguments are passed to the function. Thus `(foo + e)` means run the function `foo` on the variables `+` and `e`, or in more common notation: `foo(+,e)`.

An exception to this evaluation rule are the special forms. Whenever the first element of a list is a symbol and that symbol matches the name of a special form the list is not interpreted as a function call but instead along the rules of that special forms. Special forms are, in a way, the reserved keywords of Lisp.

Special form `lambda` Whenever the compiler or interpreter comes across a list starting with the symbol `lambda` the list is interpreted as an anonymous function. The second element in the list is interpreted as the formal argument list with the rest of the list being the function body.

The function body is a list of expressions evaluated in order. The value resulting from evaluating the last expression is used as the return value of the function while eventual preceding expressions are merely evaluated for side-effect. Thus in Lisp code written in a functional programming style function bodies usually consist of only one expression.

Examples:

```
;; an anonymous function of two arguments
```

```
(lambda (a b) ...)

;; calling an anonymous function
((lambda (a) (+ a 2)) 4) ⇒ 6

;; a becomes bound to 1, b to 2 and rst to (3 4)
((lambda (a  b . rst) rst) 1 2 3) ⇒ (3 4)

;; all becomes bound to (44 3)
((lambda all (cons 'a all)) 44 (+ 1 2)) ⇒ (a 44 3)
```

Note how anonymous functions can be called without binding them to any variable name. Note in the third example how an improper list specifies a rest-parameter where superfluous parameters are gathered to a list. The example thus accepts two or more arguments. Also note in the last example how specifying a symbol instead of a list as the formal argument list gathers all arguments passed to the function as a list. This example accepts any number of arguments, including none. LJSP thus handles formal argument lists similarily to Scheme [R^{5}RS] (§4.1.4).

Special form `nlambda` Due to the current LJSP compiler only supporting static scoping a special form for self-recursive functions becomes neccessary. The function itself needs to be bound to some name throughout the body of the function.

Enter `nlambda` (mmnemonic: named lambda)! `nlambda` is like a regular `lambda` with the addition of a name-parameter to which the function itself is bound for the extent of its body.

Example:

```
(nlambda foo (a b) ...)
```

This function takes two arguments `a` and `b` and the identifier `foo` is bound inside the function body.

Special form `quote` When evaluating a list as code and the first element is the symbol `quote` the rest of the list is not evaluated but the enclosed data structure is returned as is. Thus evaluating `(quote (+ 1 2))` yields the list `(+ 1 2)` as the result and not `3`.

Since usage of `quote` is so ubiquitous typical Lisp readers, or Lisp parsers, have special syntax such that, for instance `'foo == (quote foo)` for convenience [R^{5}RS] (§4.1.2) [CLtL2] (§1.2.7).

Special form `if` As mentioned before the Lisp `if` expression, or special form, is analogous to the C or Java ternary operator in that it always returns a value.

Whenever the compiler or interpreter comes across a list beginning with the symbol `if` it evaluates the expression that is in the second element of the list. If

that expression evaluated to non-nil, or true, the third element in the list is evaluated and the result returned as the value of the if expression. If the conditional expression on the other hand evaluated to nil, that is false, the fourth element in the list is evaluated and the result returned. If the conditional expression evaluated to nil and the if expression doesn't have a fourth element nothing is evaluated and nil is returned.

Since only either the third or the fourth element are evaluted the Lisp if expression may also be used for side-effect, or rather lack of it.

Examples:

```
(if 234 (+ 2 3) (- 2 3)) ⇒ 5
(if nil (+ 2 3) (- 2 3)) ⇒ -1
(if t (print 'eeyup) (print 'nnope)) ; prints only eeyup
(if nil (+ 33 44)) ⇒ nil
```

The construct `let` `let` introduces new variables and a new scope. Similar to the braces of C or Java. `let` can usually be implemented in terms of `lambda` but many times is implemented as part of the interpreter or compiler as a special form for performance or other reasons. In LJSP `let` is implemented as a macro that expands to a function call to an anonymous function.

`let` syntax:

```
(let (<binding>*) <body>)
```

<binding> is either (*<varname>* *<expression>*) which binds *<varname>* to the value of evaluating *<expression>* or just *<varname>* which binds *<varname>* to `nil`.

<body> is a sequence of expressions. Just like the body of a `lambda` expression they are evaluated in order and the value of evaluating the last expression is returned as the value of the `let` expression.

```
(let ((a (+ 1 1)) (b 3))
  (print a)
  (+ a b))
⇒
;; prints 2
5       ; evaluates to 5
```

is equivalent to

```
((lambda (a b) (print a) (+ a b)) (+ 1 1) 3)
```

or similar to the pseudo-C-code:

```
...
{
   int a = 2;
   int b = 3;
```

```
    print(a);
    ... = a + b;
}
...
```

The function `eq?` The built-in function `eq?` is equivalent to a pointer compare in C or the comparison operator in Java as used on object references. What it tests is if two references are referencing the same object.

The function `set` and the macro `setq` The built-in function `set` takes two arguments. The first argument must be a symbol and the second can be any object. The value the symbol is currently bound to is set to the value received in the second argument:

```
(set (quote foo) 23) ; set variable foo to 23
```

Note how foo must by quoted so that it is not evaluated. The quoting ensures that the actual symbol `foo` is sent to the function.

Since the most common use of `set` requires use of `quote` this can quickly become cumbersome. Thus most Lisp environments provide a macro traditionally named `setq` which expands to a call to `set` with `quote` around the first argument:

```
(setq bar 33) == (set (quote bar) 33)
```

`defun` `(defun` *<name>* *<arglist>* *<body>*`)` is a macro, or sometimes a special form, that defines a function globally as the name *<name>*. The function that is defined by this construct is equivalent to `(lambda` *<arglist>* *<body>*`)`.

`defvar` `(defvar` *<varname>* *<init>*`)` defines dynamically scoped global variables into existance and optionally binds them to an initial value. Thus *<init>* may or may not be present.

2.3.3 Tail-call optimization

Functional languages often eschew iteration constructs in favor of plain recursion [AIM353]. Recursion has one disadvantage however, it uses up stack frames and can lead to stack overflows given indefinite recursion. Tail-calls are a special case of recursion that lends itself to optimization allowing for boundless recursion.

What is a Tail-Call?

Whenever the last action, or the expression in tail position, in a function is to call another function this is a tail-call. For meaningful results the true and false expressions respectively of an if expression in tail position also need to be defined

inductively as themselves being in tail position [R^{5}RS] (§3.5). This makes sense since an if expression chooses what block of code will be the last one in this case.

What is tail-call optimization (TCO)?

Whenever the last action of a function is but to return the result of another function there is no longer any need to keep the stack frame of the calling function, since the variables therein will inevitably be referenced no longer. By eliminating the call instruction of a tail-call, instead replacing it by a goto instruction, allows for what syntactically is a function call, in tail position, while saving stack space.

Consider the following function:

```
(defun foo (a)
  (bar (+ a 2)))
```

Which might be compiled to something like (pseudo-assembly, RISC-style):

```
foo:
  pop a              ; receive argument a on stack
  add temp, a, 2     ; (+ a 2) -> temp register

  push ret-reg       ; save our return address on stack, so it doesn't
                     ; get clobbered by the call to bar

  push temp          ; argument to bar

  call bar           ; run (bar temp) -> result to result-reg.
                     ; ret-reg is set to program counter.

  pop ret-reg        ; restore our return address
  goto-reg ret-reg   ; return to address in ret-reg. the return instruction.
                     ; result-reg has ben set by bar,
                     ; this is what constitues the return value.
```

Replacing the `call` instruction with a `goto` one obtains:

```
foo:
  pop a
  add temp, a, 2
  push temp         ; argument to bar
  goto bar          ; transfer control to bar. which receives the
                    ; argument. ret-reg remains unchanged. bar sets
                    ; result-reg and then immediately returns to
                    ; the caller of foo (the value of ret-reg)
```

No longer is it neccessary to use the stack to save the return address. Leaving `ret-reg` untouched will have `bar` jump directly to `foo`s caller. The argument to

`bar`, pushed on the stack, is popped inside `bar`, keeping the stack from growing at all. Any stack usage, for spilled registers or the like, inside `foo` would have to be popped before the `goto`. Even if `bar` uses the stack for some temporaries the stack size would remain bounded. This is of course given that `bar` has also had its own tail-calls, to other functions as well as to itself, eliminated. [AIM443]

While eliminating tail-calls can be thought of as an optional optimization in many languages, for example GCC optimizes tail-calls for the C language which by no means requires it [gcc], for many (mostly) functional programming languages *proper tail-recursion* is a requirement of the language [R^{5}RS] (§3.5).

This is so since those languages might either have a few iteration constructs, but whose usage is generally considered non-idiomatic or non-functional in nature[2], or completely lack regular iteration statements relying completely on recursion for iterative tasks, perhaps even implementing some iterative constructs by way of recursion as library functions/syntax not in the core language [AIM353] (§1.2). LJSP, which lacks any regular iteration constructs, has this done in its core library implementing (currently a subset of the functionality of) `dolist` and `dotimes` from Common Lisp [CLtL2] (§7.8.3) by way of macros and recursive higher-order functions.

One of the big issues this thesis will tackle is how to implement TCO on top of the JVM. The JVM, being a virtual machine optimized for Java specifically, has no way of jumping between subroutines like above. In fact it completely lacks regular subroutines[3] and has only methods associated with either classes (`static` methods) or objects, since this is all that Java needs.

2.3.4 Bootstrapping

A compiler that is capable of compiling itself is also capable of freeing itself from the original environment. A compiler that has been bootstrapped is sometimes referred to as self-hosting in the sense that to generate a new version of the compiler program no other "host" system but the compiler program is required.

The extent to which the compiler can free itself of the original environment is not necessarily the same for every compiler. This holds true for dynamic programming languages especially, for which the runtime environment and the environment of the compiler need not be, and usually is not, disjoint. This is even more complicated on top of an environment such as the JVM. E.g. the case presented in this thesis still depends on some data structures originally defined in Java and, if made to bootstrap while still directly using the defined-in-Java data structures inherited from the LJSP interpreter, can't be considered fully self-hosting. Additional work

[2] An example would be the `do`-loop in Scheme, where recursion as a means of iteration is considered more idiomatic.

[3] This is not entirely true, the JVM has a form of subroutines that are local to a method used for compiling the finally-clause of a try-catch-finally exception-handling construct [JVMSpec] (chapter 6 operations `jsr`, `jsr_w` and `ret` as well as section 7.13).

on the compiler to give it the ability define the data structures independently of Java could, however, result in a truly independent compiler.

2.4 Problem statement

Implement a compiler for a, possibly extended, subset of the Lisp language LJSP.

The compiler shall be written itself in LJSP in a manner that will make it possible to, with further work than presented in this thesis, eventually bootstrap. Due to time constraints and the focus of this thesis the compiler will only be worked towards bootstrapping as a long-term goal rather than actually bootstrapping.

The compiler shall be able to compile a naive implementation of a recursive function computing the fibonacci series, as well as a more efficient tail-recursive implementation.

The compiler should exhibit *proper tail-recursion* as defined by [R^5RS], if at all possible.

2.5 Test cases

The goal is to run these two test cases with different parameters `n` and note the time it takes for the computations to finish. First they will be run interpretatively using the existing LJSP interpreter, and then they will be compiled using the LJSP compiler written for this thesis. The resulting compiled code will then be run and clocked, as well as verified to compute the same results as when the code is interpreted. Then the execution speed of the interepreted vs. the compiled versions will be compared to see if there has been any execution speed improvements with compilation.

These are the test cases:

```
;; Recursive fibonacci
(nlambda fib (n)
  (if (= n 0)
      0
      (if (= n 1)
          1
          (+ (fib (- n 1))
             (fib (- n 2))))))

;; Tail-recursive/iterative fibonacci
(lambda (n)
  ((nlambda calc-fib (n a b)
            (if (= n 0)
                a
                (calc-fib (- n 1) b (+ a b))))
   n 0 1))
```

They both compute the n:th number of the fibonacci sequence. They use the naive recursive definition (time complexity: $O(2^n)$) and a tail-recursive, or iterative if you prefer, version (time complexity: $O(n)$) respectively.

The first test, due to its time complexity and amount of function calls, is a very good performance test for small integer arithmetics and non-tail-recursive function calls, and will likely be the case where the compiler is expected to excel, since the interpreter carries significant function call overhead in particular due to how variable bindings are handled (but the full discussion of that is better suited to a paper on interpreter internals).

The second test is a good test of self-tail-recursion and is expected to be vastly faster both interpreted as well as compiled. Due to its speed it can realistically be tested with `n` big enough for a bignum result. This test probably won't have the interpreter at an as big disadvantage in part due to the interpreter being very efficient at handling tail-recursion yet a significant improvement is still expected of the compiled version.

While these two tests alone make great benchmarks due to their heavy use of function calls, and in the first test also heavy stack usage, they are not quite adequate as a test suite. Had more time been available a test suite to test the compiler for correctness would have been ideal. Since a complex program like a compiler is very prone to bugs a test suite helps immensly in finding and correcting bugs and mis-features.

Chapter 3

Methods

This chapter deals with the implementation techniques used, and not used, and (possibly) slated to be used for the LJSP compiler some time in the future. Most techniques presented are also useful in the general context of implementing dynamic languages on the JVM since some of the issues this section tackles, like first-class functions and closures, are had in common with Lisp.

3.1 General

3.1.1 Overview of compilation

General description of compiler passes in a Lisp or Lisp-like compiler on the JVM or similar environment [Kawa] (§7).

Reading
: Reads the input from a a file, string, or the interactive prompt (REPL). Parses the indata to LJSP data structures.

Semantic Analysis
: Macro expansion takes place. Lexical analysis of free variables is performed, and closures are annotated. Different sorts of rewrites are performed.

  The LJSP compiler currently lacks this step, but it is planned and will be neccessary for implementing some of the more advanced features discussed later in this chapter.

Code Generation
: Run on the resulting code form the semantic analysis. Takes LJSP datastructures and dispatches recursively, based on type and structure, on it generating bytecode fit for feeding in to Jasmin.

Assembly
: The output of the code generator is run through jasmin producing a Java class file.

Loading
The generated Java class file is loaded into the JVM, an object is instantiated and bound to a variable so the function may be called.

3.2 Functions and function application

Java doesn't have functions as first-class values. First-class functions is a prominent feature of any functional language and LJSP is just like Scheme in this regard.

Achieving this in Java is pretty straight-forward however: A `Procedure` class can be created for representing function, or procedure[1], objects. Then by subclassing and overriding a virtual method `run`[2] to contain code generated from the function body function objects in the Scheme sense becomes possible, by way of instantiating such a subclass and passing it around.

A `Procedure` class was already available from the LJSP interpreter used for, among other things, defining the various built-in functions. An advantage of using this already-available class is ready interoperability with the interpreter. That is the compiled functions can be run directly from the interpreters REPL (Read-Eval-Print-Loop or simply put a sort of command line) and called from interpreted functions with no changes to the interpreter.

Example of what a `Procedure` class might look like:

```
abstract class Procedure extends LispObject {
    ...
    public abstract LispObject run(LispObject[] args);
}
```

Using this class the primitive function `car` might be implemented in pure Java as follows [3]:

```
class car extends Procedure {
    public LispObject run(LispObject[] o) {
        return ((o[0]) == null) ? null : ((Cons)o[0]).car;
    }
}
```

[1] Which might be better nomenclature since they are not functions in the strict mathematical sense, since they can have side-effects. For instance Scheme prefers this nomenclature. However primarily "function" will be used throughout this thesis (with a few obvious exceptions).

[2] In most other literature concerning Lisp on the JVM this method is named `apply` but due to implementation details of the LJSP interpreter this name was not available.

[3] Notably omitted: Checks to ensure that the correct amount of arguments is passed. At the time of writing this is implemented in a fashion optimized for ease of implementation of primitive functions exported from the interpreter (using constructor arguments to Procedure to tell it how to do such checking). This is however slated to change to benefit the compiled version which preferrably compiles in a hard-coded equivalent of such checks. Currently compiled code simply ignores receiving too many arguments.

Variable arity functions

The `run` method takes as its argument an array of `LispObject`s and can thus support any number of arguments, including functions with variable arity, at the expense of a slightly clumsy calling convention. This is neccessary since there is no support for variable arity methods in the JVM, the variable arity methods in Java merely being syntactic sugar for passing extra arguments in an array [JLS3] (§15.12.4.2).

Due to how functions are first-class values in this language the caller may in many situations have no idea of what the actual parameter list of the function it calls looks like. The caller cannot know at compile-time how many arguments the function expects nor whether it expects to receive a rest-parameter list from say the 3rd argument onward. Perhaps the function expects only a rest-parameter list. This is in contrast with Java where the signature of the method being called is always known, and has to be known, at compile-time. If the caller was responsible for creating the list for eventual rest-parameters, like in Java [JLS3] (§15.12.4.2), every function call site would have to include multiple cases and select one at runtime. This would quickly become messy and inefficient.

The sensible solution is thus to make it the responsibility of the callee to create the linked list structure needed for any rest-parameter, and have the caller send along arguments in the same way it would for a fixed-arity procedure. This is somewhat similar to how variable arity procedures calls are handled by most C compilers, however at a much higher level of abstraction where the callee receives meta-information such as the number of arguments passed as well as their types (unsurprisingly given the dynamically typed nature of Lisp) implicitly.

An optimization for calling fixed-arity functions

While having `run` recieve an array of `LispObject`s allows for all functions to be represented using the same Java method signature it is rather ineffecient, and a toll on the garbage collector, to construct an array to hold the arguments for every function call.

Since most functions in practice are of just a small fixed number of arguments, usually less than 4, the price of the generality this affords is pretty high.

However using this fact an optimization becomes possible. All functions of fixed arity below some arbitrary finite number $K + 1$ can be compiled to a Java method of the same fixed arity: `run`x where x is the number of arguments, overloading a method defined in the `Procedure` superclass. K is picked by the compiler implementor as the largest argument count a function can have and still be called without the caller constructing a `LispObject` array. The need for the `Procedure` superclass to define all functions that can be overloaded is what bounds this technique to a finite K.

For variable arity functions and functions with an amount of arguments greater than K the `runN` method, accepting a `LispObject` array with arguments, is over-

loaded.

Modified `Procedure` class:

```
abstract class Procedure extends LispObject {
    public abstract LispObject runN(LispObject[] args);
    public abstract LispObject run0();
    public abstract LispObject run1(LispObject arg1);
    public abstract LispObject run2(LispObject arg1, LispObject arg2);
    ...
}
```

This continues up to the method `run`K.

`car`, taking exactly one argument, could then be constructed as follows:

```
class car extends Procedure {
    public LispObject runN(LispObject[] o) {
        if (o.length != 1)
            throw new WrongArguments();
        return this.run1(o[0]);
    }
    public LispObject run0() {
        throw new WrongArguments();
    }
    public LispObject run1(LispObject arg) {
        return (arg == null) ? null : ((Cons)arg).car;
    }
    public LispObject run2(LispObject arg1, LispObject arg2) {
        throw new WrongArguments();
    }
    ...
}
```

Note how the other `run` methods are still overloaded to signal an argument count error. `runN` is overloaded to pass on the arguments received in the array to the `run1` which is where `car` is actually implemented, unless the count of arguments received is bad. Thus this implementation of `car` can be invoked using any `run` method but only successfully with either `run1` and a single argument or `runN` with an array of length 1.

An example of how a variable arity procedure might be compiled:

```
class foo extends Procedure {
    public LispObject runN(LispObject[] args) {
      // Do stuff with args. If applicable check that enough
      // arguments were received.
      return some_result;
    }
```

```
    public LispObject run0() {
        return this.runN(new LispObject[]{});
    }
    ...
    public LispObject run2(LispObject arg1, LispObject arg2) {
        return this.runN(new LispObject[]{arg1, arg2});
    }
    ...
}
```

This example instead has all `run` methods except `runN` call `runN`. This function is thus also invokeable using any of the `run` methods.

This makes it possible for compiled code to avoid costly construction and deconstruction of Java arrays to pass arguments to functions. Always knowing the number of arguments it sends[4] the caller simply picks which `run` method to call based on this number, letting the callee handle any array and/or linked list construction in the case of variable arity functions. The compiler defaults to emitting code that calls `runN` if there are more than K arguments at the callers call-site. [Kawa] (§6)

3.3 Literals

This section elaborates on techniques to compile in literal constants in LJSP code.

3.3.1 Constants

Whenever the compiler stumbles across an expression like `(+ a 1)` an appropriate representation of the `1` (which according to semantics evaluates to itself) needs to be emitted.

Now 1 is a small enough integer to be represented with `LispFixnum` which is used for all integers that will fit into a Java `long`, that is a 64-bit signed two's complement integer [JVMSpec] (§2.4.1).

The simple, but probably not very efficient, solution is to simply emit code for creating a new `LispFixnum` object with a value of 1 at the very spot the literal is found. This can be done using the `LispFixnum(long)` constructor. An equivalent Java expression of how the compiler emits a literal 1 would be:

```
new LispFixnum(1)
```

or in the Jasmin representation of Java bytecode (actual compiler output with comments for clarity):

[4] With the notable exception of calling using the function `apply` which takes as it's arguments a function and a list, calling the function with the elements of the list as the actual arguments. This is neatly resolved by compiling `apply` to always call using the `runN` method.

```
new LispFixnum
dup
ldc2_w 1      ; load 1 in long representation (uses two stack positions)
invokenonvirtual LispFixnum.<init>(J)V ; Constructor. uses up
                                       ; top three stack positions
;; a reference to the object is now on top of the stack
```

Similar code would be generated for floating point numbers, however instead creating an object of type `LispFlonum`, using the `LispFlonum(double)` constructor. The same is true of character and string constants using constructors `LispChar(char)` and `LispString(java.lang.String)` respectively. Even the arrays (`LispArray`) receive roughly the same treatment.

The process is similar for bignums, integers of arbitrary size, but due to their nature of possibly not fitting in the native integer types of Java instead the number is emitted as a string (in decimal) and then passed to the **`static`** method

```
public static LispBignum LispBignum.parse(java.lang.String)
```

which then parses the string into a `LispBignum` interpretation[5]. Example compiler output (with extra comment):

```
ldc_w "123131231231231231231231231231231313123"
invokestatic LispBignum.parse(Ljava.lang.String;)LLispBignum;
;; a LispBignum reference is now on top of the stack
```

3.3.2 Complex constants

A Lisp typically has a `quote` special form that suppresses evaluation of the enclosed expression and instead returns the data structure as is allowing for complex constants of lists possibly containing sublists and more.

Code for constructing the same structure could be recursively generated and inserted into the exact place where the `quote`-expression occurred, similar to how numbers were handled in the previous section. Thus making:

```
(lambda () (quote (1 a)))
```

equivalent to the code

```
(lambda () (cons 1 (cons (intern "a") nil)))
```

The calls to cons recreate the nodes of the list-structure and the call to intern recreates the symbol.

[5]At the time of writing it simply uses the `BigInteger(java.lang.String)` constructor of the Java standard library's `java.math.BigInteger` (the internal representation currently used for `LispBignum`:s).

This is however not quite optimal, since constantly recreating constant data in this fashion upon every call to the function would make many cases with complex constants be significantly slower than their interpreted counterparts, due to excessive allocation.

This also deviates from the interpreted semantics where

```
(let ((f (lambda () (quote (1 a)))))
  (eq? (f) (f))) ⇒ t
```

holds, since the same object, the very same one that constitues part of the function body, is always returned by the function.

Furthermore Scheme, with which LJSP happens to share a good deal of its semantics, requires quoted constants to always evaluate to the same object ([Incremental] (§3.11) cf. [R^{5}RS] (§7.2)).

A method for initializing constants in a function at load-time is neccessary. In Java **`static final`** fields may be initialized at class load time using a **`static`** initializer [JVMSpec] (§2.11) [JLS3] (§8.7).

By declaring a **`static final`** field, in the class for the function object, for each quoted literal in the body of the function being compiled and emitting code in the static initializer for constructing the literal. Now that the quoted literal has been constructed at load-time, code to fetch the **`static`** field can be emitted at the place where the literal is used.

The previous example compiles to something like:

```
public class f extends Procedure {
   private static final lit1;

   static { // Initializer
      f.lit1 = new Cons(new LispFixnum(1),
                        new Cons(Symbol.intern("a"), null));
   }

   public LispObject run(LispObject[] o) {
       return f.lit1;
   }

   // some constructor stuff omitted
   ...
}
```

Thus the code for recreating the quoted constant is run once at class load-time, and `(eq? (f) (f))` ⇒ `t` holds.

Of course the "simple" constants of the previous subsection would likely benefit (both performance-wise as well as being semantically closer to the interpreter) from a similar treatment as the constants written `quote` with the quote form in this section, and a planned feature is to emit all constants to **`private static final`** fields

of the generated class with extra logic to avoid duplicate constants, and duplicate fields, as long as the data structures involved are immutable (which holds for all numerical types used in LJSP as well as for characters and symbols).

3.4 Tail-call optimization implementation strategies

This section will describe a number of approaches to implement tail-call optimization on the JVM, why they seem plausible and why they work/don't work.

3.4.1 Handling self-tail-calls

Probably the most important and most common case of tail-calls are tail-calls from a function to itself, otherwise known as tail recursion. Implementing this special case of tail-call elimination is likely the simplest, of the practically implementable alternatives presented in this thesis, since no circumvention of the fact that the JVM can only perform jumps within a method [JVMSpec] (§4.8.1) needs to be performed; for this case jumps need only be performed within the method.

The method to implement this is almost exactly the same as the conventional, and completely general on a machine permitting jumps between functions, `goto`-based approach

A label is inserted at the of the generated `run` method. Whenever the compiler find a self-tail-call the compiler generates code to first set the argument variables, instead of pushing them to stack, and then jumping the the label. Special care, in the form of use of temporaries, needs to be taken when setting the variables so the values don't change until after the arguments in the call have been evaluated.

Example:

```
(nlambda fact (n acc)
  (if (= 0 n)
      acc
      (fact (- n 1) (* n acc))))
```

Roughly compiles to (Java pseudocode with goto, and stack):

```
...
public LispObject run(LispObject[] o) {
    LispObject fact = this; // (nlambda fact ...)
    // prologue to take apart the argument array
    // and store them in local variables
    LispNumber n = (LispNumber)o[0];
    LispNumber acc = (LispNumber)o[1];

  Lselftail:
    if (null != (o[0].equals(new LispFixnum(0)) ? t : null))
        return acc;
    else {
        stack.push(n.sub(new LispFixnum(1))); // we cannot assign n directly
        stack.push(n.mul(acc));               // since n is used on this row
        acc = stack.pop()
        n = stack.pop()

        goto Lselftail;
    }
}
...
```

If compiling without eliminating the tail-call the call-site would instead look something like :

```
...
if (null != (o[0].equals(new LispFixnum(0)) ? t : null))
    return acc;
else
    return this.run(new LispObject[]{n.sub(new LispFixnum(1)), n.mul(acc)});
...
```

Even if implementing another more general approach to TCO on the JVM implementing this approach to the special case of self-tail-calls is still very useful as a further optimization. It is by far the most common case of tail-recursion and this approach is much faster than most alternatives of implementing general TCO [Kawa].

This is the only kind of TCO implemented in the LJSP compiler as of writing.

3.4.2 Method-local subroutine approach

The only way of performing method calls on the JVM is by using the `invoke*` series of instructions, and returns performed using the `*return` of instructions [JVMSpec]. The method invocation instructions take their arguments (including the object on which the method is invoked for instance methods, which in a way can be considered the first argument) on the stack and automatically store the arguments in the

method local variables before transferring control to the first instruction in the method body. The call convention of the JVM is thus, in a sense, fixed and there is no way to directly manipulate stack frames. It is not possible to perform a goto to another method nor is it possible to assign to another methods local variables since they are associated with the current frame, which is created every time a method is invoked [JVMSpec] (§3.6).

To escape this call convention imposed by the JVM functions could be implemented as subroutines all within one method and defining a new function call convention, using the operand stack of the current frame, for these subroutines. The JVM comes with three instructions, `jsr <label>`, `jsr_w <label>` and `ret <local variable>`, that in conjunction can be used to implement subroutines. Since this calling convention is done on the JVM operand stack direct stack manipulation would be possible, and for all tail-calls `goto`s could be issued, like in the example of section 2.3.3 (p. 17).

This is however not possible on a modern and standards-compliant JVM implementation since the subroutine instructions can not be used in a truly recursive manner, since the verifier forbids it [JVMSpec] (§4.8.2). In the specification for the new java standard, Java SE 7, `jsr`, `jsr_w` and `ret` have been deprecated altogether (not without backwards compatibility for code compiled conforming to an older version) [JVMSpec SE 7] (§4.10.2.5). However this may be a useful, if non-portable technique, given that there are a handful of JVM implementations that seem to blatantly disregard this part of the specification[6].

3.4.3 Trampolines

One method of achieving general tail-recursion is trampolines [Baker]. By setting up an iterative procedure like (pseudo-code):

```
function trampoline(fn, args):-
    obj = make-tramp-thunk(lambda: apply(fn, args)) ; create starter thunk
    while tramp-thunk?(obj):        ; continue 'til we get something
                                    ; that isn't fit for bouncing
        obj = apply-tramp-thunk(obj())
    return obj
```

By now transforming functions to return a *thunk*, essentially just a function of no arguments, containing the expression in tail position of the function one can implement tail-recursion on a machine lacking direct stack manipulation by way of constantly bouncing up and down.

Any non-tail-calls are made to call the trampoline function, such that functions called in non-tail-position also can achieve proper tail-recursion for themselves.

[6]Or perhaps, the author suspects in particular due to the examples of recursive `jsr` usage floating across the net, conforms to an older edition of the JVM specification (of which the author has been unable to procure a copy of)

An example tail-recursive implementation of factorial adapted to be run by a trampoline, like the one above:

```
(defun fact (n acc)
  (if (zero? n)
      acc
      (make-tramp-thunk (lambda () (fact (1- n) (* n acc))))))
```

Thus instead of performing the tail-call a thunk containing the next action to take is returned to the trampoline, the current stack frame is thus discarded, which then continues by performing the tail-call `fact` would have normally performed on its own.

The trampoline loop could be implemented in Java and the transformation could be made in the *semantic analysis* pass of the compiler.

3.5 Scoping

This section discusses how to handle the different scoping methods in compiled code.

Note that these scoping methods are not exclusive of each other. Even if an implementation has true lexical scoping with closures the implementation method for statically scoped variables serves as a useful optimization for variables that the compiler can prove as not having been captured.

3.5.1 Static Scope

Static scope, as described in section 2.3.1 (p. 8), is very straightforward to implement on the JVM since each frame can have up to 65535 local variables [JVMSpec] (§4.10). These local variables can be viewed as registers, but associated with the current stack frame, from an assembly language point of view. By simply mapping received values to these variables static scoping is achieved as static scoping is natively supported by the JVM.

nlambda

See also 2.3.2 on page 14. The `nlambda` construct is implemented by binding the local variable 0, corresponding to Javas **`this`** for all instance methods [JVMSpec] (§7.6), to the variable specified as the name of the function.

Example:

```
(nlambda foo (a b) ...)
```

⇒

```
.method public run([LLispObject;)LLispObject;
    .limit stack 255   ; java requires these be set, set
```

```
    .limit locals 255  ; them to some generic big-enough size

    ;; function prologue deconstructing arguments array
    aload_1 ; the first method argument is gotten in local variable 1
    ldc_w 0
    aaload
    astore 5
    aload_1
    ldc_w 1
    aaload
    astore 6
    ;; end prologue

    ... do stuff (aload) with local variables 0 = foo, 5 = a and 6 = b ...

    areturn
.end method
```

3.5.2 Lexical Scope and Closures

Simple copying

Let's first consider lexical scoping, and specifically lexical closures[7], where the closed over variable bindings are never mutated, that is `set` is never used on them.

Example code (`nlambda` names provided for clarity, not for any self-recursion):

```
  ;; Bind function to global variable foo
  (defvar foo
    (nlambda foo (a)
      (nlambda bar (x) (+ x a))))
...
  ;; usage could look like:
  ;; (hoge and pyon are already declared variables, perhaps declared special)
  >> (setq hoge (foo 12))
  <closure 1 bar>
  >> (setq pyon (foo 11))
  <closure 2 bar>
  >> (hoge 2)
  14
  >> (pyon 33)
  44
  >> (hoge 1)
  13
```

[7] Which is what sets true *lexical scoping* apart from the statically scoped *local* variables in the previous section.

In this case the inner lambda, `bar`, has a free variable in `a`. However it doesn't mutate the binding of `a` so we may simply copy it into the `Procedure` subclass, at function construction. Thus closures can take their free variables as constructor arguments and save these to fields (which can be made **`final`** for extra guarantees of not mutating the binding). These can be treated in the same way as statically scoped/local variables in the previous section, but instead the variable `a` in `bar`s body would be mapped to a **`final`** instance variable in the closure object.

It could be compiled as such:

```
class foo extends Procedure {
    public foo() {}
    public LispObject run(LispObject[] o) {
        return new bar(o[0]);  // return closure
    }
}

class bar extends Procedure {
    private final LispObject free1;
    public bar(LispObject free1) {
        this.free1 = free1;
    }
    public LispObject run(LispObject[] o) {
        // (+ x a) argument x, closed-over variable a
        return ((LispNumber)o[0]).add((LispNumber)free1);
    }
}
```

Mutating "functions" such as `rplaca` (replace car/head of list), `rplacd` (replace cdr/tail of list) and `aset` (set element in array) don't count as mutating the variable binding. They don't change the variable bindings like `set` does, instead they mutate the data structure that the captured variable binding is referencing. Thus even though data is being mutated just copying all free variable references like before will have correct semantics.

In fact the situation is very similar to the only conditions under which Java has lexical closures; inner classes in a non-static context can refer to local variables and instance varibles declared in the enclosing class/scope given that they have been declared **`final`** (Go to [JLS3] (§8.1.3) and check if I was right.)

Now a fully-fledged LJSP compiler isn't quite complete without `set`. Does this spell the end for this approach to lexical closures?

No. The compiler could check for usage of `set`, both in closures and the function in which the variable was defined, of captured/free variables in the *semantic analysis* stage and use this method to implement lexical closures in the absence of `set`. At the same time the semantic analysis will assess all function bodies for free variables and annotates them for the code generator.

Even better: In the presence of `set` the compiler could exploit that `rplaca`, `rplacd` and `aset` can mutate state without having to touch the binding of the free variable (which also is impossible since the instance variable was declared `final`). In the case were the `set` is used the reference free variable is rewritten using one level of indirection, with the help of a mutable data type, in this case the `array`.

An example of this nifty rewrite (adapted from [Incremental] to fit LJSP):

```
(let ((f (lambda (c)
           (cons (lambda (v) (set 'c v))
                 (lambda () c)))))
  (let ((p (f 0)))
    ((car p) 12)
    ((cdr p))))
⇒
(let ((f (lambda (t0)
           (let ((c (make-array (list t0))))
             (cons (lambda (v)
                     (aset c 0 v)
                     v)
                   (lambda () (aref c 0)))))))
  (let ((p (f 0)))
    ((car p) 12)
    ((cdr p))))
```

This rewrite can be done in the semantic analysis stage. Thus the code generator only has to handle closures over immutable bindings [Incremental] (§3.9, §3.12) . Naturally this could be done using conses or other mutable datastructures allowing for this sort of indirect referencing.

3.5.3 Dynamic Scope

In Common Lisp a variable can be declared `special` (locally as well as globally, however for the purposes of this paper only the global case will be considered) having that variable be dynamically bound, allowing to mix the differently scoped sorts of variables in a way fitting the problem at hand[8]. Using `defvar` and `defparameter` to define global variables also has the effect of making the variable `special` [CLtL2] (§9.2, §5.2).

There are two main approaches, that are basically the same for both interpreted and compiled code. Aside from that book keeping is neccessary to keep track of

[8]Useful examples include global variables that can be temporarily overridden by rebinding. For instance rebinding the global variable `*standard-output*` in Common Lisp has the effect of redirection the standard output stream, since output functions define to output to the stream object pointed to by `*standard-output*`. In fact LJSP also has a global value `*standard-output*` used in the same way.

what symbols have ben declared as a `special`, this can simply be implemented as a property of the `Symbol` object.

Value slot

Each symbol object can be made to have one field `value` that points to the current top-level binding of the variable. Whenever the variable is rebound the old variable is saved in either a local variable, thus implicitly utilizing the native java stack, or pushed down an explicit stack for retrieval upon exit of the new binding and the restoring of the old one [MACLISP] (§3.2, §6.1).

This approach has the benefit of access speed to the detriment of rebinding speed. Due to the global shared state it imposes it is also fundamentally threading-incompatible.

This is the model currently implemented by the LJSP interpreter.

The latter approach to value slot based dynamical bindings, with a separate push-down stack, has the benefit of being able to eliminate tail-calls even in an environment with only dynamic variables (The LJSP interpreter uses this to great effect) [DynaTail].

Environments

Another approach would be to supply each function invocation with an easily extendable environment object of some sort. This dynamic environment object would then be used to lookup dynamically bound variables at runtime.

This would require a slight rewrite of the, for this particular example non-optimized, `Procedure` class proposed in section 3.2 (p. 22):

```
abstract class Procedure extends LispObject {
    ...
    public abstract LispObject run(LispObject[] o,
                                   Environment dynamicEnvironment);
}
```

This environment is passed on at every function call site so if `foo` calls `bar` `bar` will inherit the dynamic environment of `foo`, possibly extending it. In the case of a mixed lexical/dynamic scoping environment like Common Lisp if the name of one of the arguments of `bar` coincides with the name of a variable declared `special` the environment will be augmented shadowing the old declaration of that variable until `bar` returns.

This method of handling dynamically scoped variables mimics almost exactly how environments are passed around in many Lisp interpreters, including the very first one [McCarthy60].

This method has the benefit that, for suitably built environment data structures, threads in a multi-threaded application would be able to share the same base-level binding of a dynamic variable yet capable of shadowing this binding with their

own to have a thread-local top-level dynamic variable binding. Different threads will reference the same base environment, but will have their own environment extensions on top of this. This can perhaps be thought of as having a multi-headed stack of some sort, with one top per thread.

The drawbacks include slower lookup of dynamic variables as well as extra overhead due to always passing on the dynamic environment, even in cases were it might not be needed (a sufficiently smart compiler might be able to alleviate this somewhat however).

Chapter 4

Results

4.1 Benchmarks

The y axles represents execution time in milliseconds. The x axles represents the size of the argument passed to the functions.

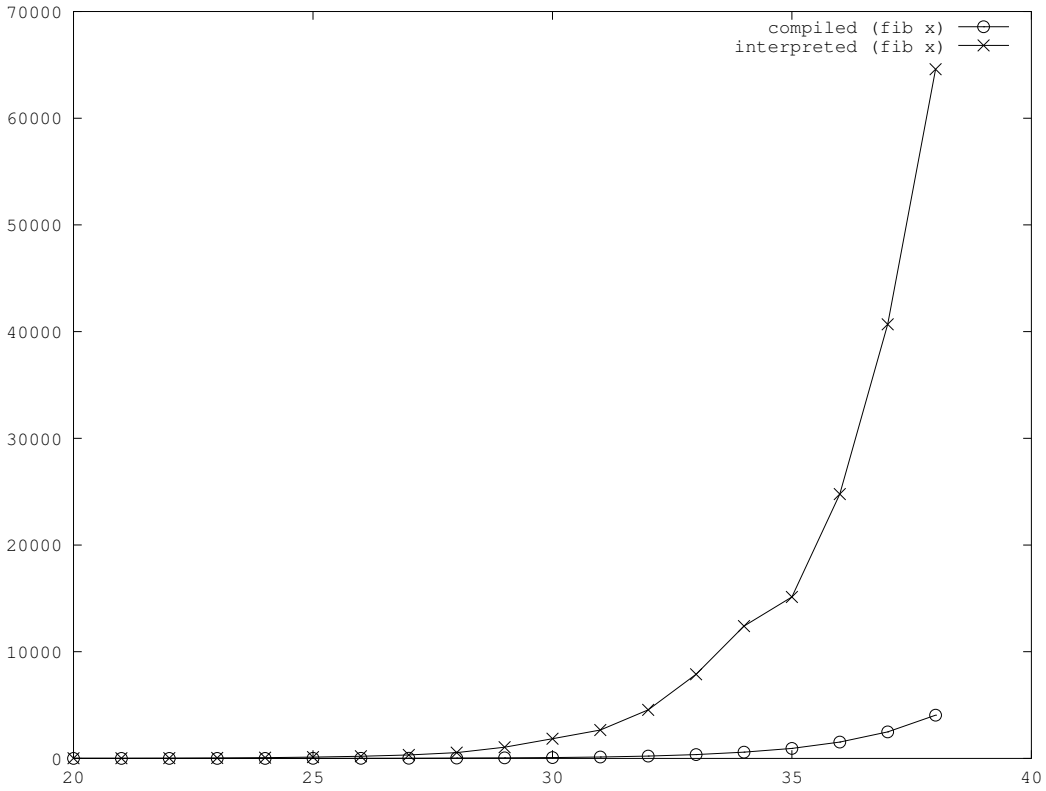

Figure 4.1. comparison of fib speeds

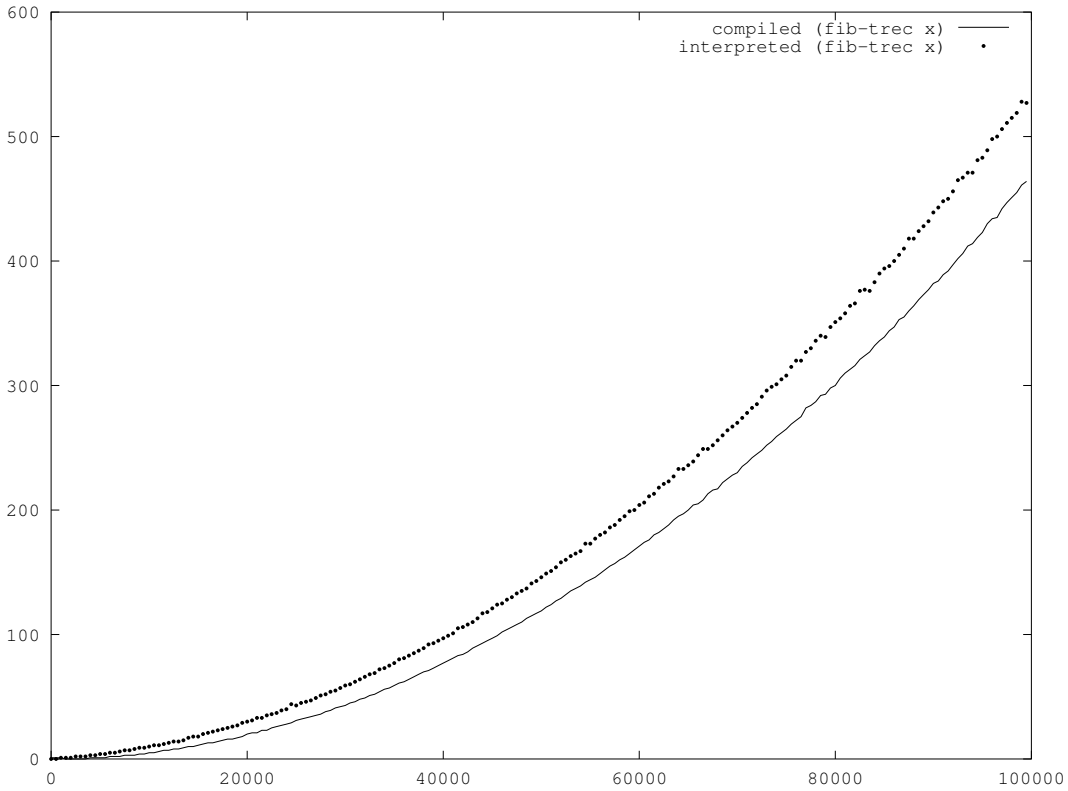

Figure 4.2. comparison of fib-trec speeds

The differences betweeen the compiled `fib-trec` and interpreted ditto is smaller than the difference between `fib` compiled and non-compiled. Likely since fib-trec is tail-recursive (time complexitiy of $O(n)$) and the result gets big very fast before it starts getting slow. Likely most of the execution time of the `fib-trec` taken up by bignum arithmetics. The difference in speed here was expected to be bigger but is logical considering the bignum overhead.

However in the case of the naive `fib` the execution time is almost improved tenfold. This magnitude of improvement was somewhat unexpected.

4.2 Conclusions

The JVM, despite it's quirks and heavy optimization towards Java, is a suitable platform for a Lisp environment, having both run-time loading of code and garbage collection for free. With the notable inconvenience of the inability to support proper tail-calls without using major kludges to the detriment of speed and Java interoperability.

A lot of time was spent on finding out just how hard it is to have proper tail-

recursion, as per [R^{5}RS], in Java. In particular how `jsr` was found to be inadequate for implementing subroutines with direct stack control, thus enabling TCO. Finally the conclusion that only self-tail-calls were practical enough to be implementable during the time frame of the project was reached, thus falling slightly short of that goal.

It was found that significant speed gains can be had even with a simple un-optimizing compiler like the one presented. This was expected but the extent of the speedup of the naive fibonacci test case was somewhat surprising, given how no arithmetics are open-coded and are performed as method calls on objects, like `LispFixnum`, allocated on the heap.

I have found that working on the compiler of a dynamic language from the inside of an existing interpreter environment, and extending it, is a very efficient way to develop a compiler quickly.

I have found that writing a compiler takes a lot of effort and a fairly long time, even for a very simple one. The project was initially much more ambitious and intended to have a fully bootstrappable compiler of an extended variant of the full language the LJSP interpreter supports, yet many things had to be scrapped due to time constraints and only a subset was implemented. Things like implementing support for lexical scope with lexical closures or even something as simple as Lisp macro support in the compiler, However by writing this report a lot of the steps to take the compiler further have been outlined and thoroughly investigated.

4.3 The future?

- Have compiled functions handle receiving, by causing an error condition, too many arguments instead of silently ignoring it.
- Implement the optimization for function calls in section 3.2 (p. 22) at the same time as the above (this makes sense as that model makes checking for function arity much more effective than otherwise.)
- Implement compiler support for variable arity procedures.
- Implement a semantical analysis stage of compilation.
- Have the compiler support macros with a macro-expansion pass prior to semantic analysis and code generation.
- Implement lexically and dynamically bound variables, preferably while retaining the current model of statically scoped variables, as an optimization, when semantic analysis has found a variable neither captured by a closure nor declared as dynamically bound.
- Implement `set` and have it work for lexical scoping (to keep it fun; closures would be too trivial otherwise) and dynamic scoping alike.

- Replace or fix the old reader currently used by the LJSP interpreter.
- Have the compiler bootstrap.
- Find out how much of the reflection-based model of Java interoperability, used by the interpreter, can be salvaged and made into a newer better defined and more easily compiled approach to Java interoperability.

Chapter 5

References

[**AIM353**]
Guy Lewis Steele Jr. and Gerald Jay Sussman

Lambda: The Ultimate Imperative

AI Memo 353, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, March 1976

[**AIM443**]
Guy Lewis Steele Jr.

Debunking The "Expensive Procedure Call" Myth, or,
Procedure Call Implementations Considered Harmful, or,
Lambda: The Ultimate GOTO

In *Proceedings of the ACM National Conference*, pp. 153-162, Seattle, October 1977. Association for Computing Machinery. Revised version published as AI Memo 443, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, October 1977.

[**Baker**]
Henry G. Baker

CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A.

DRAFT for `comp.lang.scheme.c` Feb. 4, 1994

In *ACM Sigplan Notices* 30, 9 (Sept. 1995), 17-20.

[**CLtL2**]
Guy L. Steele Jr, Thinking Machines, Inc.

Common Lisp the Language, 2nd edition

Digital Press, 1990
ISBN 1-55558-041-6

[DynaTail]
Darius Bacon

Tail Recursion with Dynamic Scope

Available from (fetched April 13, 2012):
`http://wry.me/~darius/writings/dynatail.html`

Originally published on `comp.lang.scheme`, date unknown. Otherwise unpublished.

[gcc]
Using and Porting the GNU Compiler Collection – GCC version 3.0.2

§17 Passes and Files of the Compiler

Available from, among others (fetched April 13, 2012):
`http://sunsite.ualberta.ca/Documentation/Gnu/gcc-3.0.2/html_mono/gcc.html#SEC170`

[Graham]
Paul Graham

On Lisp

Prentice Hall, 1993
ISBN 0130305529

[Incremental]
Abdulaziz Ghuloum

An Incremental Approach to Compiler Construction

Proceedings of the 2006 Scheme and Functional Programming Workshop University of Chicago Technical Report TR-2006-06

Department of Computer Science, Indiana University, Bloomington, IN 47408

[JLS3]
James Gosling, Bill Joy, Guy Steele, Gilad Bracha

The Java™ Language Specification – Third Edition

Addison Wesley (June 24, 2005)
ISBN 0-321-24678-0

[JVMSpec]
Tim Lindholm, Frank Yellin.

The Java™ Virtual Machine Specification – Second edition

Prentice Hall (April 24, 1999)
ISBN 0-201-43294-3

[**JVMSpec SE 7**]
Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley

The Java™ Virtual Machine Specification – Java SE 7 Edition

Oracle, JSR-000924 (July 2011)

Available online: `http://docs.oracle.com/javase/specs/jvms/se7/html/index.html`

[**Kawa**]
Per Bothner, Cygnys Solutions

Kawa: Compiling Scheme to Java

Revision of ***Kawa – Compiling Dynamic Languages to the Java VM***, which was presented at the *USENIX Annual Technical Conference*. New Orleans, Louisiana. June 15-19, 1998

Available online (as of May 20, 2011): `http://per.bothner.com/papers/`

[**MACLISP**]
David A. Moon

MACLISP Reference Manual

Project MAC, MIT
Cambridge, Massachusetts

Revision ∅
April 1974

[**McCarthy60**]
John McCarthy

Recursive Functions of Symbolic Expressions and Their Computation by Machine, part 1

Massachussetts Institute of Technology, Cambridge, Mass.
In *Communications of the ACM*, April, 1960

[**R^{5}RS**]
R. Kelsey W. Clinger, J. Rees (eds.)

Revised^5 Report on the Algorithmic Language Scheme

In *Higher-Order and Symbolic Computation*, Vol. 11, No. 1, August, 1998 and *ACM SIGPLAN Notices*, Vol. 33, No. 9, September, 1998

Part II

Appendices

Chapter 6

Appendix A

Contains the compiler code in all it's messy (it is still littered with old code in comments, how horrible!) glory.

Also available, together with the neccessary runtime environment, at:

`http://www.nada.kth.se/~antonki/programmering/ljsp-kandx.tar.bz2`

```
1  ;-*- Mode: Lisp -*-
2
3  ;;; IDEA: (doesn't really belong here?) Start having fexprs (or
4  ;;; similar) so you can be meaner in how you handle macros (as
5  ;;; statically as CL for instance).
6
7  ;;; Can you somehow coerce the JVM into thinking duck-typing is a good idea?
8
9  ;; TODO: DONE-ish add argument to pretty much everything to keep track of
       tail-call or not
10 ;;       * Judicious finals everywhere (we don't subclass the generated classes
       after all)
11 ;;       * Perhaps move classname out of the environment plist?
12 ;;       * More correct-amount-of-args-checking and the likes
13 ;;       * Make all environtment be ONE environment and convey static/lexical/
       dynamic using the plist instead?!?!?
14 ;;       * instead of having the creepy %literal-vars% and %literal-init% type
       variables scan code ahead of
15 ;;         time to generate a table of constants? (we don't win much on this move
       except
16 ;;         having cleaner code with less side-effects
17
18
19 (require 'java)
20
21 ;; Perhaps move this to stuff.ljsp due to it's bootstrappinessishness?
22 (unless (running-compiled?)
23   (defmacro defvar (a)
24     (unless (symbol-value (cadr a))      ; unless already bound
25       (list 'setq (cadr a) (caddr a)))))
26
27
28 ;; FOR NOW
29 (defvar cfib '(nlambda fib (n) (if (= n 0) 0 (if (= n 1) 1 (+ (fib (- n 1)) (fib
       (- n 2)))))))
30
31 (defvar cfib-trec '(lambda (n)
32                      ((nlambda calc-fib (n a b)
33                         (if (= n 0)
34                             a
35                             (calc-fib (- n 1) b (+ a b))))
36                       n 0 1)))
```

```
37
38  (defvar fcollatz '(nlambda collatz (n) (print n) (if (= n 1) nil (collatz (if (= (
        mod n 2) 0) (/ n 2) (+ 1 (* n 3)))))))
39
40  ;; differs semantically slightly from the mapcar1 in stuff.ljsp (aside from wierd
        binding-stuffs, it doesn't use end? for end of list)
41  (defvar mopcor1 '(nlambda mapcar1 (fnx lstx) (if lstx (cons (fnx (car lstx)) (
        mapcar1 fnx (cdr lstx))) nil)))
42
43  ;; differs semantically slightly from the assq in stuff.ljsp (aside from wierd
        binding-stuffs, it doesn't use end? for end of list)
44  ;; left some crud in ((lambda nil nil)) (from macro expansion), for testing, in it
         but removed others that wouldn't work in static scoping...
45  (defvar cassq '(nlambda assq (key alist) (if (eq? alist nil) ((lambda nil nil)) (
        if (eq? key (car (car alist))) (car alist) (assq key (cdr alist))))))
46
47  (defvar quote-test (subst-symbols
48                      '(lambda (a)
49                        (cons a '(#\W (123131231231231231231231231231231231231123 .
                                 5343412914294967296) (<a> <b> <c>) b #(hej din fisk (1 2
                                  3)) "potatismossa" . 12.4)))
50                      ;; since the current reader has no syntax for introducing NaN'
                            s we do this. the compiler needs to handle it
51                      ;; after all and maybe the reader supports some syntax for NaN
                              in the future
52                      '<a> (/ 0.0 0.0)
53                      '<b> (/ 1.0 0.0)     ; same for pos inf
54                      '<c> (/ 1.0 -0.0)))  ; same for neg inf
55
56  (defvar cfact '(nlambda fact (n acc) (if (= 0 n) acc (fact (- n 1) (* n acc)))))
57
58  ;; Blargh my parser is broken in many strange ways and crazy so let's
59  ;; have a crazy variable for this
60  (defvar dblfnutt (prin1-to-string '|"|))
61
62  (defvar nl "
63  ")
64
65  (defvar *compiled-body* '())
66
67  (defvar *dynamic-variables* '())
68
69  (defvar *label-counter* 0)
70  (defvar *funclabel-counter* 0)
71  (defvar *static-var-counter* 0)
72
73  ;; These are dynmic variables locally overrided to contain
74  ;; initializing code, and the static variable definitions for all the
75  ;; literals, into their static variables, for the currently compiling
76  ;; class file. Defvarring them like this makes them be SPECIAL (or whatever)
77  (defvar %literal-init% nil)
78  (defvar %literal-vars% nil)
79
80  ;; local variables 5 and above are for static environment. 0 to 5 have
81  ;; special uses. With 0 always referring to the this variable. 3 being
82  ;; a temp variable and the others are for the time being undefined.
83  (defvar +reserved-regs-split+ 5)
84
85  (defun get-label ()
86    (concat "L" (inc *label-counter*)))
87
88  (defun get-funclabel ()
89    (concat "FUN" (inc *funclabel-counter*)))
90
91  (defun get-static-var-name ()
92    (concat "lit" (inc *static-var-counter*)))
93
94
95
96  ;;;; Functions implemented using java classes that perhaps should be
97  ;;;; made built-in to ease boot-strapping and portability
98
99  ;; For portabilitys sake consider makeing this a built in subr
100 (defun concat strs
```

```
101     (let ((sb (send StringBuilder 'newInstance)))
102       (dolist (str strs)
103         (send sb 'append str))
104       (send sb 'toString)))
105
106   ;; Same: for portabilitys sake consider making this built in or similar
107   (defun load-proc (name)
108     (let ((name (if (type? 'symbol name) (prin1-to-string name) name)))
109       (send (send Class 'forName name) 'newInstance)))
110
111   (defun concat-nl strs
112     (apply concat (flatten (mapcar (lambda (x) (list x nl)) strs))))
113
114   (defun NaN? (a)
115     (send Double 'isNaN a))
116
117   (defun infinite? (a)
118     (send Double 'isInfinite a))
119
120   ;;;; End functions using java
121
122
123   ;;;; CODE WALKER FOR LEXICAL ANALYSIS
124   ;;;; Used to find free variables in lambdas (and macros) mainly
125   ;; This here thing does NOT want code with macros in it (HINT:
126   ;; remember to expand macros way early) (just think about the
127   ;; confusion let would be, for instance). Also think about: local
128   ;; macros WTF?
129
130   (defun analyze (a . rst)
131     (let ((local-variables (car rst)))
132       (uniq (sort-list (analyze-expr a local-variables) hash<) eq?)))
133
134
135   (defun analyze-expr (a local-variables)
136     (if (atom? a)
137         (if (and (type? 'symbol a)
138                  (not (member a local-variables))
139                  (not (member a *dynamic-variables*)))
140             (list a)
141             '())
142         (case (car a)
143           (quote '())           ; no variables can be captured in a quote
144           (lambda (analyze-lambda a local-variables)) ; macro?
145           (if (analyze-list a local-variables))       ; Treat if specially in future
                 ? (is there a point in closing over the VARIABLE if ?)
146           (otherwise (analyze-list a local-variables)))))
147
148   (defun analyze-lambda (a local-variables)
149     (unless (eq? (car a) 'lambda)         ; macro?
150       (error "You ought to supply me with a lambda when you want to analyze free
              variables in a lambda."))
151     (letrec ((scan (lambda (lst acc)
152                      (cond ((null? lst) (reverse! acc))
153                            ((atom? lst) (reverse! (cons lst acc)))
154                            (t (scan (cdr lst) (cons (car lst) acc)))))))
155       (analyze-list (cddr a) (append (scan (cadr a) nil) local-variables))))
156
157   (defun analyze-list (a local-variables)
158     (letrec ((roop (lambda (lst acc)
159                      (if (end? lst)
160                          acc
161                          (roop (cdr lst) (append acc (analyze-expr (car lst)
                                 local-variables)))))))
162       (roop a nil)))
163
164   ;; Remember to check if there are too many arguments as well in things like if and
        print
165
166   (defun emit-if (a e tail)
167     (let ((condition (cadr a))
168           (true-expr (caddr a))
169           (false-expr (cadddr a))
170           (label (get-label))
```

```
171         (label-after (get-label)))
172     (concat ";; " a nl
173             (emit-expr condition e nil)
174             "ifnonnull " label " ; branches to the true-expr" nl
175             (emit-expr false-expr e tail)
176             "goto " label-after " ; Don't also run the true-expr like a fool" nl
177             label ":" nl
178             (emit-expr true-expr e tail)
179             label-after ":" nl
180             ";; endif" nl)))
181
182 ;;;; Used by emit-funcall to generate code for how to structure arguments before
        the actual call
183 ;;;; This particular version is when passing arguments in an array
184 (defun emit-funargs (args e)
185   (letrec ((roop (lambda (lst e cntr asm)
186                    (if (end? lst)
187                        asm
188                        (roop (cdr lst)
189                              e
190                              (1+ cntr)
191                              (concat asm
192                                      "dup"                          nl
193                                      "ldc_w " cntr                  nl
194                                      (emit-expr (car lst) e nil)
195                                      "aastore"                      nl))))))
196     (let ((len (length args)))
197       (if (zero? len)
198           (concat "aconst_null" nl) ; very slight optimization of the no-argument
                case
199           (concat "ldc_w " len           nl
200                   "anewarray LispObject" nl
201                   (roop args e 0 ""))))))
202
203
204 ;; Version for passing arguments on stack in regular order
205 #;(defun emit-funargs (args e)
206   (if args
207       (apply concat (mapcar (lambda (x) (emit-expr x e nil)) args)))
208       "")
209
210 ;; This will need to do different things for a non-compiled function a
211 ;; compiled function a compiled or non-compiled macro according to
212 ;; their current bindings (we fearlessly ignore that for the
213 ;; dynamically scoped case our function bindings might change and
214 ;; such. This is less a problem in the lexically scoped case yet still
215 ;; a problem for some cases (which cases?))
216 ;; WHEN JSR-ing (or similar):
217 ;; Don't forget to reverse the arglist
218 ;; Don't forget to push local vars....
219 ;; TODO: Think up ways to store variables together with some sort of type data so
        we know when to do what funcall
220
221 ;; POSSIBLE OPTIMIZATION: Inline in a nice way when just a regular
222 ;; non-recursive lambda-thingy (like the case the let- or progn macro
223 ;; would generate (especially the latter one is trivial))
224 (defun emit-funcall (a e tail)
225   (let ((fun (car a))
226         (args (cdr a)))
227     (if (and tail
228              (type? 'symbol fun)
229              (print (get-variable-property fun 'self e)))
230         (emit-self-recursive-tail-call args e)
231         (concat ";; " a                    nl
232                 (emit-expr fun e nil)          ; puts the function itself on the
                       stack
233                 "checkcast Procedure"      nl
234                 "; preparing args"         nl
235                 (emit-funargs args e)
236                 "; end preparing args"     nl
237                 "invokevirtual Procedure.run([LLispObject;)LLispObject;" nl))))
238
239 ;; WRITTEN FOR STATIC ONLY
240 ;; TODO: rewrite when stuff changes...
```

```
241  ;; This currently assumes a certain layout of variables laid out by
         emit-lambda-body.
242  ;; Note how we just reuse the old state locations since a tail-call let's us
         discard the old state for this frame entirely
243  ;; However: Before we start setting the local variables we have pushed all the
         results to the stack.
244  ;; If we didn't all sorts of side-effect mayhem might occur for example for
245  ;; (nlambda foo (a b) (if (> a 100) a (foo (+ a 2) (* a b)))) a is used twice in
         the argument list
246  (defun emit-self-recursive-tail-call (args e)
247    (letrec ((funargs-push (lambda (lst e asm)
248                             (if (end? lst)
249                                 asm
250                                 (funargs-push (cdr lst)
251                                               e
252                                               (concat asm
253                                                       (emit-expr (car lst) e nil)))
                                                           )))
254             (funargs-pop (lambda (cntr offset asm)
255                            (if (zero? cntr)
256                                asm
257                                (funargs-pop (1- cntr)
258                                             offset
259                                             (concat asm
260                                                     "astore " (+ (1- cntr) offset)
                                                               nl))))))
261      (concat ";; self-recursive tail-call args: " args nl
262              (funargs-push args e "")
263              (funargs-pop (length args) +reserved-regs-split+ "")
264              "goto Lselftail" nl)))
265
266  (defun emit-quote (a e)
267    (unless (and (eq? (car a) 'quote)
268                 (= (length a) 2))
269      (error (concat "Something's wrong with your quote: " a)))
270    (unless (and (type? 'string %literal-init%)  ; compile-lambda does initialize
         these to "",
271                 (type? 'string %literal-vars%)) ; so they should always be strings
                       when we end up here
272      (error (concat "Special variables %literal-vars%: " (prin1-to-string %
           literal-vars%)
273                     " and %literal-init%: " (prin1-to-string %literal-init%)
274                     " not properly initialized")))
275    (let ((static-var (get-static-var-name))
276          (classname  (getf e 'classname)))
277      (setq %literal-vars% (concat %literal-vars%
278                                   ".field private static final " static-var "
                                       LLispObject;" nl))
279      (setq %literal-init% (concat %literal-init%
280                                   (emit-obj (second a) e)
281                                   "putstatic " classname "/" static-var "
                                       LLispObject;"))
282      (concat "getstatic " classname "/" static-var " LLispObject;" nl)))
283
284  (defun emit-java-double (a)
285    (cond ((NaN? a)
286           ;; KLUDGE: workaround using division by zero (resulting in NaN) since
287           ;; jasmin seems to have trouble, or at least is lacking any documention,
288           ;; how to load a NaN double as a constant
289           (concat ";; jasmin lacks all sort of documentation on how to push a NaN
                 double. Division by zero works as a work-around." nl
290                   "dconst_0" nl
291                   "dconst_0" nl
292                   "ddiv"     nl))
293          ((and (infinite? a) (not (neg? a)))
294           ;; KLUDGE: same thing but for positive infinity
295           (concat ";; hackaround for positive infinity" nl
296                   "ldc2_w 1.0d" nl
297                   "dconst_0"    nl
298                   "ddiv"        nl))
299          ((and (infinite? a) (neg? a))
300           ;; KLUDGE: same thing but for negative infinity
301           (concat ";; hackaround for negative infinity" nl
302                   "ldc2_w -1.0d" nl
```

```
303                     "dconst_0"      nl
304                     "ddiv"          nl))
305             (t
306              ;; that d is important, otherwise we are loading a float (not double)
307              ;; constant and introducing rounding errors
308              (concat "ldc2_w " a "d" nl))))
309
310   (defun emit-java-long (a)
311     (concat "ldc2_w " a nl))
312
313   ;; Emits code to regenerate an object as it is (quoted stuffs use
314   ;; this)
315   ;; TODO: * what about procedures and the like, while not having a
316   ;;         literal representation one might send crazy shit to the
317   ;;         compiler...?
318   ;;       * What about uninterned symbols? (Does it really make a difference?) Very
          tricky shit this :/
319   (defun emit-obj (obj e)
320     (cond ((eq? obj nil) (emit-nil))
321           ((type? 'fixnum obj)
322            (concat "new LispFixnum" nl
323                    "dup"            nl
324                    (emit-java-long a)
325                    "invokenonvirtual LispFixnum.<init>(J)V" nl))
326           ((type? 'flonum obj)
327            (concat "new LispFlonum" nl
328                    "dup"            nl
329                    (emit-java-double obj)
330                    "invokenonvirtual LispFlonum.<init>(D)V" nl))
331           ((type? 'bignum obj)
332            (concat "ldc_w " dblfnutt obj dblfnutt nl
333                    "invokestatic LispBignum.parse(Ljava.lang.String;)LLispBignum;"
                        nl))
334           ((type? 'string obj)
335            (concat "new LispString"                  nl
336                    "dup"                             nl
337                    "ldc_w " dblfnutt obj dblfnutt    nl
338                    "invokenonvirtual LispString.<init>(Ljava.lang.String;)V" nl))
339           ((type? 'array obj)
340            (concat "new LispArray"                   nl
341                    "dup"                             nl
342                    (nlet roop ((cntr (length obj))
343                                (asm (concat "ldc_w " (length obj)  nl
344                                             "anewarray LispObject" nl)))
345                      (if (zero? cntr)
346                          asm
347                          (roop (1- cntr)
348                                (concat asm
349                                        "dup"                                  nl
350                                        "ldc_w " (1- cntr)                     nl
351                                        (emit-obj (aref obj (1- cntr)) e)
352                                        "aastore"                              nl))))
353                    "invokenonvirtual LispArray.<init>([LLispObject;)V" nl))
354           ((type? 'symbol obj)
355            (concat "ldc_w " dblfnutt obj dblfnutt nl
356                    "invokestatic Symbol.intern(Ljava.lang.String;)LSymbol;" nl))
357           ((type? 'char obj)
358            (concat "new LispChar"                  nl
359                    "dup"                           nl
360                    "bipush " (char->integer obj)   nl
361                    "invokenonvirtual LispChar.<init>(C)V" nl))
362           ((type? 'cons obj)
363            (concat "new Cons"              nl
364                    "dup"                   nl
365                    (emit-obj (car obj) e)
366                    (emit-obj (cdr obj) e)
367                    "invokenonvirtual Cons.<init>(LLispObject;LLispObject;)V" nl))
368           (t (error (concat "Couldn't match type for:" a)))))
369
370   (defun emit-return-self (obj e)
371     (cond ((type? 'symbol obj) (emit-variable-reference obj e))
372           ((atom? obj)         (emit-obj obj e))
373           (t (error "Arghmewhats?"))))
374
```

```
375
376
377 ;; TODO: when/if removing multiple alists for different sorts of environments:
        REWRITE
378 ;; THIS IS REALLY A HUGE KLUDGE
379 (defun get-variable-property (var property e)
380   (or (get-static-variable-property var property e)
381       (get-lexical-variable-property var property e)
382       (get-dynamic-variable-property var property e)))
383
384 (defun get-static-variable-property (var property e)
385   (getf (cddr (assq var (getf e 'static-environment))) property))
386
387 (defun get-lexical-variable-property (var property e)
388   (getf (cddr (assq var (getf e 'dynamic-environment))) property))
389
390 (defun get-dynamic-variable-property (var property e)
391   (getf (cddr (assq var (getf e 'lexical-environment))) property))
392
393
394 ;;;; Variable lists look like ((a <storage-location> . <extra-properties-plist>) (
        b ...) ...)
395 ;;;; e.g ((a 1) (fib 0 self t))
396 (defun get-static-variable (var e)
397   (let ((static-environment (getf e 'static-environment)))
398     (cadr (assq var static-environment))))
399
400 (defun get-lexical-variable (var e)
401   (let ((lexical-environment (getf e 'lexical-environment)))
402     (cadr (assq var lexical-environment))))
403
404 (defun get-dynamic-variable (var e)
405   (let ((dynamic-environment (getf e 'dynamic-environment)))
406     (cadr (assq var dynamic-environment))))
407
408 (defun emit-variable-reference (a e)
409   (let ((static-var-place  (get-static-variable a e))
410         (lexical-var-place (get-lexical-variable a e))
411         (dynamic-var-place (get-dynamic-variable a e)))
412     (cond (static-var-place (concat "aload " static-var-place nl))
413           (lexical-var-place (concat "nolexicalyet" nl))
414           (dynamic-var-place (concat "nodynamicyet" nl))
415           (t (error (concat "Variable: " a " doesn't seem to exist anywhere."))))))
416
417 (defun emit-arithmetic (a e)
418   (unless (= (length a) 3)
419     (error (concat "You can't arithmetic with wrong amount of args: " a)))
420   (concat (emit-expr (second a) e nil)
421           "checkcast LispNumber" nl
422           (emit-expr (third  a) e nil)
423           "checkcast LispNumber" nl
424           "invokevirtual LispNumber."
425           (case (car a) (+ "add") (- "sub") (* "mul") (/ "div"))
426           "(LLispNumber;)LLispNumber;" nl))
427
428 (defun emit-integer-binop (a e)
429   (unless (= (length a) 3)
430     (error (concat "You can't integer-binop with wrong amount of args: " a)))
431   (concat (emit-expr (second a) e nil)
432           "checkcast LispInteger" nl
433           (emit-expr (third  a) e nil)
434           "checkcast LispInteger" nl
435           "invokevirtual LispInteger."
436           (case (car a) (mod "mod") (ash "ash"))
437           "(LLispInteger;)LLispInteger;" nl))
438
439
440 ;; Used, internalish, to emit dereferencing the variable t (currently special
        hardcoded, put in own function for modularity
441 (defun emit-t (e)
442   (let ((classname (getf e 'classname)))
443     (concat "getstatic " classname "/t LLispObject;" nl))) ; TODO: in the future
          try to emit a variable reference to t here instead of this hardcoded
```

```
          mishmash
444
445  ;; Used to emit the sequence to convert a java boolean to a more lispish boolean.
         Used in mostly "internalish" ways.
446  (defun emit-boolean-to-lisp (e)
447    (let ((label (get-label))
448          (label-after (get-label)))
449      (concat "ifeq " label nl
450              ;; (emit-return-self 123 nil)     ; TODO: change me to emit t later
451              (emit-t e)
452              "goto " label-after nl
453              label ":"           nl
454              (emit-nil)
455              label-after ":"     nl)))
456
457  (defun emit-= (a e)
458    (unless (= (length a) 3)
459      (error (concat "You can't = with wrong amount of args: " a)))
460    (concat (emit-expr (second a) e nil)
461            ;; "checkcast LispNumber" nl
462            (emit-expr (third a)  e nil)
463            ;; "checkcast LispNumber" nl
464            "invokevirtual java/lang/Object.equals(Ljava/lang/Object;)Z" nl
465            (emit-boolean-to-lisp e)))
466
467  (defun emit-neg? (a e)
468    (unless (= (length a) 2)
469      (error (concat "You can't neg? with wrong amount of args: " a)))
470    (concat (emit-expr (second a) e nil)
471            "checkcast LispNumber"           nl
472            "invokevirtual LispNumber.negP()Z" nl
473            (emit-boolean-to-lisp e)))
474
475  (defun emit-eq? (a e)
476    (unless (= (length a) 3)
477      (error (concat "You can't eq? with wrong amount of args: " a)))
478    (let ((label-ne (get-label))
479          (label-after (get-label)))
480      (concat (emit-expr (second a) e nil)
481              (emit-expr (third a)  e nil)
482              "if_acmpne " label-ne    nl
483              (emit-t e)
484              "goto "      label-after nl
485              label-ne ":"             nl
486              "aconst_null"            nl
487              label-after ":"          nl)))
488
489  (defun emit-eql? (a e)
490    (error "eql? not implemented"))
491
492  ;; TODO: * two-argument version of print
493  ;;       * implement without temp variable if possible. Having
494  ;;         temp-variables might grow trickier when some method
495  ;;         implementations do away with the need to (always)
496  ;;         deconstruct an array
497  (defun emit-print (a e)
498    (let ((label-nil   (get-label))
499          (label-after (get-label)))
500      (concat ";; " a                                                          nl
501              "getstatic java/lang/System/out Ljava/io/PrintStream;"           nl
502              (emit-expr (cadr a) e nil)
503              "dup"                                                            nl
504              "astore_2 ; store in the temp variable"                          nl
505              "dup"                                                            nl
506              "ifnull " label-nil                                              nl
507              "invokevirtual java/lang/Object.toString()Ljava/lang/String;"    nl
508              "goto " label-after                                              nl
509              label-nil ":"                                                    nl
510              "pop"                                                            nl
511              "ldc_w " dblfnutt "nil" dblfnutt                                 nl
512              label-after ":"                                                  nl
513              "invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V" nl
514              "aload_2 ; we return what we got"                                nl)))
515
```

```
516 (defun emit-set (a e)
517   (error "set not implemented"))
518
519 (defun emit-nil ()
520   (concat "aconst_null" nl))
521
522 (defun emit-car-cdr (a e)
523   (unless (= (length a) 2)
524     (error "You can't " (car a) " with wrong amount of args: " a))
525   (let ((label-nil (get-label)))
526     (concat (emit-expr (cadr a) e nil)
527             "dup"                                    nl
528             "ifnull " label-nil                      nl
529             "checkcast Cons"                         nl
530             "getfield Cons/" (car a) " LLispObject;" nl
531             label-nil ":"                            nl)))
532
533 (defun emit-cons (a e)
534   (unless (= (length a) 3)
535     (error "You can't cons with wrong amount of args: " a))
536   (concat "new Cons"                                          nl
537           "dup"                                               nl
538           (emit-expr (second a) e nil)
539           (emit-expr (third a)  e nil)
540           "invokenonvirtual Cons.<init>(LLispObject;LLispObject;)V" nl))
541
542 (defun emit-expr (a e tail)
543   (if (list? a)
544       (case (car a)
545         ;; To be able to pass these, where appropriate (e.g: not if), as arguments
              the bootstrap code needs to define functions that use these builtins.
              e.g: (defun + (a b) (+ a b))
546         ;; (running-compiled? (emit-return-self 1337 nil)) ; TODO: change me to
              emit t later
547         (running-compiled? (emit-t e))
548         (set       (emit-set a e))
549         (eq?       (emit-eq? a e))
550         (eql?      (emit-eql? a e))
551         ((or + - * /) (emit-arithmetic a e))
552         (=         (emit-= a e))
553         (neg?      (emit-neg? a e))
554         ((or mod ash) (emit-integer-binop a e))
555         ((or car cdr) (emit-car-cdr a e))
556         (cons          (emit-cons a e))
557         (if (emit-if a e tail))
558         (print (emit-print a e))
559         ((or lambda nlambda) (emit-lambda a e))
560         (quote (emit-quote a e))
561         (otherwise (if (car a)          ; need to be careful about nil....? (
                should this truly be here?... well it is due to the list? check (nil
                is a list))
562                        (emit-funcall a e tail)
563                        (emit-nil))))
564       (emit-return-self a e)))
565
566
567 (defun emit-lambda (a e)
568   (let ((function-class-name (compile-lambda a
569                                              (list 'static-environment  nil
570                                                    'lexical-environment (getf e '
                                                        lexical-environment)
571                                                    'dynamic-environment (getf e '
                                                        dynamic-environment)))))
572     ;; TODO: save this in a private static final field in the class? (if
573     ;; possible of course since when I introduce closures there will be cases
574     ;; where it may no longer be possible to do it that way)
575     (concat "new " function-class-name                          nl
576             "dup"                                               nl
577             "invokenonvirtual " function-class-name ".<init>()V" nl)))
578
579
580 ;; OLD CRAP COMMENT?
581 ;; TODO?: something else than compile-lambda should output whatever amounts to
582 ;; dereferencing a function after actually having compiled the function and
```

```
583 ;; stored it in an appropriate global var (otherwise we would get some strange
584 ;; form of inline call wherever a lambda is)
585
586 (defun emit-classfile-prologue (classname)
587   (concat ".class " classname "
588 .super Procedure
589
590 .field private static final t LLispObject;
591 " %literal-vars% "
592
593 .method static <clinit>()V
594     .limit locals 255
595     .limit stack 255
596
597     ldc_w " dblfnutt "t" dblfnutt "
598     invokestatic Symbol.intern(Ljava/lang/String;)LSymbol;
599     putstatic " classname "/t LLispObject;
600     " %literal-init% "
601     return
602 .end method
603
604 .method public <init>()V
605     .limit stack 2
606     .limit locals 1
607
608     aload_0
609     ldc " dblfnutt classname dblfnutt "
610     invokenonvirtual Procedure.<init>(Ljava/lang/String;)V
611     return
612 .end method
613
614 .method public run([LLispObject;)LLispObject;
615 .limit stack  255
616 .limit locals 255
617 "))
618
619 (defun emit-classfile-epilogue (classname)
620   (concat ".end method" nl))
621
622 ;; Compile a lambda/nlambda in environment e. Store jasmin source in classname.j (
      if supplied, optional argument)
623 (defun compile-lambda (a e . rst)
624   (unless (and (type? 'list a)
625                (or (eq? (car a) 'lambda)
626                    (eq? (car a) 'nlambda)))
627     (error (concat "Are you really sure you passed me a lambda: " a)))
628   (let* ((classname (if rst (car rst) (get-funclabel)))
629          (env (list* 'classname classname e))
630          (%literal-vars% "")
631          (%literal-init% "")
632          (body (case (car a)                              ; since we evaluate the
                      body also for the side effects to %literal-vars%
633                  (lambda  (emit-lambda-body  a env))     ; and %literal-init% we
                             have to evaluate this before emit-classfile-prologue
634                  (nlambda (emit-nlambda-body a env)))))
635     (with-open-file (stream (concat classname ".j") out)
636       (write-string (concat (emit-classfile-prologue classname)
637                             body
638                             (emit-classfile-epilogue classname))
639                     stream))
640     ;; HERE: compile the file just emitted too
641     classname))
642
643 (defun emit-progn (a e tail)             ; NOT TAIL RECURSIVE
644   (cond ((cdr a) (concat (emit-expr (car a) e nil)
645                          "pop" nl
646                          (emit-progn (cdr a) e tail)))
647         (a (emit-expr (car a) e tail))
648         (t "")))
649
650 ;; (nlambda <name> (a b c) . <body>)
651 (defun emit-nlambda-body (a e)
652   (emit-lambda-body (cons 'lambda (cddr a))
653                     e
```

```
654                     ;; we know ourselves by being register 0 which is "this" in
                            Java.  this variable
655                     ;; has the self property set to the parameter-list of the
                            function. emit-funcall
656                     ;; will thus know it can do self-tail-call-elimination and
                            also how the
657                     ;; parameters are to be interpreted (when to construct a list
                            out of some of
658                     ;; them etc. etc.)
659                     (acons (cadr a) (list 0 'self (third a)) nil)))
660
661 (defun emit-lambda-body (a e . rst)
662   (letrec ((static-environment-augmentation (first rst)) ; Optional argument that
          augments the generated static environment if present
663            (args (cadr a))
664            (body (cddr a))
665            (args-roop (lambda (lst alist asm cntr offset) ; TODO: variable arity
                rest-parameter stuff
666                         (if lst
667                             (args-roop (cdr lst)
668                                        (acons (car lst) (list (+ cntr offset) '
                                               static t) alist)
669                                        (concat asm
670                                                "aload_1"                nl
671                                                "ldc_w " cntr            nl
672                                                "aaload"                 nl
673                                                "astore " (+ cntr offset) nl)
674                                        (1+ cntr)
675                                        offset)
676                             (cons asm alist))))
677            (args-result (args-roop args '() "" 0 +reserved-regs-split+)) ; +
                reserved-regs-split+ is the first register that is general-purposey
                enough
678            (asm (car args-result))
679            (alist (cdr args-result))
680            (new-e (list 'classname (getf e 'classname) 'static-environment (append
                alist static-environment-augmentation))))
681       (concat ";; " a nl
682               asm
683               "Lselftail:" nl              ; label used for self-tail-recursive
                      purposes
684               (emit-progn body new-e t)    ; in a lambda the progn body is always a
                      taily-waily
685               "areturn"       nl
686               ";; endlambda" nl)))
687
688
689 ;; An emit lambda for when all arguments are passed to the method
690 ;; plain. Might be good if you want to kawa-style optimize when
691 ;; there's a smaller than N number of args to a function
692 ;; (defun emit-lambda (a e . rst)
693 ;;   (letrec ((static-environment-augmentation (car rst)) ; Optional argument that
          augments the generated static environment if present
694 ;;            (args (cadr a))
695 ;;            (body (cddr a))
696 ;;            (args-roop (lambda (lst alist cntr)
697 ;;                         (if lst
698 ;;                             (args-roop (cdr lst)
699 ;;                                        (acons (car lst) cntr alist)
700 ;;                                        (1+ cntr))
701 ;;                             alist)))
702 ;;            (new-e (list 'classname (getf e 'classname) 'static-environment
703 ;;                         (append (args-roop args '() 1) ; 0 is the very special
          "this" argument, we don't want to include it here
704 ;;                                 static-environment-augmentation))))
705 ;;       (concat ";; " a nl
706 ;;               (emit-progn body new-e t)    ; in a lambda the progn body is always
          a taily-waily
707 ;;               "areturn"       nl
708 ;;               ";; endlambda" nl)))
709
710 ;; TODO: lexical i guess
711 ;; Old emit lambda when i was preparing for JSR-based stuff (might come in handy
       again when you try your hand at TCO)
```

```
712 ;; (defun emit-lambda (a e . rst)
713 ;;   (letrec ((static-environment-augmentation (car rst)) ; Optional argument that
        augments the generated static environment if present
714 ;;            (args (cadr a))
715 ;;            (body (cddr a))
716 ;;            (args-roop (lambda (lst asm alist cntr)
717 ;;                         (if lst
718 ;;                             (args-roop (cdr lst)
719 ;;                                        (concat "astore " cntr nl asm)
720 ;;                                        (acons (car lst) cntr alist)
721 ;;                                        (1+ cntr))
722 ;;                             (cons asm alist))))
723 ;;            (args-result (args-roop args "" '() +reserved-regs-split+)) ; +
      reserved-regs-split+ is the first register that isn't reserved
724 ;;            (asm (car args-result))
725 ;;            (new-e (list 'classname (getf e 'classname) 'static-environment (
      append (cdr args-result) static-environment-augmentation))))
726 ;;       (concat ";; " a nl
727 ;;               "astore 255      ; store return address in variable 255" nl
728 ;;               asm                          ; the argsy stuff
729 ;;               (emit-progn body new-e t)    ; in a lambda the progn body is always
       a taily-waily
730 ;;               "ret 255"       nl
731 ;;               ";; endlambda" nl)))
732
733
734
735 (provide 'compile)
```