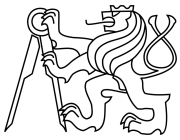
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	LLVM frontend for the Scheme language
Student:	Bc. Jan Noha
Supervisor:	Ing. Radomír Polách
Study Programme:	Informatics
Study Branch:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2016/17

Instructions

Consider implementing a new frontend for the LLVM compiler framework.
Analyse the problem of compiling the Scheme programming language.
Design and implement the Scheme programming language frontend for the LLVM compiler framework.
Test the implemented frontend on sample programs.

References

Will be provided by the supervisor.

L.S.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 17, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Master's thesis

LLVM Frontend for the Scheme Language

Bc. Jan Noha

Supervisor: Ing. Radomír Polách

6th May 2016

Acknowledgements

I would like to thank my supervisor, Ing. Radomír Polách, for his guidance and valuable advice on the topic of this thesis. I would also like to thank my partner and my family for their continued support and encouragement. Last but not least, I am very thankful for all the work of the researchers and developers of LLVM. It is a complex tool and yet relatively easy to use thanks to its well-written code and documentation.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on 6th May 2016 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Noha, Jan. *LLVM Frontend for the Scheme Language.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

Tato práce se zabývá problémem překladu programovacího jazyka Scheme do nativního kódu a zkoumá možnost vytvoření nové přední části překladače pro jazyk Scheme s využitím LLVM. Dále předkládá návrh tohoto překladače a jeho základní implementaci otestovanou na několika ukázkových programech. Nakonec hodnotí výkon přeložených programů v jazyce Scheme ve srovnání s již existujícími překladači a interprety.

Klíčová slova Scheme, LLVM, dynamický jazyk, překladač, běhové prostředí

Abstract

This thesis analyzes the problem of compiling the Scheme programming language to native code and explores the possibility of building a new Scheme compiler frontend using the LLVM framework. It proposes a design of the frontend and presents a prototype implementation of it, tested on various example programs. It also evaluates performance of the compiled Scheme programs in comparison with other existing compilers and interpreters of the same language.

Keywords Scheme, LLVM, dynamic language, compiler, runtime system

Contents

List of Figures

List of Tables

Introduction

Scheme is a functional programming language from the Lisp family, invented during the 1970s by Gerald J. Sussman and Guy L. Steele. In its nature, it is an implementation of an "Extended Lambda Calculus".[1]

As such, there are many implementations of Scheme today. Because of its minimalist design and high extensibility (system of macros that can be used to define new syntax), there are also many dialects.

Typically, Scheme programs are interpreted directly from source code, or dynamically compiled using a JIT compiler. There are also implementations of Scheme capable of translating the input source code into C.[1] However, at the time of writing, there is no working Scheme compiler that would generate native code directly (without the intermediate step of compiling to C) and use the LLVM framework to do it, although similar project exists for Common Lisp.[2]

Goal of the thesis

The objective of this thesis is to design and implement such a compiler along with a runtime environment for the language. The LLVM framework will assist us in generating native code for any of the targets it already supports. All we have to do is provide a frontend which will translate Scheme into the LLVM internal representation (IR) and also a library that can interact with the generated code and provide core functionality such as memory allocation, error handling but also compiler invocation (to support dynamic evaluation and translation of code).

[1] CHICKEN Scheme – available from `https://www.call-cc.org/`

[2] Clasp – available from `https://github.com/drmeister/clasp`

Structure of the thesis

In chapter 1, we give an overview of the LLVM compiler framework, we discuss properties of the Scheme language and we analyze the main challenges associated with translation of Scheme into the LLVM assembly (IR).

Chapter 2 contains a detailed description of the compiler and runtime system design we propose.

Chapter 3 gives us an insight into some of the implementation details, implementation problems and their solution. It also shows the project's structure and its current status.

In the last chapter, we outline the various tests and sample programs we have used during implementation and we also evaluate performance of our Scheme implementation.

Chapter 1

Analysis

1.1 LLVM compiler framework

1.1.1 Overview

LLVM is a collection of libraries and tools that make it easy to build compilers, optimizers, Just-In-Time code generators, and many other compiler-related programs. LLVM uses a single, language-independent virtual instruction set both as an offline code representation (to communicate code between compiler phases and to run-time systems) and as the compiler internal representation (to analyze and transform programs).[2]

Today, LLVM is becoming a popular platform for implementing new programming languages as well as porting the existing ones to use it. One of the most notable examples is the Clang frontend with a single unified parser for C, Objective C, C++ and Objective C++.[3] Among other projects using LLVM there is Rubinius (Ruby with Just-in-time compiler), Julia (a new dynamic language for technical computing) or Rust (safe, concurrent systems programming language).[4]

The advantage of using LLVM is that when we want to implement a new compiler for our language of choice, all we have to do is to provide a frontend which translates the language into the LLVM platform independent internal representation (IR). Then we can apply a series of optimizations and transformations to the code (already implemented to work on the LLVM IR). As the last step, we simply choose a backend that converts the optimized code to assembly of the particular machine architecture. That means we have just ported the high level language to multiple platforms at once.

Conversely, we can extend LLVM with a new backend and suddenly all the languages already compiling to LLVM IR are also able to compile to our newly supported machine code.

This is the full realization of the classical compiler model consisting of independent frontends, a common optimizer in the middle and backends. LLVM

really tried to make this theoretical model work and it succeeded.[5]

It is also relatively well documented and the codebase is comprehensible for newcomer developers, which is why we have decided to use it for a Scheme compiler.

The LLVM project has grown considerably since its conception and we will be using only parts of it. Namely the library for IR generation, the target-independent optimizer, static and just-in-time compiler (for native code generation).

For the static compilation, there is a tool called *llc* (LLVM system compiler) which takes the IR as input and emits native code in the form of text (assembly) or binary (native object files).

There is also an LLVM linker (*lld*) but it is still in its early stages of development, so we will use *clang* linker instead.

The generation of IR and JIT compilation will be handled by the LLVM's C++ API.[6][7]

1.1.2 LLVM Intermediate Representation

Now let us look more closely at the LLVM assembly language. It is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing high-level languages cleanly. It is the common code representation used throughout all phases of the LLVM compilation strategy.[8]

There are three different forms of the language: an in-memory compiler IR, an on-disk bitcode representation (suitable for fast loading by a JIT compiler), and a human readable assembly language representation.

We can use the textual representation as a debug output of our compiler, while the bitcode is suitable for piping the IR output from the frontend to the *llc* tool which then converts it to a native object file for us. The in-memory form of the assembly is represented by a set of C++ objects which can be generated conveniently with the help of LLVM API.

So, the frontend we are going to design and implement, will use this API to create the whole program representation in memory and then LLVM will be able to convert it to any of the remaining forms.

In the LLVM IR, the basic compilation unit is called **Module**. That roughly corresponds to a notion of one source file which is to be compiled to one object file. A module consists of global variables, global constants and global functions with local variables. There is also a symbol table containing those functions and globals we choose to be visible to other modules. In the C++ API, functions, variables and constants have their corresponding classes which are all derived from a base class **Value**. For that reason, we will refer to them as values. Identifiers prefixed with @ are used for the globals, whereas identifiers of locals start with %.[8]

Global variables and constants are always represented by a pointer to a memory location. Local variables can hold a value directly or they can also point to memory. In that case, it is either memory holding a previously declared/defined global or a memory location at the current function's stack frame, which we can allocate using the **alloca** instruction.

Local variables have to be in the SSA form. That means each variable is assigned exactly once and has to be defined before it is used. The LLVM IR handles definition and assignment in one step: Each instruction assigns its result to a new variable. Even the **store** instruction has a return value but it has no meaning and must not be used.

If we want to generate code which extensively operates on local variables and we would like to describe those operations more naturally, without the restrictions imposed by SSA, we can use stack variables made by **alloca** and operate on them with **load** and **store**. Assignment through **store** is not limited because the pointer itself does not change. That sort of code is not the most efficient. LLVM can, however, optimize it later by mapping in-memory variables to available CPU registers.

Every value in LLVM IR has a type. There are single value types (*i1, i8, i32, double*), pointer types, array types, structure types, function types, etc. We can use casts to convert between them.

To represent integers, for example, we use the numeric *i<num>* types, where *num* determines their bit width and can be anything between 1 (boolean) and $2^{23} - 1$.[8] In practice, we will use powers of two. For floating point numbers there are types *half, float, double, fp128, ...* of different widths. Strings are constructed similarly as in the C language – with arrays of *i8* (equivalent to *char*). There is one important difference between C and LLVM – integer values have no sign. Instead it is the operations that interpret their operands as either signed or unsigned. That means we have multiple variants of comparison instruction (**icmp**).

Generally, there are all the instructions we would expect from a real hardware instruction set. However, this instruction set is more high-level and also typed as we have already mentioned. For example, there is one instruction which handles function calls including the passing of arguments. You cannot push arguments manually onto the stack (in fact, there are no **push** and **pop** instructions) or place them in registers (which registers?) as that would be platform dependent. Instead you provide the arguments as operands to the instruction, along with a function pointer which holds information about the return and argument types, plus a selected calling convention.

LLVM defines several calling conventions:

- The C calling convention (compatible with C),

  which supports functions with a fixed or variable number of arguments.

- The fast calling convention,

which attempts to make calls as fast as possible.

- The cold calling convention,

  which attempts to make code in the caller as efficient as possible under the assumption that the call is not commonly executed.

- and others ...

Apart from calling conventions there are all sorts of function and variable attributes that can be used for a very low-level control. That includes linkage types (internal, external, weak, ...), visibility styles (default, hidden, protected), DLL storage classes (dllimport, dllexport), exception handling attributes, tail call flags (which tell LLVM whether to perform a tail call optimization), alignment specifiers and so on.

Example code written in the LLVM IR

```
%struct.IR_String = type { i32, i32, [15 x i8] }
%struct.IR_Type = type { i32 }
%struct.IR_Cons = type { i32, %struct.IR_Type*, %struct.IR_Type* }
%struct.IR_Int = type { i32, i64 }
%struct.IR_Float = type { i32, double }
%struct.IR_Symbol = type { i32, i32, [1 x i8] }
%struct.IR_Func = type { i32, i32, %struct.IR_Type* (i32, ...)* }
%struct.IR_Vec = type { i32, i32, [1 x %struct.IR_Type*] }

@igc = constant %struct.IR_String {
  i32 3, i32 4, [15 x i8] c"example string\00"
}, align 4

@nil = constant %struct.IR_Type { i32 2 }, align 4

@cell = constant %struct.IR_Cons {
  i32 7,
  %struct.IR_Type* bitcast (%struct.IR_String* @igc to %struct.IR_Type*),
  %struct.IR_Type* @nil
}, align 8

@gvar = global %struct.IR_Type* null, align 8
@glob = common global i32 0, align 4
@exit_code = internal global i32 0, align 4

define void @func() #0 {
entry:
  %t = alloca %struct.IR_Type, align 4
  %i = alloca %struct.IR_Int, align 8
  %fl = alloca %struct.IR_Float, align 8
  %str = alloca %struct.IR_String, align 4
  %sym = alloca %struct.IR_Symbol, align 4
  %c = alloca %struct.IR_Cons, align 8
  %fn = alloca %struct.IR_Func, align 8
  %v = alloca %struct.IR_Vec, align 8
  %loc = alloca i32, align 4
  %pc = alloca %struct.IR_Cons*, align 8
  %tag = alloca i32, align 4
  store i32 1, i32* %loc, align 4
  %0 = load i32, i32* %loc, align 4
  %add = add nsw i32 %0, 1
  store i32 %add, i32* %loc, align 4
  %1 = load i32, i32* %loc, align 4
  %add1 = add nsw i32 %1, 1
  store i32 %add1, i32* @glob, align 4
  %2 = load %struct.IR_Type*, %struct.IR_Type** @gvar, align 8
  %tag2 = getelementptr inbounds %struct.IR_Type, %struct.IR_Type* %2, i32 0, i32 0
```

```
  %3 = load i32, i32* %tag2, align 4
  store i32 %3, i32* %tag, align 4
  ret void
}

define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %retval = alloca i32, align 4
  %argc.addr = alloca i32, align 4
  %argv.addr = alloca i8**, align 8
  store i32 0, i32* %retval
  store i32 %argc, i32* %argc.addr, align 4
  store i8** %argv, i8*** %argv.addr, align 8
  %0 = load i32, i32* %argc.addr, align 4
  %cmp = icmp ne i32 %0, 2
  br i1 %cmp, label %if.then, label %if.end

if.then:
  store i32 1, i32* %retval
  br label %return

if.end:
  %1 = load i32, i32* @exit_code, align 4
  store i32 %1, i32* %retval
  br label %return

return:
  %2 = load i32, i32* %retval
  ret i32 %2
}
```

As we can see, it is quite verbose. Luckily, we don't have to write such code by hand. There is the C++ API for that purpose. Furthermore, if we are not comfortable with the LLVM assembly at first, or we are unsure as to what particular instructions to choose for our application and how to properly generate them with the API, we can write the same code we want to obtain, in any language LLVM already supports (C, for example). Then compile it to the intermediate representation (*clang* can do that given a special command line option) and at last use the *llc* tool with the "-march cpp" option. This way, LLVM selects the *cpp* backend and that backend converts IR to the exact LLVM C++ API calls needed to generate the given code. In other words, it can generate parts of the compiler for us as long as we can describe what the result should be in any of the supported high-level languages. This kind of feature is quite unique.

While the *cpp* backend is useful, it still produces code that is not the most concise. LLVM is still evolving and there is a class **IRBuilder** which is not used in the *llc* output. It is, however, used in the "Kaleidoscope: Implementing a Language with LLVM" tutorial [7] and we can see that it is more convenient to generate instructions with its help. One of the advantages of **IRBuilder** is that it maintains an insertion point within the module currently being generated. We can just set the insertion point to a particular basic block and then start generating instructions via the builder. They are going to be appended to the chosen block automatically. We can choose a different basic block at any time – instructions do not have to be generated sequentially. There are even methods for saving and restoring the current insertion point (useful when generating nested functions for example).

1.1.3 LLVM compiler

Generating the LLVM assembly is the most important part. After that, native code generation is practically for free. Again, with the help of C++ API, we can setup the whole backend compilation phase: choose the target architecture, relocation model (whether to emit position dependent or independent – relocatable code), the output format, etc.

For static compilation, we will use the ready-made *llc*. Otherwise, when there is need to dynamically generate new code at runtime and execute it immediately (JIT compilation), we can programmatically setup the backend to perform the same translation with one difference: output is not dumped to a file but instead it is made accessible from memory directly.

The compiled module behaves like a newly loaded dynamic library on which we can perform symbol lookup. So, when we ask for a symbol associated with particular function, a pointer to that function is returned to us and it is ready to be called. Additionally we control the way our new dynamically compiled module can perform its own symbol resolution when it depends on any previously compiled code or data (either JITted or from a native library). In the tutorial [7], this is solved by loading every new compiled module to the address space of the compiler application, so all the symbols compiled so far are immediately available to both compiler and every compiled module.

1.2 Compiling the Scheme language

Scheme is a functional programming language that follows a minimalist design philosophy specifying a small standard core with powerful tools for language extension. Along with Common Lisp, it is one of the main dialects of Lisp. The principal design ideas of languages from the Lisp family are based on Lambda calculus [9]

1.2.1 Basic properties [10]

Syntax

The syntax of Scheme is very simple. Both code and data are described in the same way – using atoms and lists (the language is homoiconic). List is just a collection of atoms or other lists, enclosed in parentheses.

In the following example, we define a global binding between the *foo* symbol and number 1. Then we define a function named *bar* which takes one argument x and returns its sum with the value of *foo*. Finally, we call the function with number 2 as its argument.

```
(define foo 1)
(define (bar x) (+ x foo))
(bar 2)
```

Generally, there are definitions and expressions in Scheme. They can be implemented either as special built-in language constructs (like *define, if, quote*), as syntactic macros derived from other constructs, or as functions. We can see that even arithmetic operators are simple functions, so any arithmetic expression will be written in a prefix form, essentially.

In-memory representation

The natural representation of Scheme code and data is a linked list. That list is composed of the so called *cons* cells and atomic values. Each *cons* cell has two slots:

1. *car* (containing a pointer to the list element)
2. *cdr* (containing a pointer to the rest of the list)

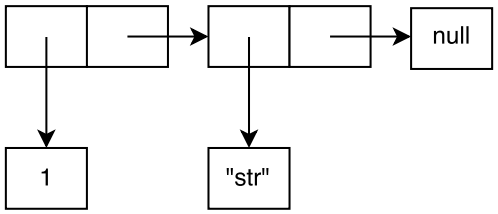


Figure 1.1: Scheme list memory layout

The *car* slot can point to an atom (number, string, symbol, ...) or another *cons* cell (nested list). The *cdr* slot usually points to *cons* cell or *null* (marking the end of list). It can also point to atom, however, forming a special type of list, the *dotted* list (because in the source text, it has a dot in between the last two elements).

For code, the linked list would be useful if we were to implement an AST interpreter. In that case, the linked list itself could serve as AST and we would interpret it easily by recursive traversal.

However, we are going to generate LLVM IR from Scheme code and for that purpose we will probably need a custom AST which can be decorated with more information.

Functions

Functions are a first-class object in Scheme. That means, we can manipulate with them like with the rest of built-in types. We can assign functions to variables, pass them as arguments to other functions or return them from functions. We can also create anonymous *lambda* functions.

Lexical scope

Unlike earlier Lisps, Scheme is lexically scoped: All possible variable bindings in a program can be analyzed by reading the text of the program without consideration of the contexts in which it may be called.[9] That means all the

bindings can be statically determined during compilation which is convenient when we want to write a compiler instead of an interpreter. Symbolic references are resolved and all that is left are anonymous values (globals or locals) with the corresponding pointers to them (instead of the original names). However, we need to preserve names of at least the global values so that we can refer to them from other separately compiled programs. This becomes an issue when we want to support library functions implemented in Scheme or compilation of new code at runtime.

Tail recursion

While Scheme has the means to write ordinary iteration using the *do* expression, it is preferable to use recursion (in accordance with the functional paradigm). Because Scheme puts emphasis on recursive functions it requires the compiled programs to perform tail call optimization. That means the resulting code supports a special type of call which does not return back to the caller function (in case the call was the last instruction followed only by return instruction) and furthermore the called function reuses the caller's stack frame. That allows for efficient iteration through recursion which uses constant stack space as opposed to linear stack space with regard to the number of calls.

LLVM supports tail call optimization but the support varies depending on the calling convention. The classic C calling convention cannot be optimized due to architectural reasons, fast calling convention can, but it does not support variadic functions. There are also two specialized calling conventions developed for Haskell and Erlang but they have worse overall performance (they make even regular calls more expensive).[8]

However, LLVM can also perform tail call elimination which means it takes a recursive function, analyzes it and transforms it to direct iteration if possible. Therefore, there is no actual call in the resulting assembly. That should be the preferred way if we want to generate as fast code as possible while adhering to the language specification.

Type system

Scheme is a dynamically typed language. Each object has its one implicit type (integer, float, string, symbol, function, ...) but variables and function arguments (generally all identifiers bound to those objects) do not. The type of an identifier is dynamic in a sense that any number of objects with different types can be bound to the same identifier throughout its lifetime (one at a time, of course).

That means there are no type annotations in a Scheme program. When we define a function which takes a certain number of arguments, they can be arguments of any type. In order to be able to work with the arguments,

there has to be a way to query their types. Alternatively, we can make an assumption about the type and just pass the variable to another function. Eventually, some form of type checking has to be done, however. For that purpose, every object contains runtime type information and most of the type checking is handled no sooner than the program is running.

On the other hand, in some special cases the compiler can infer types using control flow analysis. It can observe the actual arguments passed to functions in our compiled program and specialize the calls (or even inline them). That could be desirable for arithmetic functions which should be optimized for speed.

For example, if we prove by the analysis that an instance of a call to + function always operates on integers only, we can call a specialized implementation that doesn't do type checking or even emit a simple **add** instruction. Another example is a call to function through a function object obtained from a variable (or function argument). Without the control flow analysis, destination of the call would certainly have to be resolved at runtime, but we can examine all variants determined by the actual usage in the code and perhaps emit several versions of the function which makes this indirect call, with each call resolved to the particular destination.

As LLVM was designed with statically typed languages in mind [5], we have to work around this. All the Scheme objects will be modeled as tagged structures and variables will point to those structures.

So, each type has a unique ID and each structure has a tag (as its first element) containing that ID. By looking at that tag, we can tell types of the structures apart. In the C language, we would then probably use union to access each type representation conveniently. In LLVM, there are no unions, so we simply use type casts.

1.2.2 Features [10]

Closures

When we define a function in Scheme, a local scope is created that has access to the function arguments but also to all variables defined in parent scopes. This is intuitive and the majority of programming languages work in a similar way.

When we compare Scheme with C, we can see that C functions can also access variables from the outer parent scope. That scope is always the global scope, however. We do not allow nested functions. This has a clear advantage: All the variables accessed by a function are either local (that includes arguments – both are saved in the current stack-frame) or global (saved in the program's data section – available at all times).

In Scheme, it is more complicated. First-class functions can and often are defined inside other functions (lambda functions are almost always used in this

way) and that means a function can potentially access variables which are not local nor global. In the C terms, that means we have to somehow access not only our own stack frame but also the stack frame of the function in which our function was defined and possibly even this function's parent frame.

When a function behaves in this way (accesses variables defined inside or passed to other functions) we say it has a *closure* (and that closure contains *captured* variables). In practice, it means we have to store a *context* pointer to the function object. That context pointer points to the frame of the parent function and it will be passed to our function every time it is called (which can be in a different context). We must understand that there can be multiple function objects for the same function but with different context pointers. This can be demonstrated on a simple example:

```
(define (fn-factory y)
  (lambda (x) (+ x y)))
```

Each time we call *fn-factory* with a specific y, it returns a new function object with a context pointing to that particular value of y. Now we have to think about where all the versions of y are actually stored. Clearly, it cannot be the stack frame of *fn-factory* since that is valid only until we return from the function. We have to take care of all local variables (or arguments) that should outlive their respective function call execution time. Solution to this issue is presented in section 2.1.4.

Dynamic calls

Scheme function calls resemble the *application* operation in Lambda calculus. There is a list of expressions, we take the first one and apply the rest on it.

When a compiler processes such a call, it immediately knows how many function arguments there are, and it may or may not know which particular function is to be called. That leads to a direct or an indirect call. We can imagine how both of these calls would look like in C and it will be the same in LLVM IR. The function signature is known and the function pointer is either a constant or a variable.

There is, however, a more general way to call functions in Scheme – using *apply* (which itself is a function). We give *apply* the function we want to call and a list of arguments. Now the whole argument list is probably a variable (otherwise we would not use *apply* in the first place) and that complicates things for the compiler.

As we have mentioned before, LLVM has a **call** instruction which takes arguments as operands and we have to know how many there are, at compilation time. Not even variadic functions can help us. Although they can take variable number of arguments, at call site we have to enumerate them. There are several ways to work around this restriction [11]:

1. Choose a large enough maximum number of arguments and implement *apply* using a switch with cases for every number up to the limit.

2. Implement *apply* in assembly (for every target CPU we want to support).

3. Use a universal signature for all functions (always pass a pointer to an array of the actual arguments which is stored elsewhere).

The third approach seems to be the most reasonable although it means we give up the possibility of passing arguments in registers (so there is a performance cost). Limiting the maximum length of a list usable in the *apply* function is not a good idea and if we were to implement parts of the language in the target assembly, we would lose the advantage of LLVM's platform independence. The ideal solution would be to extend LLVM itself with this new functionality but that is beyond the scope of this project.

Details of the proposed solution (based on the third approach) follow in section 2.1.4.

Dynamic compilation

One of the defining features of Scheme which we have already mentioned is its homoiconicity. Both code and data are expressed by the same syntactic form – lists. Due to this property, we can write programs which dynamically construct new code as data. This can be achieved very elegantly using the core language functions (*cons* or *list*):

```
(define code_as_data (list 'lambda (list 'x) (list '* 'x 'x)))
```

In this example, *code_as_data* refers to a data list which can be interpreted as a simple lambda function that implements square operation.

To actually run this code, we need to evaluate it first. Scheme has a function *eval* which does exactly that. It takes the data we want to convert to code and a namespace object. The namespace provides an environment in which bindings exist and therefore symbolic references can be resolved. We can, for example, call the *make-base-namespace* function which returns a fresh environment populated with bindings to basic library functions. That namespace can be passed to *eval* and if the evaluated code makes any new definitions, the namespace will be updated with the associated bindings.

To support this mechanism, our LLVM compiler has to be accessible from the runtime library and it should implement just-in-time compilation of Scheme data. The namespace object corresponds to environment object used by the compiler for static symbol resolution (described in section 2.1.3) but in case of dynamic compilation at runtime, it is independent and can be reused across multiple calls to *eval*. For more information, refer to section 2.2.3.

1.2.3 Memory management [12]

The majority of Scheme objects is allocated on the heap and accessed through pointers (maybe with the exception of primitive types such as numbers which can fit into the pointer itself). The language was not designed with manual memory management in mind, so you typically create an object, then use it for a while assuming it is accessible as long as needed but after you are done with it, you do not free its allocated resources explicitly. Instead, there is an automated garbage collector which tracks references to objects and when it finds out there are no references to a particular object, it can safely delete it.

There are many strategies used for garbage collection.[13] Generally, they must provide a mechanism to distinguish live objects (with at least one reference pointing to them) from the dead ones (with no references left). That can be accomplished by scanning pointers in the running program.

Allocated objects typically form a recurrent tree hierarchy. One object can point to other objects. Primitive types such as numbers and symbols (pointing to nothing else) represent the leafs of this object tree. For garbage collector, however, roots are the most important, because the pointer scanning starts there (before scanning the trees completely by recursion).

Where are the roots located? Typically on the stack (we must scan all the active function stack frames), in CPU registers, and in the data section where globals are defined.

The last question remains: How do we recognize what is and what is not a pointer? The garbage collector can either guess (pointers are typically aligned, they have a certain minimum address, and so on ...) or we can provide it with additional information.

LLVM has support for garbage collectors.[14] You must provide your own implementation of a chosen strategy but it can assist in generating code that guarantees proper interoperability. For example, you can generate *stack maps* in LLVM (containing the necessary metadata for an informed root scanning) or emit read/write barrier intrinsic instructions (needed for the more complex GC algorithms).

Another option is to use the Boehm-Demers-Weiser conservative garbage collector [15] which does not require any compiler support (it is originally intended is a malloc/free or new/delete replacement for C/C++). This collector is available as a dynamic library and we can just link it to our runtime and implement object allocation with its help.

If we wanted to use our own GC, the implementation of pointer scanning would be platform dependent (accessing registers and stack directly is only possible in assembly), so using the GC library certainly is a simpler option. The only major disadvantage of the conservative collector is that it treats everything that looks like a pointer as a real pointer which may result in occasional leaks, although the authors claim it is unlikely that this will result in a leak that grows over time.[16]

Chapter 2

Design

2.1 Compiler

2.1.1 Lexical analysis

We start with lexical analysis like we would in any other compiler. Scheme is a particularly simple and elegant language, so this layer of the compiler will be quite basic.

We need to design a tokenizer which will read the input source code and give us a sequence of tokens. Each of the tokens has a type and optionally an additional parameter.

There is a total of six token types. We list them in table 2.1. **Numerical** types, each with its value, **String** type with text value, **Symbol** type with name which is used to refer to Scheme objects, **Keyword** type with the particular keyword specified and a special **Error** type used to indicate invalid tokens such as unterminated string literal.

Table 2.1: Token types

Type	Integer	Float	String	Symbol	Keyword	Error
Param.	value	value	value	name	name	–

This is the list of keywords:

`(, ), #t, #f, null, define, lambda, quote, if, let, ', and, or`

The tokenizer skips white-space characters and lines starting with ";" (comments). Then it expects a token. Each token is terminated by a white-space character or the right parenthesis keyword. Integer numbers consist exclusively of digits and an optional sign, floating point numbers consist of digits, sign and a decimal point. Strings are enclosed in double quotes and support C-like escape sequences (`\n, \t, \"`, ...). All unquoted strings not corresponding to the aforementioned types are either keywords or symbols.

In our compiler we will refer to the tokenizer as **Reader**. The Reader behaves like an input stream. We specify the input and then we can query **next token** and **current token** repeatedly. That means it has an internal state corresponding to current position in the source.

We define three different Reader types:

1. **FileReader** that reads from input file stream
2. **StringReader** that reads directly from a given string
3. **ListReader** that reads from already compiled Scheme data

FileReader is used when we want to compile source code located in a text file. StringReader processes in-memory string given from a command line argument for example. ListReader is needed for runtime code evaluation – when we convert Scheme data to Scheme source code and subsequently compile it using our frontend and LLVM JIT (see 2.2.3).

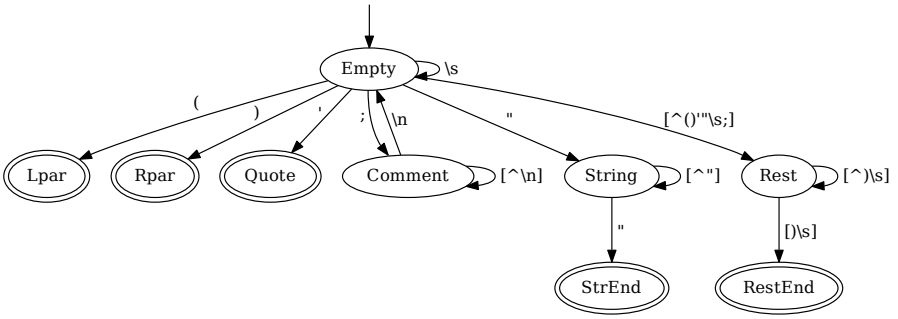


Figure 2.1: Token reader automaton (simplified)

Both FileReader and StringReader read its input one character at a time and process the characters using a finite state automaton 2.1. ListReader, however, works with a potentially nested atom/list tree-like structure loaded recursively from memory. There is no need to identify tokens from characters but rather convert runtime types to their corresponding token types. A stack is used to process elements loaded from the program's memory.

As a first step, we push the root of the data expression onto the stack. Each time the next token is queried, the top element is popped and examined. If it is an atom, we simply return the correct token type. In case of list we traverse its elements, push them in reverse order with a special "end-of-list" marker at the beginning and return the "(" keyword. When the EOL marker is reached, we return the closing ")".

2.1.2 Syntax analysis

Using the Reader we get a higher level representation of the source code consisting of lexical tokens which we can now use as terminal symbols in the language grammar.

Our variant of Scheme is based on the following grammar:

```
prog = form { form }
form = "(" def ")" | expr | "(" "require" str ")"
def = "define" sym expr
    | "define" "(" sym symlist ")" body
expr = atom | "(" callsyn ")" | "'" data
callsyn = "lambda" "(" symlist ")" body
                | "quote" data
                | "if" expr expr expr
                | "let" "(" bindlist ")" body
                | "and" { expr }
                | "or" { expr }
                | expr { expr }
atom = str | sym | int | float | true | false | null
symlist = { sym }
bindlist = { "(" sym expr ")" }
data = atom | ( "(" list ")" ) | "'" data
list = { data }
body = { form }
```

We use a recursive descent parser to generate abstract syntax tree from the tokens. Each node in the AST then corresponds to one particular language construct.

A scheme program is a sequence of forms, where each form is either definition or expression.

Definition

There are two types of definitions. The first one is used to define a binding between symbol and expression. With the second one, we can create named functions. It is actually a syntactic shortcut equivalent to the definition of binding between symbol and lambda function.

The two AST nodes created from definitions are **ScmDefineVarSyntax** and **ScmDefineFuncSyntax**. ScmDefineVarSyntax node is a parent of two other nodes representing a symbol name and the expression bound to the symbol. ScmDefineFuncSyntax node has three children nodes: function name, argument list and body list (because the body is a list of forms).

Expression

Expression can be atomic (string, symbol, number, boolean, null), it can be a function call, data (quoted atom or list) or a special construct such as `lambda`, `if`, `let`, `and`, `or`.

All non-atomic expressions are essentially lists. If the first element of the list is a keyword, the list has a special meaning. Otherwise it is a function call. The only exception is a quoted list (preceded by ' keyword) or any list nested in other quoted list. Those lists represent data.

AST nodes for atoms are quite straightforward. String, symbol, integer and float nodes are named after the corresponding token types and contain the numeric values, string values and symbol names.

ScmLambdaSyntax node represents an anonymous lambda function definition. It points to argument list and body list nodes but has no name.

ScmQuoteSyntax points to the data node (list or atom).

ScmIfSyntax points to condition expression, *then* expression and *else* expression.

ScmLetSyntax points to binding list (sequence of local binding definitions) and body list.

We also have nodes for logical *and* and *or* expressions.

Lists are constructed with **ScmCons** nodes, each pointing to *car* and *cdr* nodes. There are two special types of lists:

1. **Symlist** that can contain only symbols (function argument names)
2. **Bindlist** that contains symbol – expression pairs (used to define bindings for the local scope of body)

All the node types inherit from a base type **ScmObj**. Graph 2.2 shows the hierarchy.

The parser can reveal syntax errors such as unterminated lists, unexpected tokens in special lists, missing list elements. It further ensures that the last form of each body is an expression. We cannot permit *let* expressions or function bodies with definitions only. They must always return something.

More kinds of errors are handled by the semantic analysis pass.

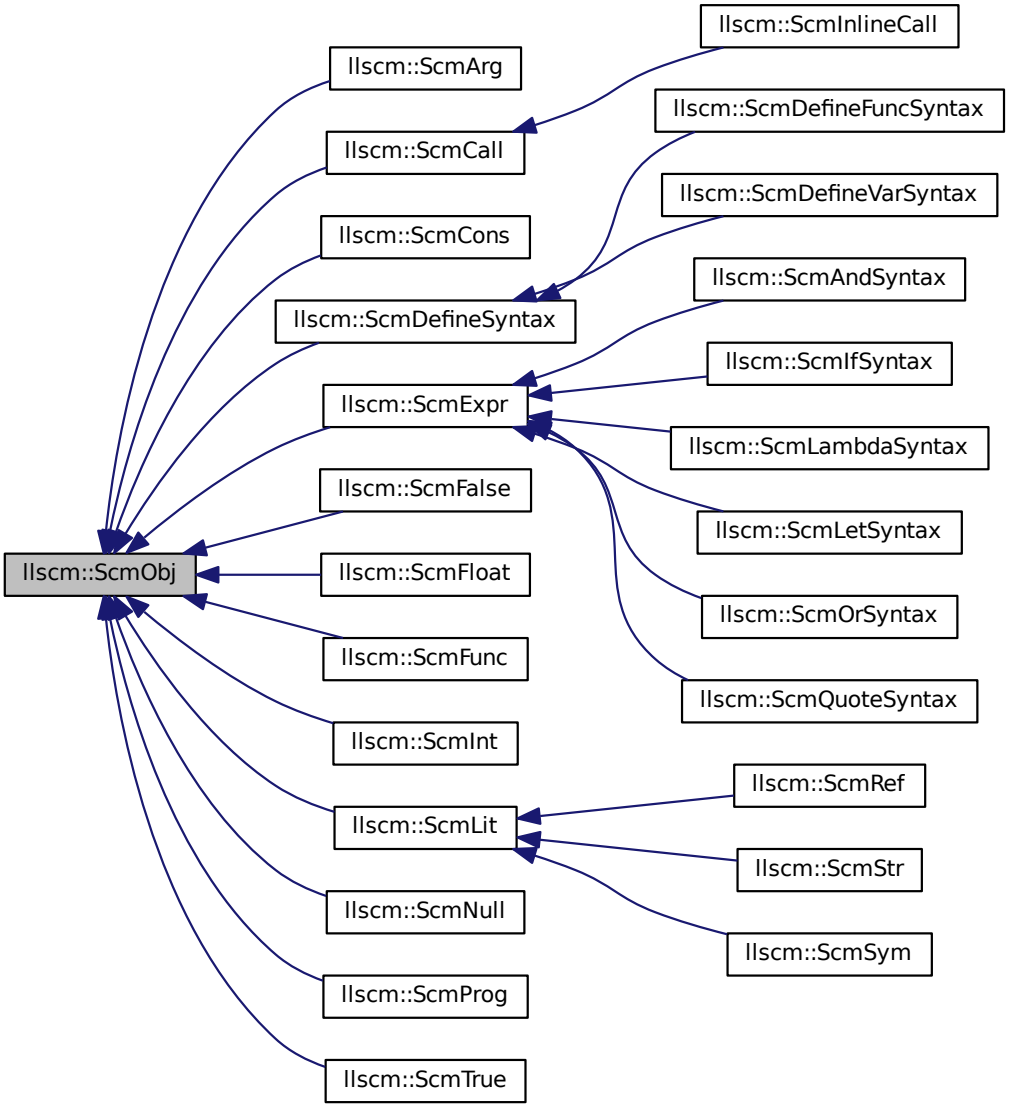


Figure 2.2: Simplified inheritance graph of the AST node types

2.1.3 Semantic analysis and AST transformation

With the AST generated, we can traverse it and perform a limited semantic validation as well as transformation aimed at simplification of the AST before code generation.

For example, all the **ScmDefineFuncSyntax** nodes are decomposed to **ScmDefineVarSyntax** and **ScmLambdaSyntax** like this:

```
(define (foo x y) (+ x y))  =>  (define foo (lambda (x y) (+ x y)))
```

Furthermore, lambda functions have to be named (by a unique identifier), their definitions moved to separate forms and symbolic references added at the original location of the in-place definition. For example:

```
(map (lambda (x) (* x 2)) lst)
  =>
(define __lambda#0 (lambda (x) (* x 2)))
(map __lambda#0 lst)
```

We call this phase of translation the *compile-time evaluation*. Each node of the AST has its **CT_Eval** method whose purpose is to recursively call **CT_Eval** on its children and then return either itself or a modified version of itself. That is exactly what is happening in the two examples. Additionally, the second example, where we create a new definition, shows that we need to have access to the whole program representation (**ScmProg** node with the list of forms) in order to extend it at the top level.

What is even more important is the mechanism used to resolve symbolic references. That is the main task of semantic analysis. When we find a definition in the AST, it means we're supposed to create a new symbol – expression binding at a specific scope. Then, each time we see the symbol declared in that definition, we can link it to the actual expression. For that purpose, we use *environments*.

Compile-time evaluation of each AST node takes place in an environment. At first, we create a global top-level environment (**ScmEnvironment** object) and populate it with symbol bindings to externally available runtime functions. Internally, the **ScmEnvironment** contains a map of symbols (keys) and AST nodes (values). Some of the nodes are not attached to our tree yet (the external functions). The environment also contains a pointer to the **ScmProg** node (so that lambda definitions can be moved) and optional pointers to parent environment and context (function that uses this environment as its local scope).

As we recursively call **CT_Eval** starting from the root of the AST, we pass around the environment, so it can be updated each time a definition is encountered. However, not all definitions are global. When a definition is located inside a function, the binding is valid in the body of that function only. Similarly, local bindings are created for each *let* block. For that reason, **CT_Eval** methods of functions and *let* blocks handle the creation of new environments.

Each new environment beside the global one must have a parent. The parent environment is always the one which was passed to **CT_Eval** where we currently set up the new child environment. So, when we evaluate a global function, for example, we create a new environment whose parent is the global environment and pass that local environment to the function body's **CT_Eval**. It is automatically local because when a symbol lookup occurs, we search for the binding in the current environment and its parents (not children). In fact, the environment does not even have a reference to its children.

When **CT_Eval** is called on a symbol (**ScmSym**), the symbol lookup takes place. We query the current environment for the object bound to the symbol. It can return another symbol when multiple bindings are chained. In that case, we want to resolve that symbol as well. If the program consists of

```
(define a 1)
(define b a)
(define c b)
(+ c 2)
```

and we ask for the binding of *c*, we get number 1. Each resolved symbol is replaced by a reference node (**ScmRef**) which contains name of the symbol, pointer to the referenced object and a relative position of the referenced object with respect to the symbol location. The relative position is defined as the number of parent environments (only those belonging to functions) which were traversed before we have found the referenced object.

```
(define x)
(define (foo c)
  (lambda (b)
    (lambda (a)
      (+ a b c x))))
```

In the example above, object bound to *a* has a relative position 0 because it is the innermost function's argument (*a* is bound to a special node **ScmArg** – the actual expression bound to argument varies at runtime).

Position of *b* is 1 (it is defined one level up) and position of *c* is 2 (analogous). We need this information because references with a position greater than zero are captured in a closure and closure data access requires special support in the code generator (see 2.1.4). Note that the position of *x* is not 3. Global variables (defined in the top-level environment) are assigned a special position value (a negative number).

To sum it up, referenced objects can be either global (position < 0), local (position $= 0$) or captured (position > 0).

The main benefit of having **ScmRef** nodes is that the compiler will not be confused by redefinitions of the same symbol name. This works because we create bindings and resolve them in one pass as we sequentially traverse the code (in the form of AST). So, if there is one definition of *foo* after which several references to *foo* follow and then we discover a redefinition of the same

symbol, all of the previous references will have already been resolved and only then we modify the environment for any future references.

There is one special case we have to handle: **forward references**. There are times when forward references can be avoided by changing order of the source expressions and definitions. However, in some cases that is not possible. Take for example two functions which call each other. In Scheme, there is no dedicated construct for function declaration, so instead of solving the problem like we would in C/C++, the compiler must detect forward references implicitly.

When we encounter symbol which is not bound to any object yet, instead of immediately raising an undefined symbol error, we just make a **ScmRef** pointing to nothing and we save a pointer to it (along with a pointer to the current environment), in a special table located in the top-level environment. In that table, we can later look up that reference by symbol name when the symbol gets defined, rerun symbol resolution and update that reference before deleting the corresponding table entry.

With this resolution system in place, we can of course detect references to undefined symbols (those that remain in the forward reference table after the whole AST has been processed) and report error to the user.

Another important function of semantic analysis is to annotate the AST with useful information for the code generator. Information such as the position of referenced objects, described above. Furthermore we need to distinguish between direct and indirect calls.

In Scheme, function is a first-class object which we can suspect just by looking at the grammar. This is the structure of a function call:

```
"(" expr { expr } ")"
```

In general case it means the first element of a list is an arbitrary expression that should evaluate to a function object at runtime. A function object which takes the specified number of arguments obtained after evaluation of the following list expressions.

When we process the function call by **CT_Eval**, we can examine the first element. In most cases, it will just be a reference to function (its name or alias) which means we can mark the call as **direct**. Direct calls have the advantage that we can check whether the number of arguments matches the function signature, at compilation time. However, it is also possible the first expression is another function call, conditional expression, etc. In that case, the final function object is unknown at compilation time. The value of the expression could depend on runtime variables or we might not get any function object at all. This kind of call is **indirect**. We must first obtain a return value from the evaluated expression and then perform runtime checks to make sure we can safely execute the call using information from the function object.

While processing definitions, we also create AST nodes for user defined functions (**ScmFunc**). Each node contains information about the expected

number of arguments, then a list of argument names, list of bodies and two important flags, *has_closure* and *passing_closure*. The first flag simply indicates whether this function has captured any objects from surrounding environments. The second flag is useful when we have nested closures and need to pass closure context from an outer function through a middle function to particular inner function. This mechanism is described closely in 2.1.4.

For a complete closure support, it is sometimes necessary that local objects defined inside functions or arguments passed to them outlive the function execution itself. That can be solved by placing captured objects on the heap instead of the function's stack frame. For this purpose, we have the *heap_local_idx* map in the **ScmFunc** object. When symbolic references are resolved and we encounter a captured object, we assign index to it using the map. That index will be used to locate the symbol within a heap array (see 2.1.4).

ScmFunc nodes are also used for external functions (from runtime or other library). They don't contain bodies or argument names and serve as declarations provided by the top-level environment.

Additionally, we can subclass **ScmFunc** and create nodes for specific functions that are either explicitly called or inlined when possible, using a specialized code. That way we can get more efficient code in some cases.

On the AST, we could also perform basic optimization passes such as constant folding or dead code elimination but since we are using the LLVM framework as backend, it would be redundant. LLVM has all these optimizations (and more) already implemented at the IR level (see 2.1.4).

At this point we have gathered enough information to proceed with code generation.

Table 2.2: Type structures

Length [B]	LLVM type	Label	Note
	`scm_type`		
4	i32	tag	
	`scm_int`		
4	i32	tag	
8	i64	value	
	`scm_float`		
4	i32	tag	
8	double	value	
	`scm_str / scm_sym`		
4	i32	tag	
4	i32	length	number of characters
≥ 1	[1 x i8]	string	zero terminated, [1 x i8] array actually has $length+1$ elements, not 1
	`scm_cons`		
4	i32	tag	
arch. specific	scm_type*	car	both car and cdr point to any type
arch. specific	scm_type*	cdr	after reading the target's tag, we can cast it to the right structure type
	`scm_func`		
4	i32	tag	
4	i32	argc	number of arguments
arch. specific	scm_type* (scm_type*, ...)*	fnptr	ptr to implementation
arch. specific	scm_type* (scm_type**)*	wrfnptr	ptr to a wrapper function receiving array of arguments
arch. specific	scm_type**	ctxptr	ptr to captured variables (or null, if there is no closure)
	`scm_vec`		
4	i32	tag	
4	i32	size	number of elements
≥ 0	[1 x scm_type*]	elems	elements of the vector (actual length of the array varies)

2.1.4 Code generation

In the code generation phase we take full advantage of the LLVM API which allows us to easily transform AST nodes to code and data expressed by the intermediate representation (IR).

We traverse the whole tree recursively and generate code for each node. We also save the generated LLVM Values back to the nodes, so that when we process **ScmRef** nodes we can determine what code they are referring to without performing any additional lookup.

Types

First, we have to define our types. Scheme is dynamically typed which means that any variable can hold any type. Because each type has a different size (and in case of strings and symbols, that size has no constant limit), it would be impractical to store them by value. So, instead, variables will point to the values represented by tagged structures. Each type has its unique tag which we store in the first structure field so we could later use it for runtime type inspection.

LLVM does not name structure members. Instead, it uses numerical indices and instruction **getelementptr** (GEP) for access. Table 2.2 summarizes the layout of type structures.

Note that for strings, vectors and symbols we use static array declared as [1 x *type*] just to express that the contents of the array are stored inside the structure. When defining particular constants of these types, the array is declared according to the actual length that is needed ([*length* x *type*]). On the other hand, values created at runtime are always allocated dynamically and the correct length is stored alongside the array regardless of the declared type.

Constants

Constants are generated as globals with internal linkage (equivalent of **static** in C). We tell LLVM to create a constant expression of a given type (one of our structures) and initialize a new global constant with it. This way, we can create constant numbers, strings, symbols and even quoted lists – we just recursively initialize `scm_cons` structures with constant pointers to globals representing the current list element and the rest of the list (another `scm_cons` structure). We need to be able to cast these constant pointers to `scm_type*`. That can be done in LLVM with the **ConstantExpr::getCast** method.

Entry point

Our compiler should support creation of standalone executables as well as libraries. It will be configurable, so that we can choose one of the desired

compilation modes.

When building the standalone executable, there has to be a **main** function (as per convention) which is going to be called by the operating system when it loads our application. It has the signature we know from C:

```
int main(int argc, char **argv);
```

We can write an equivalent declaration in LLVM IR like so:

```
declare i32 @main(i32 %argc, i8** %argv)
```

All the code generated from top-level Scheme expressions will then be placed into the **main** function. The same holds for top-level Scheme definitions (symbol – expression binding) because the bound expression does not have to be always constant. The defined symbol is translated to a global variable and we put all the code needed for its initialization into **main**.

In case of libraries, there is no entry point typically. We just export a set of functions for others to use. However, as we have just explained how Scheme definitions work, it is clear we need to support initialization code even in libraries. Where to put it if not to **main**?

The answer is we create a custom **init** function and then a special LLVM global array **llvm.global_ctors** which contains pointers to functions we want to execute at library load [8]. Ultimately, LLVM will use this information to generate platform specific code/data for invoking these functions (e.g. .ctors section in Linux ELF executable [17]).

There is one special case when we don't want a standalone executable but also don't need library initialization – when we dynamically compile a new expression using **eval**. In this case, it is desirable to emit only one "anonymous" function with a unique ID which will be called by the JIT compiler right after it is generated.

If expression

Now we introduce conditional code flow using the **if** expression. In Scheme, it is represented by a list starting with the **if** keyword, followed by *then* expression (executed if the condition is satisfied) and *else* expression (executed otherwise).

```
(if condition then else)
```

Unlike if/else block in C, this **if** has a return value. It returns the result of the one expression that gets chosen depending on the condition.

LLVM IR uses basic blocks to model instruction flow. Example function containing **if** expression (figure 2.3) starts with the *entry* basic block where the code for condition evaluation is placed.

Result of the condition expression is an arbitrary object. We examine whether that object is **#f** (*false* – tag ID equal to 0) and then generate

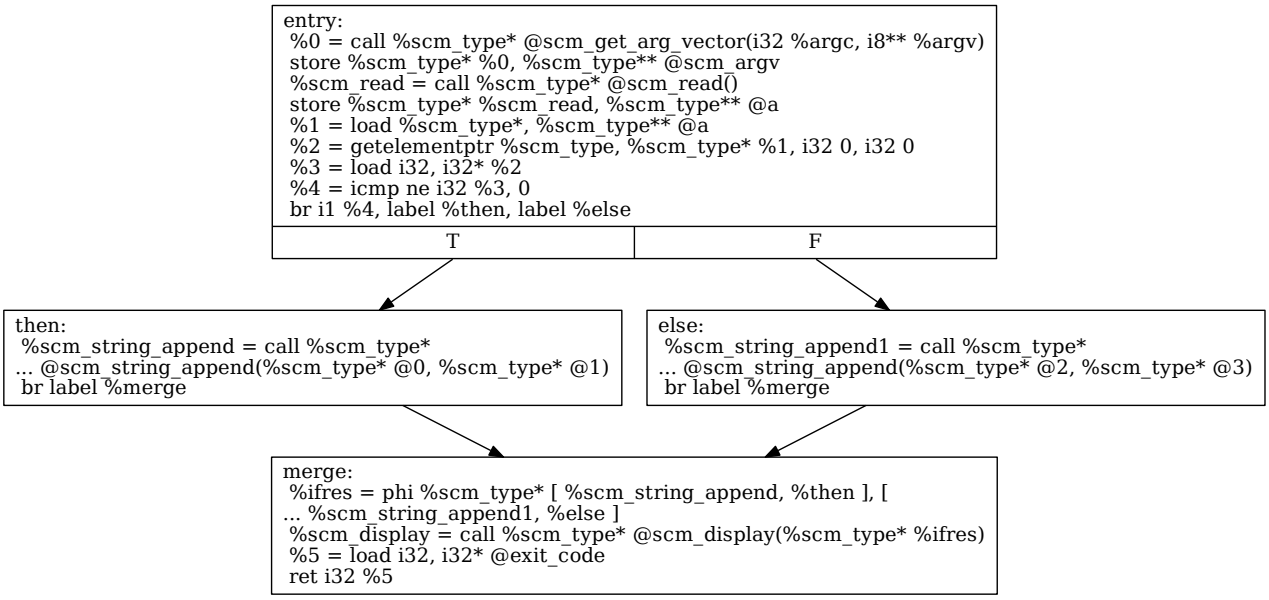


Figure 2.3: If expression in LLVM IR

conditional branch instruction **br** which takes a boolean (*i1*) value and jumps to one of the specified basic blocks – *then* or *else*.

Note that each basic block in LLVM must end with either a branch instruction or a return instruction, even if the branch is unconditional.[8] That can be seen in our example. Both *then* and *else* blocks branch to the *merge* block. In the *merge* block, we have to "choose" the correct result of the **if** expression. Because LLVM uses SSA and each instruction automatically creates a new variable, there are two versions of the value we should return, one for each branch. To obtain the final value, LLVM uses the **phi** pseudo-instruction which is a kind of conditional assignment.

```
%x0 = phi %type [ %x1, %branch_x1 ], [ %x2, %branch_x2 ]
```

In essence, it says "take the value *x1* if we have come from *branch_x1* or the value *x2* if we have come from *branch_x2* and assign it to *x0*". Both values have to be of the same type (we will cast pointers to `scm_type*` when needed).

Logical operators [18]

1. `(and` *expr* ... `)`

   - If no *expr*s are provided, then result is **#t**.

- If a single *expr* is provided, then it is in tail position, so the results of the **and** expression are the results of the *expr*.
- Otherwise, the first *expr* is evaluated. If it produces **#f**, the result of the **and** expression is **#f**. Otherwise, the result is the same as an **and** expression with the remaining *expr*s in tail position with respect to the original **and** form.

2. (or *expr* ...)

   - If no *expr*s are provided, then result is **#f**.
   - If a single *expr* is provided, then it is in tail position, so the results of the **or** expression are the results of the *expr*.
   - Otherwise, the first *expr* is evaluated. If it produces a value other than **#f**, that result is the result of the **or** expression. Otherwise, the result is the same as an **or** expression with the remaining *expr*s in tail position with respect to the original **or** form.

Example of **and** expression transformed into IR is shown in figure 2.4. As we can see, it corresponds to nested if expressions. We recursively repeat these steps starting from the first expression:

1. Evaluate the expression
2. Compare the result with **#f** (0) using the equality operator (**icmp eq**)
3. If the condition is satisfied (expression evaluates to **#f**), return **#f**
4. If this is the last expression, return its value.
5. Otherwise proceed to the next expression and repeat from step 1.

The IR code of **or** expression is analogous. We just use non-equality operator (**icmp ne**) instead of equality operator and return value of the first expression which is not **#f** (if there is such an expression) or **#f**.

Functions and closures

We have seen there are two types of functions in Scheme: anonymous lambda functions and named functions. After the compile-time evaluation we have reduced these types to the second case (lambdas are given a name and a separate definition).

Beside this criterion, we can still classify functions based on whether they have captured variables (closures) or not. This property reflects in the generated code at multiple places.

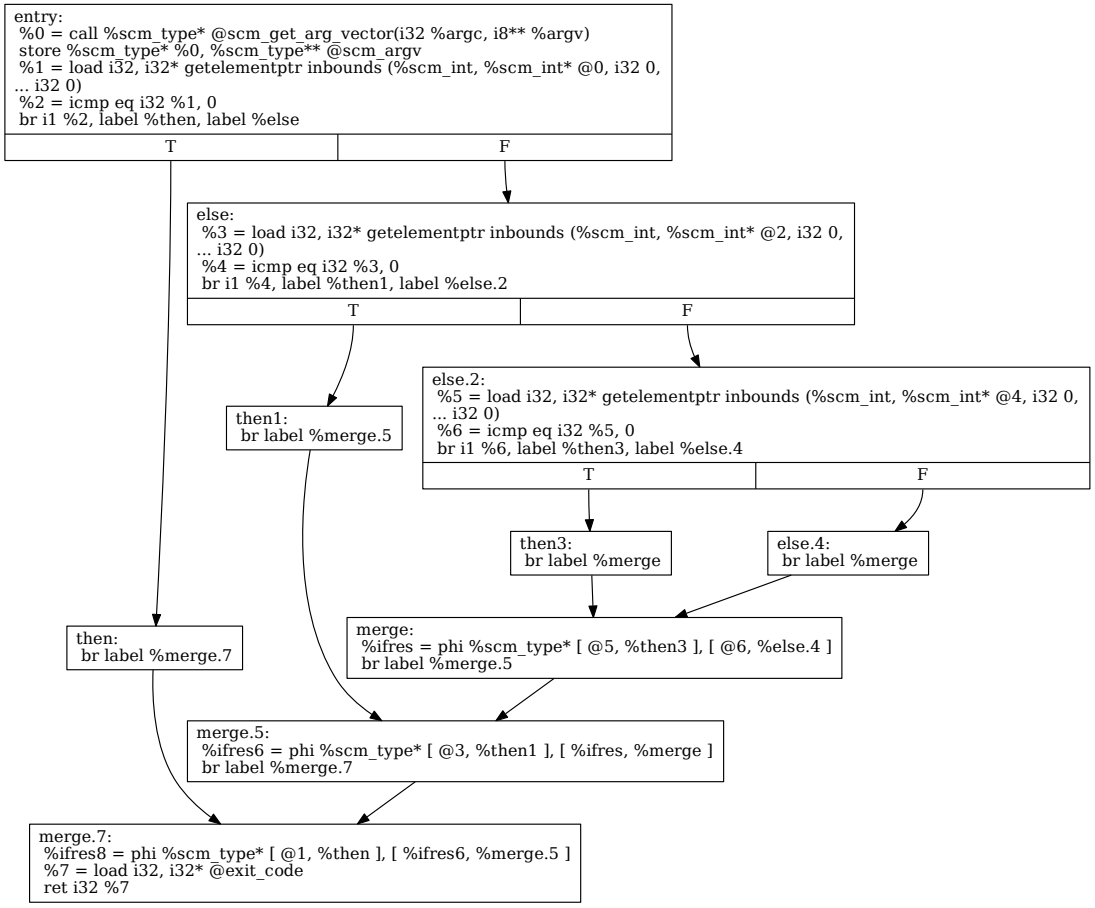


Figure 2.4: *And* expression in LLVM IR

First, when we declare a function, we construct its type (signature) based on the arguments it should take. They always have the same generic type (`scm_type*`) but each function may take different number of arguments: either fixed or variable. LLVM supports the *varargs* flag used to specify this function type property. Additionally, if the function has a closure (that information is present in the **ScmFunc** AST node), we add an extra hidden argument used to pass a context pointer to the function. Via this context pointer, we are able to access all the captured variables.

After declaring the function using the type we have just generated, we proceed with code generation of the function body. However, before actually executing the expressions located inside the body, each function with variables (or arguments) that will be captured (by another function with closure) has

to allocate heap storage for those variables (because captured variables may outlive the defining function execution time). Captured variables defined locally will be placed onto the heap directly, function arguments will be copied to it from their original location (we don't have to know whether it is stack or registers, LLVM abstracts that away from us).

The heap allocation itself is handled by a call to runtime. We just pass the number of variables we wish to store plus 1 (explained bellow) to the corresponding function which then gives us a pointer to the allocated space. It is this very pointer that we later store in function objects as context and pass it as the last hidden argument when calling those objects.

Layout of the heap storage is shown on the following example:

```
(define (foo x_1 x_2)
  (define (bar y_1 y_2)
    (define (baz z_1 z_2)
      (list (+ x_1 y_1 z_1) (+ x_2 y_2 z_2)))
    baz)
  bar)

(((foo 1 0) 2 1) 3 2)
```

When *foo* is called (with arguments 1, 0), heap storage is allocated for *x_1* and *x_2*, their values are copied there and then the function returns a new function object with a pointer to *bar* implementation and a context pointer to *foo*'s heap storage. Now we have a function object with closure which is immediately called with arguments 2, 1 and its own context pointer as the last hidden argument. The *bar* function must allocate heap storage for its arguments but also for the context pointer it received (because *baz* will access variables not only from *bar* but also from *foo*). That is why we always reserve the first element of heap storage array (at index 0) to the parent context pointer.

The memory layout is shown in figure 2.5. Now, the *bar* function returns a function object pointing to the implementation of *baz* and to the new heap storage. When it is called with arguments 3, 2 and the corresponding context pointer, *baz* can access all the values of *x_*, y_*, z_** because we know from the semantic analysis, where they are located. Local variables *z_1, z_2* can be found in the current function's frame, captured variables *y_1, y_2* are one level up through the context pointer (received as last argument), so a pseudocode for accessing them would look like this: `ctxptr[1]`, `ctxptr[2]`. Finally, variables *x_1, x_2* can be reached via the parent context pointer we have previously placed into the *bar*'s heap storage: `ctxptr[0][1]`, `ctxptr[0][2]`. This mechanism will of course work for any number of nested closures.

As we can see, unlike function frames the lifetime of heap storage arrays is not limited. The garbage collector will eventually free the allocated resources but that won't happen before all the associated function objects are also freed.

All the function objects contain not one, but two function pointers. The

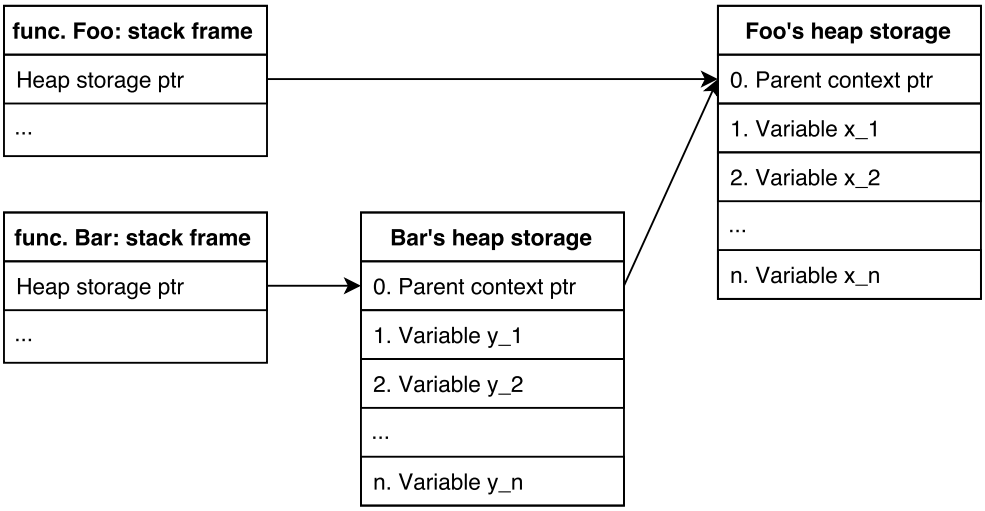


Figure 2.5: Heap storage layout

first one points to the implementation which can be called directly when we know the number of arguments at compilation time. To properly support the Scheme *apply* function, we add another pointer to a special wrapper function that takes only one argument – pointer to an array of the actual arguments we need to pass. The only purpose of the wrapper is to take the first n elements of the argument array (where n is the number of arguments) and pass them to the implementation. If n is not constant (variadic function), we read the array until we reach a *null* pointer. That *null* pointer will always be added at the call site because at runtime we don't have the information whether a target function is variadic or not.

To avoid further indirections, each function has its own wrapper. They are quite short, so we can afford the cost of additional space. We prefer speed. That is why we provide two ways of calling each function instead of just using the universal calling convention (with a pointer to argument array) alone.

So, we generate all the necessary code to support closures, then the function wrapper, and at last we can generate the function body by recursively calling the code generator on all the expressions found in the body list. We save the return value of the last expression and generate **ret** instruction to actually return the value (when the function is called). The code generator returns the function itself (as LLVM **Function** object).

Definitions

When we encounter a definition (**ScmDefineVarSyntax**) in the AST, it means we have assigned a symbolic name to an expression (or a function), but since all the symbolic references are already resolved in the semantic ana-

lysis, is there any code that needs to be generated?

The answer is yes, because definitions also affect scope of the defined expressions. When we write a top-level definition, we expect the bound expression to be available globally. In case of functions, that is already the case but let's say we want to bind a result of some calculation to a specific name. That means the calculation will be performed in the body of an entry point function (*main* or library *init*) and the result has to be stored in a global variable.

So, for each global expression definition, we create a global variable, generate its initializer (evaluation of the expression) and set up the corresponding AST node, so that it refers to the global variable location instead of the original expression value because that would only be local to the function where it was evaluated.

Local expression definitions usually don't require any additional code. We already have the expression value and it is local to the function where it is evaluated. There is one exception: When the value is captured in a closure, we need to copy it to the function's heap storage. In the semantic analysis we have obtained all the information we need to do so: the heap array index assigned to the value and the current function pointer where the heap storage code is already generated, so we can refer to the array pointer.

The same applies to definitions inside the *let* expression. They are always local but of course the defined values can be captured in closures sometimes.

References

The simplest case of reference is a reference to local variable (located in the stack frame of corresponding function). We have the **ScmRef** node pointing to AST node of the referenced value whose LLVM Value is already generated, so we just return that Value, because we are in the same scope.

When dealing with a reference to heap variable (captured by a closure), we instead generate code for indirect access through the context pointer. We just need access to AST node of the function where our referenced value is located – that is where we have stored the heap array indices (*heap_local_idx* map). Context pointer, on the other hand, is associated with the function where we have found the reference itself. The resulting sequence of array indices is formed by $k - 1$ zeros and the last index found in *heap_local_idx* of the referenced value, where k is the relative position as defined in section 2.1.3.

Different code is needed for function references. A function object representation needs to be generated. That means a structure containing the right type ID, the expected number of arguments, two function pointers and a context pointer. If there is no context pointer (ordinary function), the structure can be defined as a global constant. However, if the function has a closure, the context pointer always has to refer to the current function's heap storage (which can be different with every function call) and because the storage is dy-

namically allocated, the whole function object needs to be created at runtime. So, instead of generating a reference to global constant, we generate a call to runtime function *alloc_func* and pass the correct arguments to it (all constant at compilation time except the context pointer).

The last special case is a reference to global variable. LLVM Values of globals are actually pointers to program's data section, so when we define a global variable, the original Value is replaced by a pointer to Value. That means we have to generate **load** instruction now.

Function calls

Code for direct calls is simple. First, we take the given symbolic reference to function (**ScmRef**) but we don't generate the whole function structure – only the raw function pointer (contained in the referenced object) is needed. Then we generate code for evaluation of arguments. The LLVM Values corresponding to the resulting argument values will be passed as operands to **call** instruction along with the function pointer.

In case the called function has a closure, we also pass the current function's heap storage pointer to the call (as a last argument). We know it must be the current function's heap storage. If it weren't, it would mean we got the function reference from some other context but that leads to an indirect call.

Additionally, if the called function is variadic, a *null* pointer is passed as the last argument.

Indirect calls begin with evaluation of the function reference. This time we are not sure it actually refers to a function, so we have to generate code that returns the whole Scheme object. The object is always a structure with a tag, so the next code inspects its type:

First it compares the tag with function type ID. If there is a different tag, we call a runtime function that will signal an error. Otherwise we can cast the structure to function object and look at its expected number of arguments. That will be compared against the given number (constant at the call site). In case the numbers do not match, we call another runtime error function. Only now we can safely invoke the indirect call by extracting function and context pointers from the obtained object. Arguments are processed the same way as before (with the direct call) and the context pointer is also passed as the last argument. We can pass it even if the target function doesn't have a closure. In that case the context is *null*. Because we use C calling convention where the caller is responsible for cleaning up arguments from the stack, all extra arguments will be just ignored.[8]

Optimizations

LLVM supports a wide range of IR optimizations [19] such as:

- **Basic Alias Analysis**

  A basic alias analysis pass that implements identities (two different globals cannot alias, etc), but does no stateful analysis.

- **Combine redundant instructions (peephole optimiz.)**

  Combine instructions to form fewer, simple instructions. This pass does not modify the CFG. This pass is where algebraic simplification happens.

- **Reassociate expressions**

  This pass reassociates commutative expressions in an order that is designed to promote better constant propagation, etc.

- **Global Value Numbering (common subexpression elimination)**

  This pass performs global value numbering to eliminate fully and partially redundant instructions. It also performs redundant load elimination.

- **Simple constant propagation**

  This pass implements constant propagation and merging. It looks for instructions involving only constant operands and replaces them with a constant value instead of an instruction.

- **Interprocedural constant propagation**

  This pass implements an extremely simple interprocedural constant propagation pass.

- **Merge Duplicate Global Constants**

  Merges duplicate global constants together into a single constant that is shared.

- **Simplify the CFG (control flow graph)**

  Performs dead code elimination and basic block merging.

- **Tail Call Elimination**

  This file transforms calls of the current function (self recursion) followed by a return instruction with a branch to the entry of the function, creating a loop.

- **Promote Memory to Register**

  This file promotes memory references to be register references. It promotes alloca instructions which only have loads and stores as uses.

- **Dead Instruction Elimination**

  Dead instruction elimination performs a single pass over the function, removing instructions that are obviously dead.

- **Dead Code Elimination**

  Dead code elimination is similar to dead instruction elimination, but it rechecks instructions that were used by removed instructions to see if they are newly dead.

- **Function Integration/Inlining**

  Bottom-up inlining of functions into callees.

- **Unroll loops**

  This pass implements a simple loop unroller.

We could use many of these optimization passes. However, the question is how successful they would be given that our code is composed mostly of sequences of calls and conditional branches. For example, the constant propagation will work on integral types or floating point types only. Not on our general Scheme number type which is actually a structure. That means we need to perform some sort of type inference first in order to generate more specialized code and subsequently leverage the power of LLVM optimizations designed for statically typed code.

There is one important optimization for Scheme that works regardless of types – the **tail call elimination**. Because we usually use recursion instead of iteration in Scheme, we want it to be as efficient as possible. LLVM will help us with that – it transforms all the self-recursive tail calls to jumps, effectively converting recursion to direct iteration.

2.1.5 Modules

When we compile a program, there should be a way for the compiler to automatically resolve external function (or global variable) references. The core Scheme functions which will be implemented in C++ will also be enumerated in a header file (`runtime.h`) and we can generate a list of their names for the compiler (using a pre-build script) or hardcode it in its source.

However, parts of the runtime library could certainly be implemented in Scheme and we can of course compile additional libraries. Because there are no header files in Scheme, we need to devise a way of storing the equivalent metadata in the compiled binaries.

Our situation is simplified as Scheme is a dynamically typed language. That means a function signature does not specify types of its arguments or the return value. We only need to store function names, the number of arguments

of each function and the names of global variables, to be able to reconstruct all the declarations in LLVM IR.

To implement this, we can just design a custom format and store the information as an array of structures into a special global variable. When we later compile a program that depends on library functions, we load the library, read contents of this metadata array and compose a list of available functions and globals from it.

The runtime library will be automatically loaded during every compilation. If our Scheme program wishes to use functions from an additional library, it has to inform the compiler by using the `(require "lib_path")` statement. The library path is relative to the current working directory or the directory where our executable is stored (both locations will be searched).

Although this solves the problem of missing headers, linking still has to be done manually as our compiler produces object files only. The compiler could of course call the linker for us, but this approach is more flexible until we develop an extended interface which is able to pass all the necessary options. Our implementation will only pass basic options to the *llc* compiler.

On the other hand, when we compile code dynamically by calling **eval** and we evaluate a **require** statement, the target library is actually linked to our running application, so that we could use the library functions next time we invoke the JIT compiler.

2.2 Runtime System

To actually run any of the programs built by our compiler we will of course need a runtime library that will provide a set of core functions (implemented in C/C++) and an interface to the compiler (for **eval** support). It will also manage memory and handle runtime errors.

2.2.1 Core functions

Among the core functions written in C/C++ are:

- **Arithmetic operators:** *+*, -, ***, */*, =, >

  All these functions have to inspect the given objects, determine their type and operate on the raw values inside them (float or integer value inside a structure). They can receive both integer and floating point operands mixed. When at least one operand is float, the result is also float. Otherwise it is integer.

- **Printing functions:** *display*, *print*

  Functions that print out textual representation of various objects. The *print* function represents objects verbatim – in the way they were written

in the source code. For example a string `"two\nlines"` is printed exactly as it is – with the quotes and the newline escape sequence. When we call *display* on it, we get

```
two
lines
```

- **Function for obtaining command line arguments**

  We have to take the command line arguments given to us by the operating system and wrap them in Scheme compatible objects.

- **Functions for list creation and manipulation:** *cons*, *list*, car, *cdr*

  These functions operate on the internal linked-list representation.

- **Functions that test the type of an object:** *null?*, *eof-object?*, ...

- **Comparison functions:** *eq?*, *equal?*

  The *eq?* function compares pointers. If they are the same, it means we have two references to the same object – in that case *eq?* returns `#t`. To compare objects by value (recursively), we use *equal?*. We can say that two objects are *equal* if they have the same type and value. In case of lists, the same must hold for each element.

- **The *apply* function**

  When we call *apply* on a function and a list of arguments, it needs to copy elements of that list into an array and pass the array to a function wrapper of the target function we want to call.

- **The *read* function**

  This function reads user input (from **stdin**) and parses it as Scheme data (atoms, lists). For that purpose it can reuse the **FileReader** tokenizer described in section 2.1.1 which reads from an input stream.

- **File operations:** *open-input-file*, *close-input-port*, *read-line*, ...

  Apart from accessing standard input and output, we would like to be able to work with files. For that purpose we define the `scm_file` type which is just another tagged structure, this time wrapping the native file handle. Then we implement a standard set of functions capable of opening files, closing them, reading their content and writing new content. All these functions will use `scm_file` and other Scheme types on the outside, while working with native file operations inside.

- **String operations:** *string=?*, *string-append*, *string-replace*, ...

  Because string types are atomic in Scheme, we can manipulate with their representation be means of native functions only. Common functions such as string comparison, string appending, string splitting, and replacing of substrings can be implemented.

2.2.2 Memory management

The runtime system is responsible for object allocation as well as garbage collection. We have chosen the Boehm-Demers-Weiser conservative garbage collector for this task. Our objects are in fact structures representing the Scheme values.

There will be a dedicated allocation function for each Scheme type (and also the closure heap storage), serving as a constructor of sorts. Scheme integer is constructed using a native `int64_t`, Scheme float constructor will take a `double`, string and symbol constructor an array of `char`s, and so on. In each constructor, we determine the exact amount of space needed for the structure (constant for all types except strings, symbols and vectors – they have a variable number of characters/elements) and then we actually allocate the memory using the `GC_MALLOC` function.

Each memory block allocated this way will be freed automatically by the collector, when it discovers there are no references left to that particular block.

The garbage collector library also supports C++ objects with non-trivial destructors. That is convenient because we have the namespace (`scm_nspace`) type which encapsulates compiler environment (**ScmEnv**). The namespace is a first class Scheme object, therefore garbage collected, whereas the memory management of **ScmEnv** is deterministic, at least when it is run from the context of static compiler that will be written in plain C++ – without garbage collection.

So, when we create an instance of C++ class in a garbage collected structure, we need to ensure all the class members' destructors are called when the collection occurs. To implement this, we just inherit from the `gc_cleanup` class provided by the GC library.

2.2.3 Just-in-time compilation

To be able to dynamically add new code to a running application and to allow that code to interact with other parts of the program, we maintain an environment (**ScmEnv**) across multiple compilations. Scheme provides a function *make-base-namespace* which creates such an environment encapsulated in the namespace object and populates it with runtime library function references.

There is always one namespace that is marked as current. We can use function *current-namespace* to get its current value or set a new one.

The current namespace is global, so we can access it from everywhere. Certainly, we could just have a global binding like this:

```
(define ns (make-base-namespace))
```

That namespace can be passed to any *eval* function in our program. More accurately, we can evaluate new code in the environment provided by that namespace. However, what if we want even the evaluated code to see this definition? Maybe we would like to call *eval* inside *eval* using the same environment. Evidently, the *ns* symbol does not exist within the environment (it is only a name we have given it outside) and evaluating the exact same definition would only create a new independent environment. And yet, we can call *current-namespace* in the evaluated code which will indeed give us access to the original environment given that we have previously set it as the current.

This way, we can implement a REPL (read-eval-print loop) in Scheme using *eval* on the user input which itself may contain another *eval*. All the code will share the same environment nevertheless.

The *eval* function itself uses **ListReader** tokenizer (section 2.1.1) on the given expression (first argument), initializes parser on top of that, parses the expression, performs compile time evaluation in the given environment (second argument), sets the correct build type and expression name, runs the LLVM IR code generator and at last the LLVM JIT. The JIT is set up to perform various optimizations and to generate native code (in memory).

After the native code is generated, new symbols become available in our program – namely the symbol corresponding to our expression. We look up its address and then simply call that address after casting it to the right function type.

We also perform an environment cleanup before the next *eval* call – all the new symbol bindings left in the environment have to be marked as external and their pointers to LLVM Values reset.

2.2.4 Error handling

Because Scheme is dynamically typed, not all the necessary type checks can be performed at compilation time. We need to handle runtime errors such as invalid types of arguments passed to a function, wrong number of arguments given to an indirectly called function, or even an invalid call of object which is not a function.

Additionally, when we call the compiler via *eval*, all of the compiler's errors need to be propagated as well.

How should we react to these errors? The most straightforward approach is to terminate the program after displaying an appropriate error message. We assume these runtime errors should not occur if the program is well-formed.

Therefore, when they do occur, it means the programmer should fix his implementation.

Of course, there are times when we would like to handle an error in a different way, possibly continuing execution of the application afterwards. Scheme has a support for exceptions for that purpose. However, its exception system is built on top of continuations [10] and it is more flexible than the exception support provided by LLVM.

In LLVM we can model the traditional **try** and **catch** blocks as well as the **throw** operation which unwinds the stack until it hits the frame where we have placed a **try** block. From that frame, execution control is passed to the corresponding **catch**.[20] On the other hand, Scheme relies on the continuation passing style and the exception handler is just another function which we transfer control to when the exception is raised. That means we don't have to always unwind stack before the handler is invoked and therefore the exceptions are continuable [21] (in the sense that execution can continue anywhere since we don't lose the stack frames between "try" and "throw").

The continuation passing style is not directly supported in LLVM – every function ends with a return statement, so we cannot choose explicitly where the execution will continue. This behavior can be emulated but we still do not have a direct control over function stack frames.

To keep the initial implementation simple, we choose not to implement exceptions. In case of runtime errors, program termination is sufficient most of the times. Of course, the exception system is also intended for user defined errors, so a complete implementation of Scheme should certainly support it. We are implementing just a subset of the language, however.

Chapter 3

Implementation

3.1 Choice of language and build tools

We have chosen C++ as the implementation language of our compiler and runtime library. The main reason for this decision is that LLVM framework itself is written in C++ (C++11, to be exact) and while there are various unofficial bindings of LLVM API to other languages (and also official bindings to C, LLVM-C[22]), it would be less practical to choose any of them because we want the compiler to be directly accessible from runtime.

Let's say we would write the compiler in a higher level language like Ruby, for example, or we would even like to implement a self-hosting compiler. That can certainly be done (except there are no LLVM bindings to our Scheme yet), but at the same time the runtime library has to be a native application with core functions that the target language relies on and which cannot be written in Scheme itself. The natural choice for a runtime library is C or C++ (for efficiency), and if we expect the runtime to be frequently calling the compiler, we should make their interoperability as simple as possible: We write both the runtime and the compiler in the same language. We choose C++ because the LLVM-C interface is not complete.[23]

Because LLVM is written in modern C++ and relies on the latest language and standard library features, we have an opportunity to also use these features in our code. That means we take full advantage of STL containers, custom template functions and classes (including variadic templates), lambda functions, smart pointers, template metaprogramming, etc.

Our project is compiled using the clang compiler (also built on top of LLVM) and Makefiles are generated by CMake[3] which is integrated into the JetBrains CLion IDE[4].

[3] `https://cmake.org/`

[4] `https://www.jetbrains.com/clion/`

3.2 Project structure

```
llscheme ..... project config files, README, LICENCE, etc.
├── bin ..... executables
│   ├── Debug ..... debug version
│   └── Release ..... release version
├── build ..... out-of-tree build directory
├── include ..... compiler header files
│   ├── elfio ..... library for ELF parsing
│   ├── linenoise ..... line editing library
│   ├── optionparser ..... command line argument parser lib.
│   └── runtime ..... runtime header files
├── src ..... compiler source files
│   ├── runtime ..... runtime source files
│   └── test ..... unit tests
└── test ..... post-build test scripts
    └── lls_programs ..... Scheme example programs
        └── lib ..... Scheme example libraries
```

Figure 3.1: Project directory structure (see the enclosed CD)

3.2.1 Used libraries

Apart from LLVM framework and GC library, we use these additional libraries:

- **ELFIO**[5]

  A C++ header-only library for reading and generating ELF files.
  We can load the Scheme metadata using this library (see 3.3.3).

- **Linenoise**[6]

  An alternative to the GNU Readline library written in C, providing support for user input with line editing capabilities (terminal control sequences, history, autocompletion, ...). We use it to implement the Scheme REPL.

- **The Lean Mean C++ Option Parser**[7]

  A freestanding header-only C++ library for command line argument parsing. It supports the short and long option formats of *getopt*, *getopt_long* and *getopt_long_only* but has a more convenient interface. It

[5] `http://elfio.sourceforge.net/`
[6] `https://github.com/antirez/linenoise`
[7] `http://optionparser.sourceforge.net/`

allows you to access arguments directly by their name, rather than looping through all of them sequentially.

- **Boost.Filesystem**[8]

  A C++ library for accessing and manipulating files and directories. It is used in our compiler to perform a library lookup necessary before loading the library and reading its metadata.

3.2.2 Source and header files

```
llscheme
└── src
    ├── ast.cpp ..... AST implementation (ScmObj hierarchy)
    ├── ast_visitor.cpp ..... AST visitor base class
    ├── codegen.cpp ..... LLVM IR code generator (AST visitor)
    ├── driver.cpp ..... standalone compiler interface (schemec)
    ├── environment.cpp ..... ScmEnv class implementation
    ├── fs_helpers.cpp ..... library lookup implementation
    ├── libmetainfo.cpp ..... library metadata loading and storing
    ├── lib_reader.cpp ..... metadata extraction from ELF
    ├── parser.cpp ..... Scheme parser implementation
    ├── reader.cpp ..... Scheme tokenizer implementation
    ├── runtime ..... runtime library (llscmrt)
    │   ├── error.cpp ..... runtime error handling functions
    │   ├── memory.cpp ..... memory allocation functions
    │   ├── readlinestream.cpp ..... custom istream for user input
    │   ├── runtime.cpp ..... core language functions
    │   └── scmlib.scm ..... additional functions written in Scheme
```

Figure 3.2: Source files

Each source file has its corresponding header under `llscheme/include` containing class and function declarations plus implementation of template functions. However, there are also header files without source files (fig. 3.3).

The **any_ptr** class is used as a return type of AST visitor methods, so we could implement multiple visitor classes that work with various types. We use this wrapper for convenience only. It provides automatic conversion from an arbitrary pointer type to **any_ptr** (via template constructor) and the **any_ptr_cast** operator for converting the pointer back to its original type.

The **internal.hpp** header contains implementation of variadic functions which are exposed to Scheme through two types of wrappers: the first supports C variadic calling convention, the second takes a pointer to argument array.

[8] `http://www.boost.org/doc/libs/1_60_0/libs/filesystem/doc/index.htm`

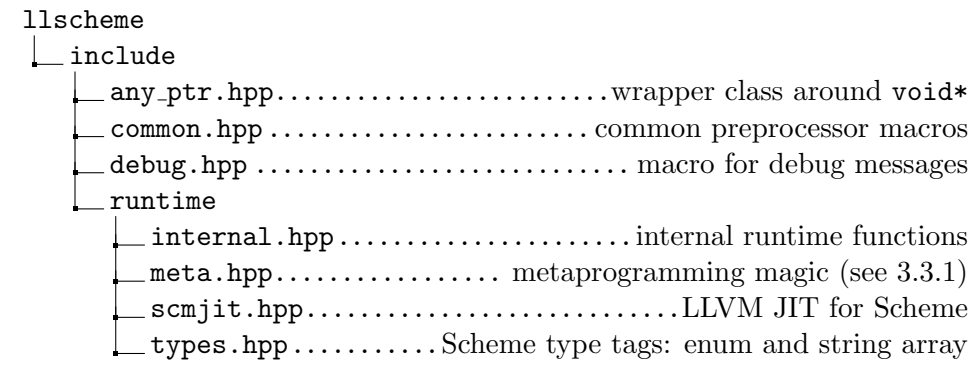


Figure 3.3: Header files

To abstract away this difference, the wrappers pass two lambda functions into the internal implementation: *first_arg* and *next_arg*. As their names suggest, *first_arg* returns the first argument, whereas calling *next_arg* repeatedly yields the rest of them. Note that there are no indirect calls after the code is compiled with optimizations. The internal functions are declared as inline, so there will actually be two versions of each function generated (inlined to both wrappers).

The **ScmJIT** class defined in **scmjit.hpp** is a modified version of LLVM JIT used in the Kaleidoscope tutorial [7]. It provides methods for adding and removing individual LLVM Modules and a method for symbol lookup. When a module is added to the JIT, it is compiled in memory and its symbols become available.

The Scheme type tags are defined in **types.hpp**. We use X Macros to automatically generate a **Tag** enum and an array of strings (used for error messages) from one list of type tag names:

```
#define TYPES_DEF(T) T(FALSE), T(TRUE), T(NIL), T(INT), T(FLOAT), \
T(STR), T(SYM), T(CONS), T(FUNC), T(VEC), T(NSPACE), T(EOF), T(FILE)

#define T_STR(name) "S_" #name
#define T_ENUM(name) S_##name

enum Tag {
    TYPES_DEF(T_ENUM)
};

static const char * TagName[] = {
    TYPES_DEF(T_STR)
};
```

3.3 Implementation details

3.3.1 Automatic function wrapper generation

As discussed in section 2.1.4, each function has a wrapper that allows us to call it with a variable number of arguments where the exact number is not known at compilation time (so we can *apply* a Scheme list of arguments to function). When we implement a function in Scheme, our compiler's code generator emits this wrapper for us. On the other hand, when a Scheme function is written in C++, its wrapper should be defined in the C++ source as well.

To avoid writing all the wrappers manually, we use a generic template function that can be instantiated with the needed number of arguments and the corresponding inner function as many times as needed. This is possible thanks to variadic templates (since C++11).

```
template<typename>
struct Arguments;

template<int... idx>
struct Arguments<RangeElems<idx...>> {
  template<typename T, T * func>
  inline static scm_type_t * argl_wrapper(scm_type_t ** arg_list) {
    return func(arg_list[idx]...);
  };
};
```

So, how does it actually work? Here we can see a template structure **Arguments** with its partial specialization taking a parameter pack of `int`s. Inside the structure, there is a static inline method which takes the argument array **arg_list** and calls the *func* function (obtained as a template parameter) with arguments taken from the array. If the **idx** parameter pack contains integers from 0 to n, `arg_list[idx]...` translates to `arg_list[0], arg_list[1], ..., arg_list[n]`.

Because we don't want to enumerate all the indices from zero up to the function's arity minus one every time we instantiate a version of the wrapper, we use a compile-time range:

```
template<int... Elems>
struct RangeElems{
    typedef RangeElems<Elems...> type;
};
// (1) recursive template
template<int Left, int Next, int... Elems>
struct GetRange: GetRange<Left-1, Next+1, Elems..., Next> {};

// (2) partial specialization to end the recursion
template<int Next, int... Elems>
struct GetRange<0, Next, Elems...>: RangeElems<Elems...> {};

template<int End>
struct Range: GetRange<End, 0> {};
```

Using template metaprogramming, we recursively define a type alias of **RangeElems** containing all the indices as template parameter pack. We can demonstrate on an example, how the C++ compiler deduces the given type:

1. Let's say we start with **Range**<3> type.
2. That type inherits from **GetRange**<3, 0>,
   matching template *(1)*: *Left* = 3, *Next* = 0, *Elems* is empty.
3. **GetRange**<3, 0> inherits from **GetRange**<2, 1, 0>,
   matching template *(1)*: *Left* = 2, *Next* = 1, *Elems* = 0.
4. **GetRange**<2, 1, 0> inherits from **GetRange**<1, 2, 0, 1>,
   matching template *(1)*: *Left* = 1, *Next* = 2, *Elems* = 0, 1.
5. **GetRange**<1, 2, 0, 1> inherits from **GetRange**<0, 3, 0, 1, 2>
   matching template *(2)*: *Left* = 0, *Next* = 3, *Elems* = 0, 1, 2.
6. **GetRange**<0, 3, 0, 1, 2> inherits from **RangeElems**<0, 1, 2>,
   which defines alias **type** for itself.

That means **Arguments<typename Range<3>::type>** translates to **Arguments<RangeElems<0, 1, 2>>**

To make all this even simpler to use, we provide the following helper code:

```
template<int C> // Template alias
using Argc = Arguments<typename Range<C>::type>;

template<typename R, typename... Args> // Compile-time expression
constexpr int arity(R(Args...)) { // Takes a function
    return sizeof...(Args); // Returns its number of arguments
}

#define SCM_ARGLIST_WRAPPER(f) \
scm_type_t * argl_##f(scm_type_t ** arg_list) { \
  return Argc<arity(f)>::argl_wrapper<decltype(f), &f>(arg_list); \
}
```

The **SCM_ARGLIST_WRAPPER** macro (which takes a name of the function we want to wrap) seemingly creates another wrapper around the template static method, but that method will actually be inlined. We do this to give an appropriate name to the wrapper and also to be able to declare it in the **extern "C"** block. Template functions cannot have C linkage because they rely on C++ symbol name mangling.

3.3.2 Helper functions for code generation

In the LLVM IR code generator, we use various helper functions to avoid code repetition and to reduce the amount of boiler-plate code needed to generate functions, basic blocks and instructions. One of these functions is intended for if/else expression generation and it is used not only for the Scheme **if** but also for the logical expressions (**and**, **or**) as well as object type checking when performing indirect calls. It is defined as such:

```
template<typename F1, typename F2, typename F3>
Value * genIfElse(F1 cond_expr, F2 then_expr, F3 else_expr) {
  // Get value of the condition expression
  Value * cond_val = cond_expr();
  // Get current function
  Function * func = builder.GetInsertBlock()->getParent();

  // Create the necessary basic blocks, attach "then" block to func.
  BasicBlock * then_bb = BasicBlock::Create(context, "then", func);
  BasicBlock * else_bb = BasicBlock::Create(context, "else");
  BasicBlock * merge_bb = BasicBlock::Create(context, "merge");

  // Generate conditional branch
  builder.CreateCondBr(cond_val, then_bb, else_bb);

  // Generate body of "then" block
  builder.SetInsertPoint(then_bb);
  Value * then_val = then_expr();
  builder.CreateBr(merge_bb);
  then_bb = builder.GetInsertBlock();

  // Generate body of "else" block after attaching it to func.
  func->getBasicBlockList().push_back(else_bb);
  builder.SetInsertPoint(else_bb);
  Value * else_val = else_expr();
  builder.CreateBr(merge_bb);
  else_bb = builder.GetInsertBlock();

  // Attach the "merge" block
  func->getBasicBlockList().push_back(merge_bb);

  // Create PHI node
  builder.SetInsertPoint(merge_bb);
  PHINode * phi = builder.CreatePHI(then_val->getType(), 2, "ifres");
  phi->addIncoming(then_val, then_bb);
  phi->addIncoming(else_val, else_bb);

  return phi;
}
```

This is a generic if/else expression taking three lambda functions that will be called in order, generating code for the condition, "then" block and "else" block. They have to be called in the right context so that the **IRBuilder** insertion point is set to the correct basic block.

3.3.3 Extracting the metadata from shared object files

When we build a Scheme library, our compiler generates metadata containing the names and signatures of exported functions (names of global variables can be added in the future). This information is stored in a global byte array named **__llscheme_metainfo__** which consists of a sequence of **FunctionInfo** structures.

```
struct __attribute__ ((__packed__)) FunctionInfo {
    int32_t argc; // Number of function arguments
    char name[1]; // Name of the function (null-terminated)

    void * operator new(std::size_t s, void * p, int32_t nlen);

    // Disable default new
    void * operator new(std::size_t s) = delete;
    void * operator new(std::size_t s, void * p) = delete;

    FunctionInfo(int32_t argc, const char * name);

    static size_t size(size_t nlen);

    size_t size();
};
```

The structures are allocated using an overloaded placement **new** operator which takes sizeof(**FunctionInfo**) implicitly, pointer to the location where the structure should be placed (our metadata array) and the length of our function name. We do this, because each structure should contain the whole function name, therefore its size is not what the compiler infers from sizeof operator – we must add the corresponding string length:

```
void *FunctionInfo::operator new(size_t s, void *p, int32_t nlen) {
    // Allocate enough space for variable length function name
    size_t size = s;
    size += nlen * sizeof(char);
    return ::operator new(size, p);
}
```

At the end of **__llscheme_metainfo__** array, there is always an empty entry with **argc** and **name**[0] set to 0.

In order to refer to library functions from other Scheme programs, the compiler has to load the metainfo array. In case of runtime library (which is partly written in Scheme), this happens automatically. Other libraries are loaded on demand via the **require** statement.

At first, we wanted to access the array by simply linking (dynamically) each library before the compilation. The compiler would use LLVM's equivalent of the Unix *dlopen* function to load all the library symbols including **__llscheme_metainfo__**. Unfortunately, this didn't work for the runtime library because it also contains the compiler. These two programs share portions

of code (statically compiled into both of them) so they cannot be linked together without errors (multiple definitions etc.)

This problem could be solved by moving the compiler to a separate shared library that would be linked to a standalone executable as well as the runtime library. On the other hand, the whole process of linking seems a little redundant when the only thing we need is the contents of one array. That is why we have decided to use a custom loader which only parses the ELF library symbol table and gives us a pointer to the metainfo array.

Instead of parsing ELFs manually, we use the ELFIO library:

```
struct LibReader::Impl {
  ELFIO::elfio reader;
  ELFIO::section * dynsym;
};

bool LibReader::load(const string & libname) {
  if (!impl->reader.load(libname)) {
    return false;
  }

  Elf_Half sec_num = impl->reader.sections.size();
  for (uint32_t i = 0; i < sec_num; ++i) {
    section * sec = impl->reader.sections[i];
    if (sec->get_type() == SHT_DYNSYM) {
      impl->dynsym = sec;
      break;
    }
  }
  return true;
}

void * LibReader::getAddressOfSymbol(const string & symname) {
  const symbol_section_accessor symbols(impl->reader, impl->dynsym);

  for (uint32_t j = 0; j < symbols.get_symbols_num(); ++j) {
    string name; Elf64_Addr value; Elf_Xword size;
    uint8_t bind; uint8_t type;
    Elf_Half section_index; uint8_t other;

    symbols.get_symbol(j, name, value, size, bind,
                       type, section_index, other);

    if (name == symname) { // This is our symbol
      section * data_sec = impl->reader.sections[section_index];
      int64_t offset = value - data_sec->get_address();
      const char * data = data_sec->get_data() + offset;

      return (void*)data;
    }
  }
  return nullptr;
}
```

3.3.4 Garbage collected objects and smart pointers

Another interesting problem is the interoperation between garbage collected objects created by the Scheme runtime (such as the namespace object) and C++ objects used by the compiler (**ScmEnv** inside the namespace). When working with the compiler objects, we often use shared pointers (shared_ptr) which implement automatic reference counting. This approach certainly simplifies memory management but how do we manage an instance of **ScmEnv** (the compiler environment) that was created as part of the Scheme namespace structure?

As we have already mentioned in section 2.2.2, the GC library provides a class **gc_cleanup** which can be used for C++ objects whose destructors need to be called during garbage collection. This is accomplished through inheritance but we don't want all of the instances to be collectable. GC should manage only C++ objects inside Scheme objects. Furthermore, we sometimes need a shared pointer to a garbage collected C++ object. All this can be implemented quite elegantly using templates:

```
// Wrapper class for objects that are
// to be automatically garbage collected.
template<class C>
class GCed: public C, public virtual gc_cleanup {
  static std::set<GCed<C>*> instances;
  static std::shared_ptr<bool> anchor;
public:
  // Constructor passthrough
  template<typename ...Args>
  GCed(Args &&...args): C(std::forward<Args>(args)...), gc_cleanup() {
    instances.insert(this);
  }
  virtual ~GCed() {
    instances.erase(this);
  }

  std::shared_ptr<C> getSharedPtr() {
    // We use the shared_ptr aliasing constructor
    // to get something like "shared_from_this()"
    // for an object managed by the Boehm's GC
    return std::shared_ptr<C>(anchor, this);
  }

  // If the GC doesn't delete all of the instances
  // We can still call this manual cleanup at exit
  static void cleanup() {
    for(auto obj: instances) {
      delete obj;
    }
  }
};
```

Then we can use the **GCed** class like this:

```
struct scm_nspace_t {
  int32_t tag;
  GCed<ScmEnv> * env;
};

// the macro expands to scm_type_t * scm_make_base_nspace()
DEF_WITH_WRAPPER(scm_make_base_nspace) {
  // We call the constructor the same way
  // we would without the wrapper class
  GCed<ScmEnv> * env = new GCed<ScmEnv>(nullptr);
  ...

  return alloc_nspace(env);
}
```

The **GCed<ScmEnv>** inherits all the **ScmEnv**'s methods but we can also call *getSharedPtr()* on it, which returns **shared_ptr<ScmEnv>**. A shared pointer to a garbage collected object cannot be created in a regular way. We cannot use *make_shared* – that way the object would be deleted as soon as the reference count became zero. Calling the **shared_ptr** constructor on an existing pointer would also cause trouble. The object deletion code invoked when the pointer goes out of scope would be invalid. Even if we provided a custom deallocator, we don't want the object to be freed when our shared pointer is deleted.

While there is no way of directly controlling the pointer's reference count, we can use a little trick – the **shared_ptr**'s *aliasing constructor*:

```
template< class Y >
shared_ptr( const shared_ptr<Y>& r, element_type *ptr );
```

"The aliasing constructor: constructs a shared_ptr which shares ownership information with *r*, but holds an unrelated and unmanaged pointer *ptr*. Even if this shared_ptr is the last of the group to go out of scope, it will call the destructor for the object originally managed by *r*. However, calling get() on this will always return a copy of *ptr*. It is the responsibility of the programmer to make sure that this *ptr* remains valid as long as this shared_ptr exists, such as in the typical use cases where *ptr* is a member of the object managed by *r* or is an alias (e.g., downcast) of r.get()."[24]

So, this certainly isn't the typical use case. Our shared pointer *r* is a static **shared_ptr** to **bool** (named *anchor*) which only serves as a placeholder in the alias constructor. This way we create a **shared_ptr** pointing to our **GCed** object but sharing ownership with *anchor*, that is the reference count is at least two and we know the last **shared_ptr** to go will be the *anchor* (because it is static). Documentation says that we have to make sure the unmanaged pointer is valid as long as *r* (*anchor* in our case), but we know the pointer is valid until the garbage collector frees our object, and that does not happen before the pointer itself is erased, exactly as we need.

3.3.5 Line editing functionality and user input parsing

We have an implementation of Scheme REPL in Scheme. To make it more interactive, linenoise library is used for processing user input. This gives us line editing capabilities such as those we find in interactive shells. Now the question is: How do we parse the user input and convert it to a Scheme data structure?

We already have the **StringReader** tokenizer but how do we tell where the string starts and where it ends? While we could use this tokenizer, it would limit us to one line input only (if we said that newline is the end of string).

A better way is to implement a custom input stream which will read the user input line by line into a buffer (using linenoise), while we take individual characters from it using our **FileReader**. That way, each next line is read only if the input parser expects another token (when we have an incomplete list or string).

The **FileReader**'s constructor expects a reference to C++ **istream** and we can construct **istream** with a custom **streambuf** that overrides virtual method *underflow* (called when we read from the stream):

```
streambuf::int_type readlinebuf::underflow() {
    if (gptr() < egptr()) {
        return traits_type::to_int_type(*gptr());
    }
    int base = 0, start = 0;

    if (eback() == &buffer[base]) {
        size_t diff = egptr() - &buffer[base];
        size_t put_back = diff < put_back_max ? diff : put_back_max;

        memmove(&buffer[base], egptr() - put_back, put_back);
        buffer.resize(put_back);
        start += put_back;
    }
    else { buffer.resize(0); }

    char * line = linenoise(prompt.c_str());
    prompt = "";
    if (!line) {
        return traits_type::eof();
    }
    linenoiseHistoryAdd(line);
    size_t n = strlen(line);

    buffer.insert(buffer.begin() + start, line, line + n);
    buffer.push_back('\n');
    n++;
    free(line);
    setg(&buffer[base], &buffer[start], &buffer[start] + n);
    return traits_type::to_int_type(*gptr());
}
```

3.3.6 Runtime library initialization and cleanup

When a compiled Scheme application is executed, the OS linker loads our runtime library, which is written partly in C++ and partly in Scheme. As we have discussed, each Scheme library can have a constructor function *init* whose pointer is placed in the global constructors array. We want to perform initialization at the C++ level as well. For that purpose, we can declare a static global instance of a class and place the necessary code in that class' constructor. The only downside to this approach is that if we have multiple static objects like this, there is no defined order in which their constructors will be called.

Still, we define a class **LibSetup** with its global instance and use it for initialization and also for cleanup (destructor is called at program exit):

```
LibSetup::LibSetup() {
    // Init random seed
    srand((uint32_t)time(nullptr));
    // Init absolute path to current working directory
    // and add it to search path (for library lookup)
    initCWDPath();
}

LibSetup::~LibSetup() {
   // Delete C++ objects that weren't garbage collected yet
   mem_cleanup();
}

static LibSetup lib_setup;
```

Apart from setting up random seed and search path, we need to initialize the JIT before its first use. However, since it may never be used, we don't want to do this in the static object's constructor every time. Instead, we can lazy load the JIT initialization class by declaring it static local:

```
InitJIT::InitJIT() {
    InitializeNativeTarget();
    InitializeNativeTargetAsmPrinter();
    InitializeNativeTargetAsmParser();
    jit = make_unique<ScmJIT>();
}

ScmJIT * InitJIT::getJIT() {
    return jit.get();
}

static ScmJIT * getJIT() {
    static InitJIT jit_obj;
    return jit_obj.getJIT();
}
```

Now, each time we want to use the JIT, we call the free-standing *getJIT* function. The constructor of **InitJIT** is called only the first time, however.

3.4 Implementation status

We have implemented a strictly functional subset of Scheme: All variables are immutable, functions do not have side effects, iteration has to be expressed by recursion. For an overview of the particular language constructs and functions implemented, refer to README.md in the project's source tree and to the example programs located at `llscheme/test/lls_programs`. Every implemented feature is demonstrated there on an example.

Among the major language features that are missing are exceptions, continuations, macros and the *set!* function. Also, the only data structure implemented is a list. Vectors are supported partly (command-line arguments are stored in a vector that can be read from Scheme).

We also haven't implemented any type inference algorithm yet.

Chapter 4

Testing

4.1 Correctness

We use a set of automated regression tests (written in Ruby) which compare compiler output (parser's debug and error messages) with the expected result of various basic example programs (valid and invalid):

- empty program
- incomplete string
- incomplete list
- incomplete definition
- invalid function definition
- invalid variable name in definition
- missing expression in definition
- incomplete argument list
- invalid expression in argument list
- incomplete binding list
- invalid expression in binding list
- incomplete body
- empty quote
- incomplete quote
- empty lambda
- incomplete lambda
- empty if
- missing then expression in if
- missing else expression in if
- empty list
- incomplete function call
- invalid keyword at first list position
- definition inside expression
- atom
- quoted keyword
- quoted list
- function call
- nested function call
- if expression
- nested if expression
- lambda function
- passing lambda function in a call
- calling lambda function in place
- function definition (long form)
- function definition (short form)
- variable definition
- let expression
- let inside if
- closure inside closure

Consistency of the generated code is checked by the LLVM function verifier and a set of custom assertions placed in our code. We also have several unit

tests verifying the functionality of individual C++ classes.

The runtime behavior of compiled programs is tested manually (that could be improved in the future). However, the runtime library (as well as compiler) is compiled with the clang address sanitizer (in case of debug build), so we can detect memory management problems including memory leaks.

Example and test programs written in Scheme can be found in README and the `llscheme/test/lls_programs` directory:

- **apply** – testing the *apply* function
- **closure** – testing simple and nested closures
- **cmdargs** – processing the command line arguments
- **eval** – testing the *eval* function
- **fwdref** – testing forward references
- **hellow** – "Hello world!"
- **map** – testing the *map* function
- **mulmat** – testing matrix multiplication
- **plus** – testing addition
- **redef** – testing redefinition of variables and functions
- **repl** – read-eval-print loop
- **sat** – randomized SAT solver

4.2 Performance

To test performance of our Scheme implementation, we have generated a set of random matrix pairs of different dimensions and multiplied them using a naive matrix multiplication algorithm. The source code of the algorithm was compiled to a native executable (by our compiler) before the computation. We have also run the same test in the Racket[9] and Guile[10] interpreters for comparison (see table 4.1).

As we can see, our implementation is quite fast on small input (there is no compilation overhead at the start). It also appears that llscheme is slightly faster than others when doing floating point arithmetic but we only use fixed size doubles (other versions of Scheme usually implement arbitrary-precision arithmetic). Overall, Racket (which probably performs JIT compilation and

[9] `https://racket-lang.org/`

[10] `http://www.gnu.org/software/guile/`

Table 4.1: Performance test results

Pairs	**Dimensions**	**Time [s]**		
		llscheme	**Racket**	**Guile**
Integer matrices				
10000	3×3	0.205	0.469	1.390
10000	10×10	5.367	2.377	16.750
100	100×100	162.336	19.608	60.138
Real number matrices				
10000	3×3	0.283	0.734	1.777
10000	10×10	5.720	6.360	21.220
100	100×100	170.972	27.628	88.042

optimization) is faster. Our implementation loses even to Guile when the algorithm processes large 100×100 matrices. We suspect this is caused by an inefficient garbage collection strategy (mark&sweep pauses caused by the GC library). This could be improved by custom implementation of a more sophisticated GC method (e.g. generational garbage collector).

Conclusion

Summary

In this thesis, we have analyzed properties of the LLVM internal representation and of the Scheme programming language. Then we have successfully devised a way of translating Scheme into LLVM IR.

For that purpose we have designed and implemented a compiler frontend capable of generating native executables (standalone programs, libraries and dynamically compiled expressions) from Scheme source code. We have also designed and implemented a runtime environment for the language.

Our implementation of Scheme supports:

- global and local definitions
- conditional expressions (**if**)
- tail recursion
- first-class functions, named or anonymous (**lambda**)
- closures (single level or nested)
- local variables (using the **let** block)
- list operations (*cons*, *car*, *cdr*, ...)
- numerical, logical, string and I/O operations
- dynamic code evaluation (*eval* function)
- dynamic calls (*apply* function)
- modules (implemented as native libraries with custom metadata)

We have verified correctness of the implementation using automated unit and regression tests and by compiling and running a set of sample programs.

We have also assessed performance of the compiled programs and discussed how it could be improved.

Future work

There are many language features we could add in the future: arbitrary-precision arithmetic, macros, exceptions, continuations, compound data types and mutable variables to name a few.

We should also focus on optimization techniques specific to dynamically typed languages, which are not implemented in LLVM. Type analysis and inference appears to be the most important step towards generating more efficient code that can be further improved by traditional optimization passes usually used for statically typed code. Runtime tracing and JIT compilation could also be used to identify further optimization opportunities and to emit faster specialized code respectively.

Garbage collection should also be improved, preferably by replacing the GC library with our own custom solution that can leverage runtime type information and that is incremental if possible.

Another idea would be to add foreign function interface (FFI) for simple interaction with C library functions, access the LLVM API this way and implement a self-hosting compiler (written in our Scheme and compiled by it).

Bibliography

[1] Sussman, G. J.; Steele, G. L., Jr. An Interpreter for Extended Lambda Calculus. Technical report, Cambridge, MA, USA, 1975.

[2] Lattner, C.; Adve, V. The LLVM Compiler Framework and Infrastructure Tutorial. In *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, Sep 2004.

[3] clang: a C language family frontend for LLVM [online]. 2016, [Cited 2016-04-15]. Available from: `http://clang.llvm.org/`

[4] Projects built with LLVM [online]. 2016, [Cited 2016-04-15]. Available from: `http://llvm.org/ProjectsWithLLVM/`

[5] Lattner, C. The Design of LLVM [online]. 2012, [Cited 2016-04-15]. Available from: `http://www.drdobbs.com/architecture-and-design/the-design-of-llvm/240001128`

[6] LLVM Programmer's Manual [online]. 2016, [Cited 2016-04-15]. Available from: `http://llvm.org/docs/ProgrammersManual.html`

[7] Kaleidoscope: Implementing a Language with LLVM [online]. 2016, [Cited 2016-04-15]. Available from: `http://llvm.org/docs/tutorial/index.html`

[8] LLVM Language Reference Manual [online]. 2016, [Cited 2016-04-15]. Available from: `http://llvm.org/docs/LangRef.html`

[9] Scheme (programming language) [online]. 2016, [Cited 2016-04-15]. Available from: `https://en.wikipedia.org/wiki/Scheme_(programming_language)`

[10] Scheme Reports Process [online]. 2015, [Cited 2016-04-15]. Available from: `http://www.scheme-reports.org/`

[11] [LLVMdev] Scheme on LLVM IR [online]. 2013, [Cited 2016-04-21]. Available from: https://groups.google.com/forum/#!topic/llvm-dev/F4LscGXEmjU

[12] Guile Reference Manual: Garbage Collection [online]. 2014, [Cited 2016-04-22]. Available from: https://www.gnu.org/software/guile/manual/html_node/Garbage-Collection.html

[13] Wilson, P. R. Uniprocessor Garbage Collection Techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, London, UK, UK: Springer-Verlag, 1992, ISBN 3-540-55940-X, pp. 1–42. Available from: http://dl.acm.org/citation.cfm?id=645648.664824

[14] Garbage Collection with LLVM [online]. 2016, [Cited 2016-04-22]. Available from: http://llvm.org/docs/GarbageCollection.html

[15] A garbage collector for C and C++ [online]. 2016, [Cited 2016-04-22]. Available from: http://hboehm.info/gc/

[16] FAQ: A garbage collector for C and C++ [online]. 2016, [Cited 2016-04-22]. Available from: http://hboehm.info/gc/faq.html

[17] Linux Programmer's Manual – ELF(5) [online]. 2016, [Cited 2016-04-23]. Available from: http://man7.org/linux/man-pages/man5/elf.5.html

[18] 3.12 Conditionals: if, cond, and, and or [online]. 2016, [Cited 2016-04-24]. Available from: https://docs.racket-lang.org/reference/if.html

[19] LLVM's Analysis and Transform Passes [online]. 2016, [Cited 2016-04-28]. Available from: http://llvm.org/docs/Passes.html

[20] Exception Handling in LLVM [online]. 2016, [Cited 2016-04-30]. Available from: http://llvm.org/docs/ExceptionHandling.html

[21] Dybvig, R. K. Exceptions and Conditions [online]. 2009, [Cited 2016-04-30]. Available from: http://www.scheme.com/tspl4/exceptions.html

[22] Frequently Asked Questions (FAQ) [online]. 2016, [Cited 2016-05-01]. Available from: http://llvm.org/docs/FAQ.html

[23] LLVM-C: C interface to LLVM [online]. 2016, [Cited 2016-05-01]. Available from: http://llvm.org/docs/doxygen/html/group__LLVMC.html

[24] cppreference.com – std::shared_ptr::shared_ptr [online]. 2016, [Cited 2016-05-03]. Available from: http://en.cppreference.com/w/cpp/memory/shared_ptr/shared_ptr

Appendix A

Acronyms

API Application programming interface

AST Abstract syntax tree

CFG Control flow graph

CPU Central processing unit

ELF Executable and Linkable Format

EOL End of list

FFI Foreign function interface

GC Garbage collector

GEP Get element pointer

I/O Input/output

IDE Integrated development environment

IR Intermediate representation

JIT Just-in-time (compiler)

REPL Read-eval-print loop

STL Standard Template Library

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `llscheme` ..... project config files, README, LICENCE, etc.
    - `bin` ..... executables
      - `Debug` ..... debug version
      - `Release` ..... release version
    - `build` ..... out-of-tree build directory
    - `include` ..... compiler header files
      - `elfio` ..... library for ELF parsing
      - `linenoise` ..... line editing library
      - `optionparser` ..... command line argument parser lib.
      - `runtime` ..... runtime header files
    - `src` ..... compiler source files
      - `runtime` ..... runtime source files
      - `test` ..... unit tests
    - `test` ..... post-build test scripts
      - `lls_programs` ..... Scheme example programs
        - `lib` ..... Scheme example libraries
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format