KING'S College LONDON

University of London

6CCS3PRJ Final Year
LJSP - A LISP to asm.js compiler

Final Project Report

Author: Jan Söndermann
Supervisor: Dr. Christian Urban
Student ID: 1134853
Programme of Study: BSc Computer Science

April 20, 2014

Originality avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary.

I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service.

Jan Söndermann
April 20, 2014

Word count of this report: 21741

Abstract

Over the course of the last decade, JavaScript has turned from an insignificant scripting language to one of the most widespread and commonly used programming languages in existence. The ubiquity of web browsers paired with the exclusivity of JavaScript in web programming have caused the language to be used in increasingly complex applications. This has made performance a top priority for browser developers.

The speed at which JavaScript code can be executed has improved greatly over the last few years, but can still be limiting when building larger web applications. One of the most hotly discussed topics in the community of JavaScript developers in 2013 was the release of asm.js, a small, statically typed subset of JavaScript that can be compiled to efficient machine code. Asm.js promises huge speed gains over plain JavaScript when used as the compilation target for other languages.

To evaluate this new technology and see if asm.js can live up to the hype surrounding it, we design a language called LJSP and build a compiler that compiles LJSP to asm.js. We further implement a ray tracer that serves as a benchmark for different reimplementations of small parts of the ray tracer in asm.js.

Acknowledgements

I would like to thank my supervisor Dr. Christian Urban for supervising my project. During the last year, he provided me with both freedom to explore my own ideas and guidance when I was stuck.

Contents

List of Figures

1 Introduction

One of the main trends of today's Internet is the shift from traditionally offline or standalone programs to online applications on the web. This phenomenon, often called "Web 2.0", is exemplified in the success of web applications such as GMail and Facebook.

This success of the web has brought with it the rise of the programming language it is build on: JavaScript, the only language supported by all major browsers. Statistics, such as the number of repositories created on GitHub (a popular code sharing website) using JavaScript, reflect this success. In 2013, JavaScript led this list by a substantial margin [2, 5][1].

Since JavaScript is no longer confined to being used for simple animations and input-validation tasks, the complexity of web applications has increased tremendously. Modern JavaScript frameworks, such as Ember.js or Meteor use the full register of functionality the language offers. These JavaScript frameworks are just as complex as traditional web frameworks that use other languages such as Ruby or Python.

Unfortunately, this proliferation of JavaScript has taken the language far beyond the tasks it was initially designed for. Additionally, there is a widespread consensus that the language has many shortcomings in its design. Brendan Eich, the creator of JavaScript, writes [13] about its early history:

> In April 1995 I joined Netscape in order to "add Scheme to the browser." [...]
>
> So in 10 days in May 1995, I prototyped "Mocha," the code name Marc Andreessen had chosen. [...]
>
> To overcome all doubts, I needed a demo in 10 days. I worked day and night, and consequently made a few language-design mistakes (some recapitulating bad design paths in the evolution of LISP), but I met the deadline and did the demo.

and further in [20]:

> If I had done classes in JavaScript back in May 1995, I would have been told that it was too much like Java or that

[1] Running the query given at [5] adjusted to include the entire year of 2013 yields 498467 repositories for JavaScript and 395097 for Ruby on the second place.

> JavaScript was competing with Java I was under marketing orders to make it look like Java but not make it too big for its britches [it] needed to be a silly little brother language.

Criticism of JavaScript has mostly focussed on two areas:

- Inconsistencies and mistakes in the design of the language. Common examples given for this include the confusion around JavaScript's large number of falsy values (`0`, `''`, `NaN`, `false`, `null` and `undefined`) and its set of reserved keywords, which includes a large number of words not in use by the language but does not include `NaN` and `undefined`. This makes it necessary to test for `undefined`ness using `typeof x === 'undefined'` avoid comparing against a redefined `undefined`. Further problems include unusual scoping and confusing automatic casts. For a long list of problems with JavaScript, see [7].

- Slow execution speed. Writing fast interpreters for JavaScript has been an extraordinarily difficult task for browser developers. This is due to the weak typing and generally extremely dynamic nature of JavaScript.

Programmers have responded to the first problem in ways that include limiting themselves to a subset of JavaScript that excludes the inconsistent and badly-designed parts (cf. the very popular book "JavaScript: The Good Parts" [7]). Another response has been to create new languages that compile to JavaScript, such as the languages described in the 'Similar Languages' section of the Background Chapter.

The second problem has been partly remedied by a new generation of JavaScript engines spearheaded by Google's V8, released in 2008 as part of Google Chrome. The other browser makers, including Mozilla soon followed by rewriting their own JavaScript engines. These new engines often brought impressive speed gains.

In early 2013, Mozilla released asm.js, a project that claims to take these two approaches to their logical conclusion. It defines an extremely limited, statically typed subset of JavaScript that can be executed very quickly. This subset is intended as compilation targets for high level languages.

The goal of our project is twofold:

- To design such a high level and provide a compiler for it that compiles the language to asm.js.

- To test the performance of this new language that we called LJSP and compare it to plain JavaScript.

To achieve the second goal, the project includes a ray tracer in JavaScript, parts of which were rewritten in LJSP and compiled to

asm.js. Rendering time of a 3D scene functioned as a benchmark that made it possible to evaluate the results of the project.

Over the course of the project, a LJSP to asm.js compiler was implemented. This compiler was later extended to also compile LJSP to C and LLVM IR, which in turn were compiled to asm.js code using Emscripten, a program described in the next chapter. The different versions of asm.js code were then compared using the included ray tracer.

2 Background

This chapter will describe the background and context of the LJSP project. It will give technical details of the technologies involved and justify the motivation for starting this project. It will also describe similar and related work done by others.

2.1 Relevant technologies

This section will give an overview of the programs and platforms used in the LJSP project.

2.1.1 asm.js

Asm.js is a small subset of JavaScript that can be compiled to very fast machine code. Benchmarks [23] show running times around twice the speed of native code. To showcase the speed of asm.js, researches at Mozilla compiled the Unreal gaming engine to asm.js, producing smooth and stutter free rendering inside the browser [4].

After its release, asm.js has generated enormous interest in the community in places such as [3]. The goal of LJSP is to evaluate the performance achievable with a language that compiles to asm.js and find out if the hype surrounding it is justified. This section will describe the concepts behind asm.js and give a technical description of its specification. The Implementation chapter below will build on this description to explain the asm.js code that the LJSP compiler outputs.

Asm.js is designed as a very limited subset of JavaScript that includes information typically not included in JavaScript code. If the author of asm.js compatible code adds a statement that marks the code as compatible, and the browser of the user viewing a website that includes this code supports asm.js, the JavaScript engine of the browser can use that additional information to compile the code to very efficient machine

code. This compilation happens ahead of running the program (this is sometimes called Ahead-of-time-compilation).

The way to opt-in to this optimisation is to add a `"use asm";` statement to the top of the function that contains the code[1]. Supporting browsers (currently only Firefox) will then compile the code and output a string such as `"successfully compiled asm.js code (total compilation time 25ms)"` in the case of successful compilation to the console when loading this function.

Most of the additional information that asm.js forces the programmer to include in his programs that would typically not be part of idiomatic JavaScript is type information. Unlike JavaScript, asm.js is statically typed, meaning every variable has a type inferable at compile time. The following examples will illustrate this difference. We start with a small function in standard JavaScript shown in Figure 2.1.

```
1 function add(a, b) {
2     return a + b;
3 }
```

Figure 2.1: A simple addition function in JavaScript

The name of this function suggests that it was probably conceived to add two numbers. It could, however, be called with `add("hello ", "world");` and would yield the concatenated string `"hello world"`. Neither is it restricted in the type of numbers it accepts: it can be called with an arbitrary combination of integers and floating point numbers.

The same function restricted to integers in asm.js would look as shown in Figure 2.2.

```
1 function add_ints(a, b) {
2     a = a|0;
3     b = b|0;
4
5     return (a + b)|0;
6 }
```

Figure 2.2: An asm.js function that computes the sum of two ints

The additional lines 2 and 3 are the kind of type information mentioned above. They declare both parameters `a` and `b` to be of type int. In other languages such as C, the parameters would be written as `int a, int b`. In valid asm.js, all parameters to every function get a type as-

[1]The syntax for this opt-in is copied from the popular and widespread `"use strict";` directive that makes JavaScript engines less lenient when parsing JavaScript code.

signed at the very top of the function. The somewhat strange syntax for this type declaration is explained in detail in the next few paragraphs.

The other change from the original JavaScript version to the asm.js version of the addition function is the inclusion of a bitwise OR[2] with 0 to the result of the sum in line 5. This is necessary because of the other main feature of asm.js besides the possibility of compiling it efficiently:

The asm.js specification [12] guarantees that valid asm.js code evaluates to the same result regardless of whether or not the browser has asm.js support. If we consider the situation where the browser does have asm.js support and the function `add_ints` defined in the example above is called with two very large integers, the compiled, native code would add these two large ints and cause an integer overflow. If, however, the browser does not support asm.js and treats the code as normal JavaScript, adding two integers large enough that normally an overflow would occur causes the result to be cast to a floating point number according to the JavaScript specification.

The bitwise OR in the line `return (a + b)|0;` ensures that even in the case of missing asm.js support, the result returned by the function would still be an overflowed integer. The reason for this is that following the JavaScript specification, the bitwise OR with 0 coerces the variable to type int before executing the OR-operation, as bitwise operations are not defined on floating point variables. These expressions are therefore known as "type coercions" and asm.js makes heavy use of them.

To illustrate type coercions with an example, consider the interactive JavaScript session shown in Figure 2.3.

```
1 > 2147483647 + 1
2 2147483648
3 > (2147483647 + 1) |0
4 -2147483648
```

Figure 2.3: Demonstrating type coercion in the JavaScript console

2147483647 is equal to $2^{31} - 1$, the largest value that can be saved in a 32 bit signed integer variable. If we add one to this value without any coercion, the result is transparently cast to a double and returned. If, however, we include a type coercion, the result overflows to -2^{31}.

Asm.js uses type coercions both to ensure that the code produces the same result in all circumstances and as type annotations for parameters as Figure 2.2 demonstrated.

In the remaining part of this section, will give an overview of the asm.js specification with a restriction on the parts that are relevant to the LJSP compiler. For reasons of space, this overview will leave out information that is unimportant for our project and is not meant as a

[2]The pipe symbol `|` is used for bitwise OR in JavaScript

general introduction to all the specifics of asm.js.

Asm.js code is organised in asm.js modules which are made up of a surrounding function that includes the `"use asm";` directive mentioned at the beginning of this section. As parameters, it can take a `stdlib` object that makes it possible to import certain functions from the JavaScript standard library into the asm.js module and an `ArrayBuffer`[3] object often called "heap". Besides this, the function can include:

- Global variables
- Functions
- Function tables
- A return statement

Valid asm.js functions include enough type information to be statically typable as explained above. They can declare variables, perform arithmetic operations on these variables, branch using `if` statements, read from and write to the heap and call functions either by name or by looking them up in a function table.

Function tables in asm.js are arrays of functions. These arrays must be of a size equal to a power of two. When using a function table to call a function, the variable that is used as index must be masked bitwise to ensure that function table lookup never fails with an out of bounds error [11]. This is a complex idea best illustrated with an example. Suppose our code includes a function table `ftable` that contains 8 functions, and a variable `x` that holds the index to a function in `ftable` that we would like to call. This call could be achieved with the statement `ftable[x & 7](param1, param2, ...);`. The bitwise AND with 7, or 111_2 in binary, ensures that regardless of the value `x` holds, no out of bounds error can occur, as the result of the bitwise AND will always be between 0 and 7. The reason asm.js allows multiple function tables is because all functions in a table must have the exact same type signature in parameters and return types.

The return statement at the end can make functions defined inside the module visible to the outside.

Concluding this section, we will explain the `ArrayBuffer` object mentioned above and the way that asm.js modules interact with it. In JavaScript, `ArrayBuffer` objects represent blocks of memory that can hold any kind of data. It is, however, not possible to access this memory directly using the `ArrayBuffer` object. Instead, a subclass of another class, `ArrayBufferView` is created as a view on the `ArrayBuffer`. These views all interpret the memory held in the `ArrayBuffer` to be of a specific type. The example given in Figure 2.4, created with the Chrome JavaScript console, illustrates these classes and their usage.

[3] In JavaScript, `ArrayBuffer` objects represent blocks of memory.

```
1  > var buf = new ArrayBuffer(12);  // 12 bytes
2  > var i16view = new Int16Array(buf);
3  > var i32view = new Int32Array(buf);
4
5  > i16view;
6  [0, 0, 0, 0, 0, 0]
7
8  > i32view;
9  [0, 0, 0]
10
11 > i32view[0] = 42; i32view[1] = 65536;
12
13 > i32view;
14 [42, 65536, 0]
15 > i16view;
16 [42, 0, 0, 1, 0, 0]
17
18 > new Float32Array(buf)
19 [5.885453550164232e-44, 9.183549615799121e-41, 0]
```

Figure 2.4: Demonstration of the `ArrayBuffer` class and its views

In line 1 of Figure 2.4, an `ArrayBuffer` that holds 12 bytes of memory gets created. In lines 2 and 3, we create two views to this `ArrayBuffer`. Both are of type integer, but `i16view` interprets the memory to be made up of ints with 2 bytes (16 bits), whereas `i32view` is a view that interprets the buffer as ints with length 4 (32 bits). This explains the result of the next two statements in lines 5 and 8: i16view is an array of length 6, because the 12 bytes of the `ArrayBuffer` get grouped into 6 ints of length 2 each. Equivalently i32view: its length is $12/4 = 3$.

In line 11, the four byte view is used to write two numbers to the array. The result of this is confirmed in lines 13 and 14 which reflect that change. Line 15 proves that the views do not hold any data by themselves but are all views to the same underlying `ArrayBuffer`. Note how the value $65536 = 2^{16}$ overflows from the less significant two bytes at `i16view[2]` to the next two bytes.

The fact that views interpret the underlying bits to be of a specific type is again exemplified by line 19 which does not, as one might expect, return `[42.0, 65536.0, 0]`, but instead interprets the bits of the two integers as 32 bit floats. This produces the scrambled values returned by the statement.

Similarly to function table calls explained above, accessing the values in an `ArrayBuffer` requires additional safeguards to make sure the result is not undefined. According to the asm.js spec, array dereference expressions must be of the form `x[(e & mask) >> i]` with `mask` being

equal to the number of bytes in the `ArrayBuffer`[4] minus 1 and `i` being equal to the number of bytes the data type of the view used for the access is based on (e.g. 4 in the case of `Int32View`). In combination, they prevent out-of-bounds accesses by discarding bits larger than the length of the array and shifting the index to the right size for the view. In practice, the mask turned out not to be required in heap access expressions for Firefox to accept the code as valid asm.js.

2.1.2 LLVM and Emscripten

In order to gather more data with which the performance of the compiler could be compared, the project was extended to also interact with LLVM and Emscripten. LLVM is a compiler project that includes and has spawned a large number of tools and programs related to the compilation of programs. At its core, it defines an intermediate representation called LLVM IR and an optimizer for this language. This intermediate representation aims to be a shared interface for front ends that parse programming languages and compile them to LLVM IR and back ends that take LLVM IR code and produce native code for a specific architecture such as x86 or arm.

While a large number of these front and back ends for various languages and architectures have been written, only one is relevant to the LJSP project: a LLVM IR to JavaScript compiler called Emscripten. Emscripten was written by the same group of researches that invented asm.js; in fact, asm.js was created to fill a need the Emscripten developers had for a better compilation target. They describe the challenges of writing a compiler from LLVM IR, a low-level language, to JavaScript, a high-level language in [22].

The LJSP compiler emits both C and LLVM IR code. This generated code can be compiled with Emscripten to asm.js. The Evaluation Chapter will compare the running time of this code that Emscripten emits with the asm.js code that LJSP generates directly.

2.1.3 Ray Tracing

Ever since the first computers were built, programmers have sought ways to generate pictures with programs. Especially with the rise of computer games, this has become a heavily researched area. Ray tracing is one such technique to generate images from description of 3D scenes. Because generating these images is a computationally intensive task, it is suited to be used for benchmarking.

Before we can describe the details of ray tracing, we have to introduce two terms that we will refer to in our explanation: A **3D Scene** is an arrangement of objects at specific coordinates in 3D space. Object can include shapes such as spheres, cubes or planes, lights and a camera.

[4] Which must be a multiple of 4096

The **camera** is not a visible object, but instead gives the perspective an angle from which the scene will be rendered. **Rendering** denotes the process of generating an image from a 3D scene and the program that does this is called a renderer.

Ray tracing produces relatively impressive results with simple techniques. It works by shooting rays from the camera into the scene. The ray tracer then traces these rays and determines whether they hit an object in the scene. If they do, the process of shooting rays into the scene is repeated recursively with the intersection point as the origin of the rays to achieve photorealistic effects such as reflection. If they do not, black or some other background colour is displayed.

2.2 Similar Languages

Because of the problems with the JavaScript language mentioned in the Introduction, a large number of programming languages that compile to JavaScript or replace it altogether have been developed. These languages all try to fix some of JavaScript's problems.

The most successful of these language is CoffeeScript[5]. CoffeeScript is a relatively thin layer on top of JavaScript and it is often easy for the programmer to predict for a snippet of CoffeeScript what the result of compiling it to JavaScript will look like. In fact, this is one of the features that the makers of CoffeeScript advertise their language with, but it also means that the language can only fix the more superficial problems with JavaScript's syntax, but it does nothing about execution speed.

Another such language is TypeScript[6], developed and published by Microsoft. As its name suggests, TypeScript adds static type information to JavaScript. Like CoffeeScript, TypeScript compiles to JavaScript. This means that while the type information included with TypeScript programs can be used at compile time to catch mistakes in the program that violate its types, it is no longer available at run time to speed up execution.

The third major language in this family of languages that compile to JavaScript is Dart[7], developed and published by Google. Unlike TypeScript, Dart features its own syntax separate from JavaScript's. While Dart can be compiled to JavaScript, Google also offers a virtual machine that can execute Dart code directly. Dart currently does not seem to have a lot of traction.

Besides these industrial strength languages intended for serious development, there is a large number of compilers for LISP-dialects to

[5] http://coffeescript.org/
[6] http://www.typescriptlang.org/
[7] https://www.dartlang.org/

JavaScript. Examples for this king of language are Outlet[8], Parenjs[9] and LiScript[10]. These projects implement LISPs of varying complexity and generate JavaScript code.

The languages mentioned above are similar to LJSP in that they compile to JavaScript. Unlike LJSP, however, they compile to high-level, often very readable JavaScript. This is not the goal of the LJSP project, which was motivated by the prospect of using asm.js as a compilation target.

One language that does compile to asm.js code is Low-Level JavaScript (LLJS)[11], a project that, like asm.js, was developed by people at Mozilla. LLJS's is strongly reminiscent of C with structs and manual memory management. Due to the limitations of asm.js, LLJS as well is quite limited in what can be expressed with it and does not seem to have seen a lot of adoption so far.

3 Specification

This chapter will introduce the LJSP language itself, first with a simple example and later with a number of formal descriptions. It will also give a formal grammar of the Intermediate Representation used in the compiler described further below.

3.1 LJSP

LJSP is a dialect of LISP, a language known for its unique syntax. It is a functional language that includes functions as first class objects. A LJSP program is made up of a number of `define`s that define functions and an optional expression that can call these functions.

With one small exception, LJSP is a subset of Scheme[1], which is another dialect of LISP. This property was very useful in the testing of LJSP code, because it made it possible to evaluate LJSP programs using existing Scheme interpreters.

After ray tracing was chosen as benchmark, LJSP's grammar was designed with the purpose of expressing the kind of functions used in a

[8] https://github.com/jlongster/outlet

[9] https://bitbucket.org/ktg/parenjs/overview

[10] https://github.com/viclib/liscript

[11] http://mbebenita.github.io/LLJS/

[1] The exception is LJSP's `neg` operator, which negates the sign of a number. Scheme uses - for this purpose.

ray tracer. This is why it includes a large number of primitive operations, including `sqrt`, `min/max` and `neg`, but currently no lists.

Figure 3.1 shows an example program in LJSP that calculates the 6th Fibonacci number. The Fibonacci function fib is defined as follows:

$$fib(n) = \begin{cases} 1 & \text{if } n \text{ is equal to 0 or 1} \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

The LJSP code is a natural expression of this mathematical definition. It is made up of a function definition and the expression `(fib 6)`, that calls this function.

```
1 (define (fib n)
2     (if (or (= n 0)
3             (= n 1))
4         1
5         (+ (fib (- n 1))
6            (fib (- n 2)))))
7 (fib 6)
```

Figure 3.1: Example program that calculates $fib(6)$

Figure 3.2 shows another example of a LJSP program. This program defines a simple function `apply_func` that takes two parameters and applies the first to the second parameter. In the expression, the successor function is then defined and `apply_func` gets called with that successor function and the value 3.

```
1 (define (apply_func f n) (f n))
2 (define (succ n) (+ n 1))
3
4 (apply_func succ 3)
```

Figure 3.2: Example of functions as first class objects

3.1.1 LJSP's Grammar

The grammar of LJSP in Extended Backus-Naur Form is given below.

⟨*program*⟩ ::= ⟨*defines*⟩ [⟨*expr*⟩] ⟨*defines*⟩

⟨*defines*⟩ ::= ⟨*define*⟩ ⟨*defines*⟩ | ϵ

⟨*define*⟩ ::= '`(define (`' ⟨*ident*⟩ ⟨*paramsPlus*⟩ '`)`' ⟨*expr*⟩ '`)`'

⟨*expr*⟩ ::= ⟨*double*⟩
  | ⟨*ident*⟩
  | '`(if`' ⟨*expr*⟩ ⟨*expr*⟩ ⟨*expr*⟩ '`)`'
  | ⟨*lambda*⟩
  | '`(let (`' ⟨*letblocks*⟩ '`)`' ⟨*expr*⟩ '`)`'
  | '`(`' ⟨*primOp*⟩ ⟨*args*⟩ '`)`'
  | '`(`' ⟨*expr*⟩ ⟨*args*⟩ '`)`'

⟨*double*⟩ ::= `-?(\d+(\.\d*)?|\d*\.\d+)`

⟨*ident*⟩ ::= `[a-zA-Z=*+/<>!?-][a-zA-Z0-9=*+/<>!?-_]*`

⟨*lambda*⟩ ::= '`(lambda (`' ⟨*paramsPlus*⟩ '`)`' ⟨*expr*⟩ '`)`'

⟨*params*⟩ ::= ⟨*ident*⟩ ⟨*params*⟩ | ϵ

⟨*paramsPlus*⟩ ::= ⟨*ident*⟩ ⟨*params*⟩

⟨*letblocks*⟩ ::= ⟨*letblock*⟩ | ⟨*letblock*⟩ ⟨*letblocks*⟩

⟨*letblock*⟩ ::= '`(`' ⟨*ident*⟩ ⟨*expr*⟩ '`)`'

⟨*primOp*⟩ ::= '`+`' | '`-`' | '`*`' | '`/`' | '`neg`'
  | '`>=`' | '`<=`' | '`=`' | '`not`' | '`<`' | '`>`' | '`and`' | '`or`'
  | '`min`' | '`max`'
  | '`sqrt`'

⟨*args*⟩ ::= ⟨*expr*⟩ | ⟨*expr*⟩ ⟨*args*⟩

The front end stages of the compiler (explained in detail in the Design Chapter) make use of a number of functions that are not part of LJSP's grammar, i.e. not meant for programmers writing code in LJSP. These functions are used internally, but they do appear in LJSP code that the front end stages generate. These internal functions are shown below.

⟨*expr*⟩ ::= ⟨*env*⟩
  | '`(make-closure`' ⟨*lambda*⟩ ⟨*env*⟩ '`)`'
  | '`(make-env`' ⟨*idents*⟩ '`)`'
  | '`(nth`' ⟨*int*⟩ ⟨*expr*⟩ '`)`'
  | '`(get-env`' ⟨*expr*⟩ '`)`'
  | '`(get-proc`' ⟨*expr*⟩ '`)`'
  | '`(hoisted-lambda`' ⟨*ident*⟩ ⟨*expr*⟩ '`)`'

⟨*int*⟩ ::= `-?(0|[1-9][0-9]*)`

⟨*idents*⟩ ::= ⟨*ident*⟩ ⟨*idents*⟩ | ϵ

3.2 Semantics of LJSP

In this section, we formally define the semantics of the LJSP language. In the expressions below, i stands for an identifier, v stands for a generic value, e stands for an expression, c stands for a context that an expression is evaluated in, d stands for a double value and Λ stands for a `(lambda...)`-expression.

3.2.1 Small-Step Semantics

$$\text{var}\frac{}{\langle i, c\rangle \;\to\; \langle v, c\rangle} \text{ if } c(i) = v$$

$$\text{let1}\frac{\langle e_1, c\rangle \;\to\; \langle e_1', c\rangle}{\langle(\text{let } i_1 = e_1,\; \ldots \text{ in } e), c\rangle \;\to\; \langle(\text{let } i_1 = e_1',\; \ldots \text{ in } e), c\rangle}$$

$$\text{let2}\frac{}{\langle(\text{let } i_1 = v,\; \ldots \text{ in } e), c\rangle \;\to\; \langle(\text{let } i_2 = e_2,\; \ldots \text{ in } e, c[i_1 \mapsto v]\rangle}$$

$$\text{let3}\frac{}{\langle(\text{let } i = v \text{ in } e), c\rangle \;\to\; \langle e, c[i \mapsto v]\rangle}$$

$$\text{func\_appl1}\frac{\langle e, c\rangle \;\to\; \langle e', c\rangle}{\langle(f_{name}\;\ldots,\; e,\;\ldots), c\rangle \;\to\; \langle(f_{name}\;\ldots,\; e',\;\ldots), c\rangle}$$

$$\text{func\_appl2}\frac{}{\langle(f_{name}\; v_1,\;\ldots,\; v_n), c\rangle \;\to\; \langle f_{body}, c[f_{param_1} \mapsto v_1,\;\ldots,\; f_{param_n} \mapsto v_n]\rangle}$$

$$\text{lambda\_appl1}\frac{\langle l, c\rangle \;\to\; \langle \Lambda, c\rangle}{\langle(l\;\ldots,\; e,\;\ldots), c\rangle \;\to\; \langle(\Lambda\;\ldots,\; e,\;\ldots), c\rangle}$$

$$\text{lambda\_appl2}\frac{\langle e, c\rangle \;\to\; \langle e', c\rangle}{\langle(\Lambda\;\ldots,\; e,\;\ldots), c\rangle \;\to\; \langle(\Lambda\;\ldots,\; e',\;\ldots), c\rangle}$$

$$\text{lambda\_appl3}\frac{}{\langle(\Lambda\; v_1,\;\ldots,\; v_n), c\rangle \;\to\; \langle \Lambda_{body}, c[\Lambda_{param_1} \mapsto v_1,\;\ldots,\; \Lambda_{param_n} \mapsto v_n]\rangle}$$

$$\text{prim\_op1}\frac{\langle e, c\rangle \;\to\; \langle e', c\rangle}{\langle(prim\;\ldots,\; e,\;\ldots), c\rangle \;\to\; \langle(prim\;\ldots,\; e',\;\ldots), c\rangle}$$

$$\text{prim\_op2}\frac{}{\langle(prim\; d_1,\;\ldots,\; d_n), c\rangle \;\to\; \langle d, c\rangle} \text{ if } prim(d_1,\;\ldots,\; d_n) = d$$

$$\text{if}\frac{\langle e_1, c\rangle \;\to\; \langle e_1', c'\rangle}{\langle(\text{if } e_1, e_2, e_3), c\rangle \;\to\; \langle(\text{if } e_1', e_2, e_3), c'\rangle}$$

$$\text{if}_{false}\frac{}{\langle(\text{if } \#\text{f}, e_2, e_3), c\rangle \;\to\; \langle e_3, c\rangle}$$

$$\text{if}_{true}\frac{}{\langle(\text{if } \#\text{t}, e_2, e_3), c\rangle \;\to\; \langle e_2, c\rangle}$$

3.2.2 Big-Step Semantics

$$\text{var}\frac{}{\langle i, c\rangle \;\Downarrow\; \langle v, c\rangle} \text{ if } c(i) = v$$

$$\text{let}\frac{\langle e_1, c\rangle \;\Downarrow\; \langle v_1, c\rangle \;\ldots\; \langle e_n, c\rangle \;\Downarrow\; \langle v_n, c\rangle \quad \langle e, c[i_1 \mapsto v_1,\;\ldots,\; i_n \mapsto v_n]\rangle \;\Downarrow\; \langle v, c[\ldots]\rangle}{\langle(\text{let } i_1 = e_1 \;\ldots\; i_n = e_n \text{ in } e), c\rangle \;\Downarrow\; \langle v, c\rangle}$$

$$\text{func\_appl}\frac{\langle e_1, c\rangle \;\Downarrow\; \langle v_1, c\rangle \;\ldots\; \langle e_n, c\rangle \;\Downarrow\; \langle v_n, c\rangle \quad \langle f_{body}, c[f_{param_1} \mapsto v_1,\;\ldots,\; f_{param_n} \mapsto v_n\rangle \;\Downarrow\; \langle v, c[\ldots]\rangle}{\langle(f_{name}\; e_1,\;\ldots,\; e_n), c\rangle \;\Downarrow\; \langle v, c\rangle}$$

$$\text{lambda\_appl}\frac{\langle l, c\rangle \;\Downarrow\; \langle \Lambda, c\rangle \quad \langle e_1, c\rangle \;\Downarrow\; \langle v_1, c\rangle \;\ldots\; \langle e_n, c\rangle \;\Downarrow\; \langle v_n, c\rangle \quad \langle \Lambda_{body}, c[\Lambda_{param_1} \mapsto v_1,\;\ldots,\; \Lambda_{param_n} \mapsto v_n\rangle \;\Downarrow\; \langle v, c[\ldots]\rangle}{\langle(l\; e_1,\;\ldots,\; e_n), c\rangle \;\Downarrow\; \langle v, c\rangle}$$

$$\text{prim\_op}\frac{\langle e_1, c\rangle \;\Downarrow\; \langle v_1, c\rangle \;\ldots\; \langle e_n, c\rangle \;\Downarrow\; \langle v_n, c\rangle}{\langle(prim\; e_1,\;\ldots\; e_n), c\rangle \;\rightarrow\; \langle v, c\rangle} \text{ if } prim(v_1,\;\ldots,\; v_n) = v$$

$$\text{if}_{false}\frac{\langle e_1, c\rangle \;\Downarrow\; \langle \#\text{f}, c\rangle \quad \langle e_3, c\rangle \;\Downarrow\; \langle v, c\rangle}{\langle(\text{if } e_1,\; e_2,\; e_3), c\rangle \;\rightarrow\; \langle v, c\rangle}$$

$$\text{if}_{true}\frac{\langle e_1, c\rangle \;\Downarrow\; \langle \#\text{t}, c\rangle \quad \langle e_2, c\rangle \;\Downarrow\; \langle v, c\rangle}{\langle(\text{if } e_1,\; e_2,\; e_3), c\rangle \;\rightarrow\; \langle v, c\rangle}$$

3.3 The Intermediate Representation (IR)

The Intermediate Representation (IR) of the LJSP compiler is a separate language used in the intermediate stages of the compiler. It is not to be confused with the LLVM Intermediate Representation, which is an output target of the LJSP compiler.

3.3.1 Grammar

The grammar of the IR is given below, again in Extended Backus-Naur Form. It uses some of the rules defined in the grammar for LJSP above.

⟨*iModule*⟩ ::= ⟨*iFunctions*⟩

⟨*iFunctions*⟩ ::= ⟨*iFunction*⟩ ⟨*iFunctions*⟩ | ϵ

⟨*iFunction*⟩ ::= '`function`' ⟨*ident*⟩ '`(`' ⟨*params*⟩ '`)`' '`{`' ⟨*iStatements*⟩ '`}`'

⟨*iStatements*⟩ ::= ⟨*iStatement*⟩ ⟨*iStatement*⟩ | ϵ

⟨*iStatement*⟩ ::= ⟨*ident*⟩ '`=`' ⟨*iExpr*⟩
  | '`if (`' ⟨*ident*⟩ '`) {`' ⟨*iStatements*⟩ '`} else {`' ⟨*iStatements*⟩ '`}`'
  | ⟨*iExpr*⟩

⟨*iExpr*⟩ ::= ⟨*ident*⟩
  | ⟨*double*⟩
  | ⟨*ident*⟩ '`(`' ⟨*iArgs*⟩ '`)`'
  | ⟨*iUnaryPrimOp*⟩ ⟨*iExpr*⟩
  | ⟨*iExpr*⟩ ⟨*iBinaryPrimOp*⟩ ⟨*iExpr*⟩
  | '`make-env(`' ⟨*iIdents*⟩ '`)`'
  | '`make-hl(`' ⟨*ident*⟩ '`,`' ⟨*ident*⟩ '`)`'
  | ⟨*ident*⟩ '`[`' ⟨*index*⟩ '`]`'

⟨*iArgs*⟩ ::= ⟨*iExpr*⟩ | ⟨*iExpr*⟩ ⟨*iArgs*⟩

⟨*iUnaryPrimOp*⟩ ::= '`sqrt`'

⟨*iBinaryPrimOp*⟩ ::= '`+`' | '`-`' | '`*`' | '`/`'
  | | '`==`' | '`<`' | '`>`'
  | '`min`' | '`max`'

⟨*iIdents*⟩ ::= ⟨*ident*⟩ | ⟨*ident*⟩ ⟨*iIdents*⟩

⟨*index*⟩ ::= `0 | [1-9][0-9]*`

4 Design

This chapter will describe the design of the LJSP compiler. It will give an abstract, theoretical description of all the subsystems that make up the compiler. The next chapter will then explain the concrete implementation details of these subsystems. Together, these two chapters aim to give a full understanding of the LJSP compiler.

4.1 Architecture of the Compiler

The compiler is made up of two major components: The first is a hierarchy of classes that get instantiated to create an internal representation of the program called the Abstract Syntax Tree (AST). The second is a substantial number of compilation stages that perform various transformation on this AST to bring it into a form closer to the desired output.

When compiling a program, the compiler will first create the AST of the input program given. It will then run this AST through a subset of the available stages depending on the desired output (e.g. asm.js or C) before finally converting the resulting AST back to a textual representation.

In the following, we will first describe the AST classes before going through all the individual stages of the compiler in detail.

4.1.1 The Abstract Syntax Tree

The Abstract Syntax Tree (AST) is the representation of the program that the compiler uses internally. It gets generated by the first stage, parsing, whose purpose it is to convert the user's textual representation of the program to an AST. Correspondingly, the compiler includes functions that convert an AST back to program code in text form.

As its name suggests, the Abstract Syntax Tree is a Tree representation of the code. As an illustration of the structure of these trees, we give the AST for the expression `(* (+ x 1) 2)` in Figure 4.1.

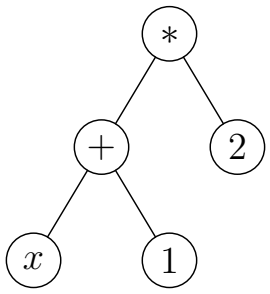

Figure 4.1: Example of an Abstract Syntax Tree

All nodes in the Abstract Syntax Tree are objects. These objects are instances of classes that very closely reflect the grammar of the language the program that the AST represents is written in. This means that every language used in the LJSP compiler (LJSP itself, the Intermediate Representation described in the next section and all output languages) has its own hierarchy of AST classes. Some compilation stages manipulate the AST without changing the types of the objects it is made up of and some convert the AST from one class hierarchy (i.e. language) to another.

4.1.2 Compilation Stages

Because compiling a program from one programming language to another is often a very complex transformation, most compilers break up this transformation into a number of stages[1] that each accomplish a

[1] We will use the terms "stage", "conversion" and "translation" interchangeably.

small part of this overall transformation. While these stages can vary quite significantly in complexity ranging from simple expansions of specific expressions (an example would be converting expressions of the form `1 + 2 + 3 + 4` to `((1 + 2) + 3) + 4` to make the order of execution explicit) to more complex transformations such as CPS-translation, they are all of manageable complexity when compared to the overall goal of transforming code from the input to the output language.

All stages in the compiler take an produce an Abstract Syntax Tree which they manipulate[2]. Like most compilers, including industrial-strength compilers such as LLVM, the LJSP compiler is made up of compilation stages as described above. These stages make up the core of the compiler and will be described in detail in the rest of this chapter. An important observation on compilation stages is that they should not change what the AST evaluates to, i.e. not change the meaning of the program.

Overview of compilation stages

One important consideration when structuring a compiler in this way is to avoid code duplication. Figure 4.2 shows a high level overview of the compilation stages of the LJSP compiler. It shows that regardless of the possible output languages the user of the compiler chooses, all programs pass through the intermediate stages coloured in red. This makes them ideal for transformations that are shared among all output languages, such as code optimisations.

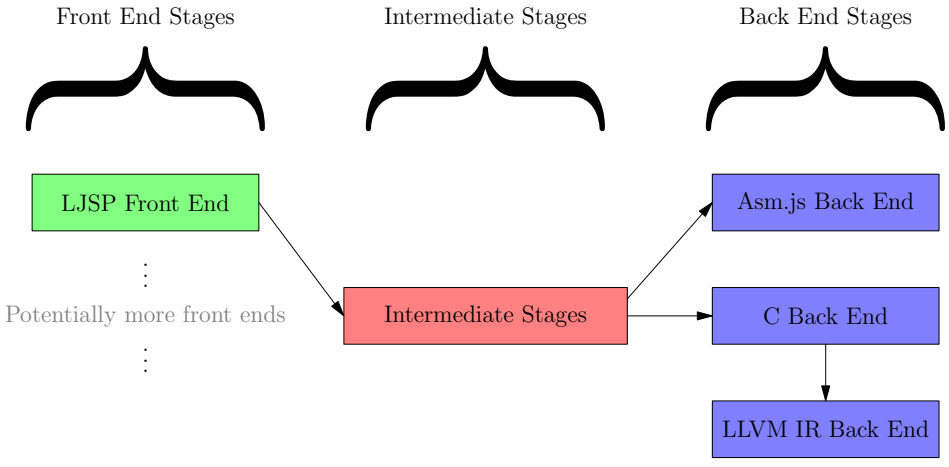

Figure 4.2: High level overview of LJSP compilation stages

[2]An exception to this are the very first and very last stages, whose purpose it is to convert between textual representations of the program and the AST.

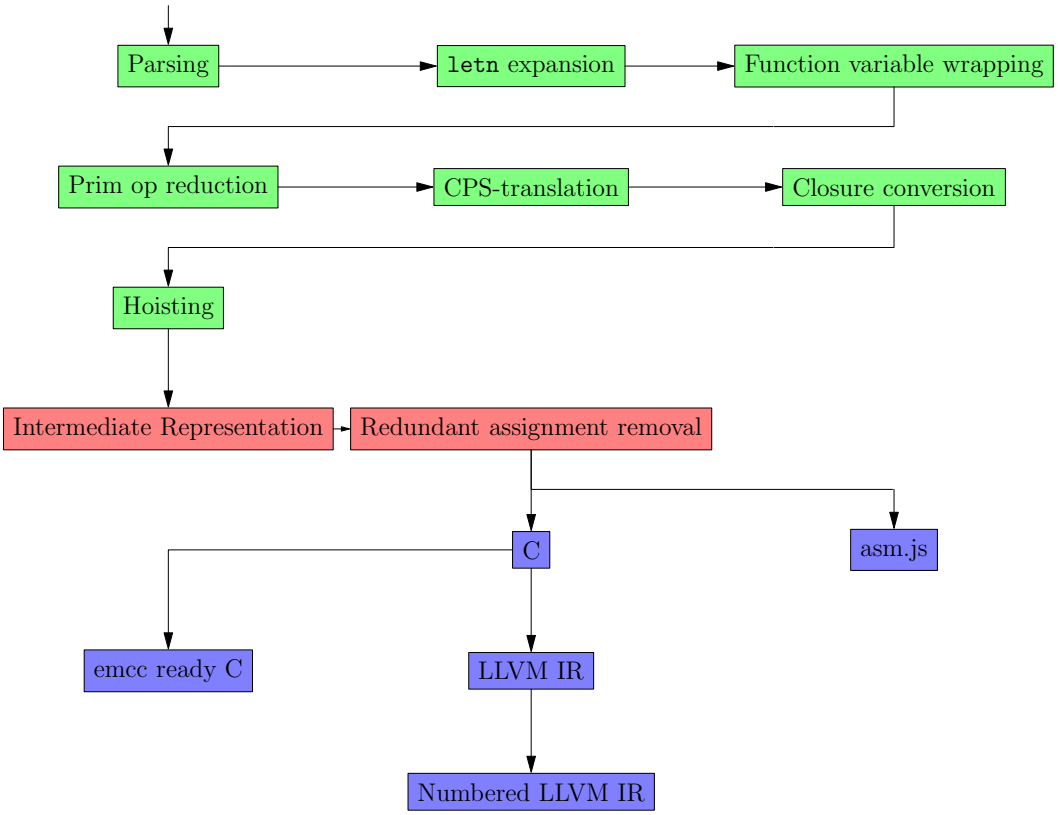

Figure 4.3: Flowchart for all compilation stages in LJSP

Figure 4.3 shows all the compilation stages that the LJSP compiler is made up of. The colours of the stages in the detailed flowchart correspond to the three colours used for front end, intermediate and back end stages in Figure 4.2.

The stages of the compiler can be grouped into three sections:

- **Front end stages**: These first seven stages make up the first part of the transformations the code passes through. Their structure is that of a simple list and there is no branching of stages in the front end. These stages all operate on LJSP ASTs, reducing the complexity of the code. The decision to not use internal languages for as long as possible in the chain of stages was made early in the development of the compiler. It simplified testing these stages because the result of all of these stages can be tested with a Scheme interpreter. Letn expansion and prim op reduction reduce the number of distinct instructions in the code, function wrapping makes it possible to use function names as variables, CPS-translation makes control flow explicit while the latter two stages remove anonymous function objects by turning them into

named functions. This reduction of complexity is necessary to bridge the gap between LJSP, a high-level language that includes such concepts as functions as first class variables, and low-level output languages such as LLVM IR that do not include these concepts. In the diagram, front end stages are depicted as boxes with a green background.

- **Intermediate stages**: These stages convert the LJSP AST obtained after passing through the front end stages into a different language called the Intermediate Representation (IR) of the LJSP compiler before performing various optimisations on this IR code. IR is an imperative, untyped language, that is general enough to be converted to both asm.js and C but close enough to these languages to avoid code duplication. This part of the compiler currently consists of two stages, one that converts LJSP to IR and a simple optimisation stage that removes redundant assignments. It offers a suitable framework for adding additional optimisations that automatically get shared among all output languages. In the diagram, intermediate stages are shown as boxes with a red background.

- **Back end stages**: This part of the compiler comprises all the stages that result in an output language. The relationships between these stages are more complex and include some branching. In the diagram, back end stages are shown as boxes with a blue background.

The compiler initially included a limited number of front end stages, no intermediate stages and asm.js as its only back end. When the C back end was added, the Intermediate Representation was introduced to remove code duplication between the asm.js and the new C back end. From there, both the front end and back end stages grew in number. Additional front end stages were included to add functionality to the LJSP language and more back ends were added to compile LJSP to other programming languages such as LLVM IR.

4.2 Detailed Description of the Compilation Stages

We will now describe all the stages of the LJSP compiler in detail. For the stages were this added useful information, the explanation will include a set of equations to define the stages in a precise, formal way. In these equations, y will stands for a generic variable, d for a double constant and f for a fresh identifier. Functions of the form $\mathcal{A}[\![e]\!]$ will denote transformation $\mathcal{A}$ applied to expression e. This syntax is borrowed from [16].

4.2.1 Parsing

Parsing is the first stage in the tree of compilation stages in the LJSP compiler. It takes the textual representation of the program and builds an equivalent Abstract Syntax Tree. It does so according to the grammar of the LJSP language described in the Specification Chapter.

The LJSP compiler implements parsing using parser combinators that combine parsers for simpler expressions to form parsers for more complex expressions. This reflects the way that LJSP's grammar is written down. For example, Figure 4.4 shows the grammar rule for `define`-expressions. This is one of the more complex expressions and it combines three simpler rules: ⟨*ident*⟩, ⟨*params*⟩ and ⟨*expr*⟩.

⟨*define*⟩ ::= '`(define (`' ⟨*ident*⟩ ⟨*params*⟩ '`)`' ⟨*expr*⟩ '`)`'

Figure 4.4: Grammar rule for `define`s

4.2.2 `letn` expansion

This is a small stage that expands all `let`-expressions that introduce more than one identifier into a number `let`-expressions that all define one variable each. Applying `letn` conversion to the expression `(let ((a 1) (b 2)) (+ a b))` would yield `(let ((a 1)) (let ((b 2)) (+ a b)))`. While not strictly necessary, this conversion greatly simplifies later stages such as CPS-translation, as they do not have to consider the possibility of multiple definition blocks inside `lets`. Equations for this stage are given in Figure 4.5.

$$\mathcal{L}[\![(\text{let } ((idn_1\ e_1)\ (idn_2\ e_2)\ldots (idn_n\ e_n))\ e)]\!] \stackrel{\text{def}}{=} (\text{let } ((idn_1\ e_1))\ (\text{let } ((idn_2\ e_2))\ldots (\text{let } ((idn_n\ e_n))\ e)))$$

$$\mathcal{L}[\![e]\!] \stackrel{\text{def}}{=} e \text{ with } \mathcal{L} \text{ applied to all sub-expressions in e}$$

Figure 4.5: `letn` conversion

4.2.3 Prim op reduction

This stage reduces a number of primitive operations (these are operations such as additions or boolean operations) to a combination of other primitive operations. This reduces work in back end stages, as only as limited subset of primitive operations that are part of the LJSP grammar need to get converted to the respective language that the back end generates. One example a primitive operation that the parser recognises

but that back end stages do not have to handle is the negation operator `neg` that switches the sign of a double. Prim op reduction reduces expressions of the form `(neg x)` to simple subtraction: `(- 0 x)`.

After the completion of this stage, the following primitive operations will have been removed:

- The arithmetic negation operator `neg`. In most other languages, `-` is used for this. This operator is reduced to a subtraction from zero; the expression `(neg x)` gets converted to `(- 0 x)`.
- The boolean negation operator `not`. This is often written as `!` in computer science or $\neg$ in mathematics. If-expressions of the form `(if (not` e_1`)` e_2 e_3`)` have their true and false branch swapped and get reduced to `(if` e_1 e_3 e_2`)`.
- The boolean `and`. Expressions of the form `(if (and` e_1 e_2`)` e_3 e_4`)` are converted to two nested if-expressions: `(if` e_1 `(if` e_2 e_3 e_4`)` e_4`)`
- The boolean `or`: If-expressions of the form `(if (or` e_1 e_2`)` e_3 e_4`)` get converted to two nested if-expressions: `(if` e_1 e_3 `(if` e_2 e_3 e_4`))`, analogously to `and` conversion
- `>=`. Expressions of the form `(>=` e_1 e_2`)` are reduced to the expression `(not (<` e_1 e_2`))` which then gets reduced recursively according to the rule to remove `not`s explained above.
- `<=`, equivalent to `>=` above.

4.2.4 Function Variable Wrapping

$$\mathcal{W}[\![y]\!] \stackrel{\text{def}}{=} \lambda p_1, \ldots .(y\ p_1, \ldots)$$
$$\text{if } y \text{ is the name of a function}$$
$$\mathcal{W}[\![e]\!] \stackrel{\text{def}}{=} e \text{ with } \mathcal{W} \text{ applied to all sub-expressions in e}$$

Figure 4.6: Function Variable Wrapping

This stage wraps function variables that contain named functions (those that are defined with a `define` statement in the program) in `lambda` expressions. To motivate this problem, consider the program shown in Figure 4.7.

In line 6, both `f1` and `f2` contain the successor function and the expression evaluates to 3. They do, however, vary in the type of function they contain: `f1` contains a reference to a named function, whereas `f2` contains a `lambda` or anonymous function. While this does not cause any

```
1 ; The successor function
2 (define (succ n) (+ n 1))
3
4 (let ((f1 succ)
5       (f2 (lambda (n) (+ n 1))))
6   (f1 (f2 1)))
```

Figure 4.7: Two types of function variables

problems at this stage of the compilation, once the program has been closure converted, lambdas and functions have to get called differently[3]. Because the variables do not give any information about what kind of function they contain, it would then be impossible to determine how they should be used.

To solve this problem, this stage wraps `lambda`-expressions around all named functions that are used as variables and effectively converts all function variables to lambdas. These lambdas take the same number of parameters as the named function they wrap and do nothing but call the function itself with the parameters given to the lambda. Line 4 of the program shown in Figure 4.7 would get converted accordingly to: `(let ((f1 (lambda (n) (succ n))) ....`

4.2.5 CPS-Translation

This stage converts the program to continuation-passing style (CPS). Because it is the central stage of the compilation pipeline, we will first explain continuation-passing style in general before describing the particularities of the CPS-translation in the LJSP compiler.

Continuation-Passing Style

Code that is in continuation-passing style gives explicit names to every intermediate computation and makes control flow explicit by never returning from a function call [6, 16]. To illustrate this concept, we show the step-by-step transformation of a simple function `f`, given in Figure 4.8, to continuation-passing style.

```
1 function f(a, b) {
2   return a + b + 10
3 }
```

Figure 4.8: A simple function that is to be CPS-translated

[3]For more details on why that is, please see the description in the section on closure conversion below.

First, we give a name to the result of the additions that the function returns. This change results in the code given in Figure 4.9.

```
1 function f1(a, b) {
2   t1 = a + b + 10
3   return t1
4 }
```

Figure 4.9: Assigning the sum to temporary variable

To conform to the requirement that every intermediate result be explicitly named, we need to break up the long sum into two computations. This makes it clear that the first addition gets precedence before the second. The result can be seen in Figure 4.10.

```
1 function f2(a, b) {
2   t2 = a + b
3   t1 = t2 + 10
4   return t1
5 }
```

Figure 4.10: Breaking up the addition further

The last step in our example is the one that gives continuation-passing style its name. Functions in CPS-conform programs do not return, but instead receive a function, usually called "continuation", that the caller passes in an additional parameter. This continuation can be understood as the remainder of the program that processes the result computed by the function. This is shown in Figure 4.11.

```
1 function f3(cont, a, b) {
2   t2 = a + b
3   t1 = t2 + 10
4   cont(t1)
5 }
```

Figure 4.11: Introducing the continuation in an additional parameter

LJSP Code that is CPS-conform is nearly linear: it consists of a sequence of let-expressions followed by a single function call. The only expression that violates this linearity is the if-expression that branches execution flow.

To further illustrate the concept of continuations, we give a short program written in Scheme below. Scheme is unique among most programming languages in that it gives the programmer access to the con-

tinuations in his program using a function called `call/cc`[4]. `call/cc` takes as its one parameter a function, which gets called with the current continuation as argument. Our example program uses `call/cc` to save and later reuse a continuation.

We first define a global variable `cont` that will later hold our saved continuation. Global variables in Scheme need to be defined before they can be assigned to with `set!` later; right now `cont` holds a placeholder value of 0. We also define a function `set-cont` that takes a parameter, sets `cont` to its value and returns 1. This may seem redundant, but it simplifies the following code snippets.

```
1 > (define (cont) 0)
2 > (define (set-cont c) (set! cont c) 1)
```

Figure 4.12: Preparations for the `cont/cc` example

Next, we assign our continuation. `call/cc` gets called with set-cont which in turn gets called with the continuation (remember that `call/cc` takes a function which it calls with the current continuation as its parameter, hence the name "call with current continuation"). The return value of set-cont, which is 1, is the value that gets passed to the arithmetic operations and this line returns 3.

```
1 > (+ 1 (* 2 (call/cc set-cont)))
2 3
```

Figure 4.13: Saving a continuation

We can now call this continuation, saved in `cont`, with arbitrary arguments value as shown in Figure 4.14.

```
1 > (cont 2)
2 5
3 > (cont 10)
4 > 21
```

Figure 4.14: Calling a saved continuation

One way of visualising the continuation is to write it as `(+ 1 (* 2 ...))`. This is the continuation of the code at `...` Whatever value this code computes, it passes it on to our continuation. It is worth noting that the continuation is not just simply a function that multiplies its argument by 2 and adds 1, as the code in Figure 4.15 illustrates.

[4]call/cc stands for "call with current continuation". Unlike in most other languages, identifiers in Scheme can contain the '/' character.

```
1 > (+ 1000 (cont 2))
2 5
```

Figure 4.15: Replacing one continuation with another

Calling `cont` substitutes the continuation that we see in the code above, `(+ 1000 ...)`, with our saved continuation.

CPS-Translating LJSP

$$
\begin{aligned}
&\mathcal{K}[\![y]\!]k && \stackrel{\text{def}}{=} k(y)\\
&\mathcal{K}[\![d]\!]k && \stackrel{\text{def}}{=} k(d)\\
&\mathcal{K}[\![(\text{if } e_1\ e_2\ e_3)]\!]k && \stackrel{\text{def}}{=} \mathcal{K}[\![e_1]\!]\lambda x.(\text{if } x\ \mathcal{K}[\![e_2]\!]k\ \mathcal{K}[\![e_3]\!]k)\\
&\mathcal{K}[\![(\text{define } (name, p_1, \ldots, p_n)\ e)]\!]k && \stackrel{\text{def}}{=} k((\text{define } (name, f_{cont}, p_1, \ldots, p_n)\\
&&&\qquad \mathcal{K}[\![e]\!]f_{cont}))\\
&\mathcal{K}[\![(\text{lambda } (p_1, \ldots, p_n)\ e)]\!]k && \stackrel{\text{def}}{=} k((\text{lambda } (f_{cont}, p_1, \ldots, p_n)\\
&&&\qquad \mathcal{K}[\![e]\!]f_{cont}))\\
&\mathcal{K}[\![(proc\ p_1, \ldots, p_n)]\!]k && \stackrel{\text{def}}{=} \mathcal{K}[\![p_1]\!]\lambda x_1. \ldots\\
&&&\qquad \mathcal{K}[\![p_n]\!]\lambda x_n.\\
&&&\qquad\quad (proc\ \lambda x_k.k(x_k), x_1, \ldots, x_n)\\
&\mathcal{K}[\![(prim\ p_1, p_2, \ldots, p_n)]\!]k && \stackrel{\text{def}}{=} \mathcal{K}[\![(prim\ (prim\ (prim\ p_1, p_2)\ p_3) \ldots p_n)]\!]k\\
&\mathcal{K}[\![(prim\ p_1, p_2)]\!]k && \stackrel{\text{def}}{=} \mathcal{K}[\![p_1]\!]\lambda x_1.\mathcal{K}[\![p_2]\!]\lambda x_2.\\
&&&\qquad (\text{let } f = (prim\ x_1, x_2) \text{ in } k(f))\\
&\mathcal{K}[\![(\text{let } ((idn\ e_1))\ e_2)]\!]k && \stackrel{\text{def}}{=} \mathcal{K}[\![e_1]\!]\lambda x_1.\\
&&&\qquad (\text{let } idn = x_1 \text{ in } \mathcal{K}[\![e_2]\!]\lambda x_2.(k(x_2)))
\end{aligned}
$$

Figure 4.16: CPS-translations of LJSP expressions

Figure 4.16 shows CPS-translations for all LJSP expressions. In the equations, $\mathcal{K}[\![e]\!]k$ stands for the CPS-translation of expression e with continuation k.

The transformation for both the `define` and `lambda`-expressions CPS-translate the body of the function with a continuation passed to the function in a new parameter called f_{cont} in the equations and `cont_n` in the code. This parameter has a correspondence in the translation for applications in the next equation: it is the $\lambda x_k.k(x_k)$ that gets added to the application when translating. The translation for primitive opera-

tions consists of two equations in the diagram, one equation to convert operations with more than two operands into multiple operations with two operands each and one equation to convert it to continuation-passing style. Note that unlike the translation of regular function applications, the application of primitive operations does not include the additional continuation parameter just mentioned, as primitive operations are allowed to return a result.

This stage makes one additional change to the code that solves a problem that arises when non-CPS-translated code wants to call a CPS-translated function: The function requires a continuation that it can call with its result, but the calling code is not in continuation-passing style and requires the function it calls to return an ordinary return value. Therefore, after CPS-translating the program, the LJSP compiler makes a copy of each function in the program with the suffix `_copy` added to its name. These new functions take the same number of arguments as the original version they copied minus one (the new continuation parameter added to the functions in the program). The body of the copied functions, however, is replaced by a call to the original function plus the identity function as continuation.

We illustrate this with a simple function. The original function before CPS-translation is given in Figure 4.17. The CPS-translation stage turns this into the two functions shown in Figure 4.18.

```
1 function add(a, b):
2     return a + b
```

Figure 4.17: Flowchart for compilation stages in LJSP

```
1 function add(cont, a, b):
2    cont(a + b)
3
4 function add_copy(a, b):
5     return add(lambda x: x,
6                a, b)
```

Figure 4.18: Flowchart for compilation stages in LJSP

These copied functions later function as entry points to the code. This is so that the user of the generated modules does not have to give a continuation herself; instead, it gets constructed by the `_copy` function.

4.2.6 Closure Conversion

Besides CPS-translation, closure conversion is the other core stage in the front end of the LJSP compiler. The goal of closure conversion,

$$
\begin{array}{ll}
\mathrm{FV}(y) & \stackrel{\mathrm{def}}{=} \{y\} \\
\mathrm{FV}(d) & \stackrel{\mathrm{def}}{=} \emptyset \\
\mathrm{FV}((\text{if } e_1\ e_2\ e_3)) & \stackrel{\mathrm{def}}{=} \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \cup \mathrm{FV}(e_3) \\
\mathrm{FV}((\text{let } ((idn\ e_1))\ e_2)) & \stackrel{\mathrm{def}}{=} \mathrm{FV}(e_1) \cup (\mathrm{FV}(e_2) - \{idn\}) \\
\mathrm{FV}((\text{define } (name, p_1, \ldots, p_n)\ e)) & \stackrel{\mathrm{def}}{=} \mathrm{FV}(e) - \{name\} - \{p_1, \ldots, p_n\} \\
\mathrm{FV}((\text{lambda } (p_1, \ldots, p_n)\ e)) & \stackrel{\mathrm{def}}{=} \mathrm{FV}(e) - \{p_1, \ldots, p_n\} \\
\mathrm{FV}((proc\ p_1, \ldots, p_n)) & \stackrel{\mathrm{def}}{=} (\mathrm{FV}(proc) - \{\text{defined functions}\}) \cup \\
 & \qquad \{f \mid f \in \mathrm{FV}(p) \text{ with } p \in \{p_1, \ldots, p_n\}\} \\
\mathrm{FV}((prim\ p_1, \ldots, p_n)) & \stackrel{\mathrm{def}}{=} \{f \mid f \in \mathrm{FV}(p) \text{ with } p \in \{p_1, \ldots, p_n\}\}
\end{array}
$$

Figure 4.19: Finding free variables

$$
\begin{array}{ll}
\mathcal{C}[\![(\text{lambda } (p_1, \ldots, p_n)\ e)]\!] \stackrel{\mathrm{def}}{=} & \text{make-closure(} \\
 & \quad (\text{lambda } (env, p_1, \ldots, p_n)\ e_{new})) \\
 & \text{with } e_{new} = \\
 & \quad \text{assign } \mathrm{FV}(e) \text{ to env elements} \cup \\
 & \quad \mathcal{C}[\![e]\!]
\end{array}
$$

Figure 4.20: Closure converting lambdas

$$
\begin{array}{ll}
\mathcal{C}[\![(func\ p_1, \ldots, p_n)]\!] & \stackrel{\mathrm{def}}{=} (func\ \mathcal{C}[\![p_1]\!], \ldots, \mathcal{C}[\![p_n]\!]) \\
\mathcal{C}[\![(closure\ p_1, \ldots, p_n)]\!] & \stackrel{\mathrm{def}}{=} (\mathcal{C}[\![closure]\!].code\ \mathcal{C}[\![closure]\!].env \\
 & \qquad \mathcal{C}[\![p_1]\!], \ldots, \mathcal{C}[\![p_n]\!]) \\
\mathcal{C}[\![e]\!] & \stackrel{\mathrm{def}}{=} e \text{ with } \mathcal{C} \text{ applied to all expressions in e}
\end{array}
$$

Figure 4.21: Calling functions and lambdas after closure conversion

in conjunction with the stage that follows it, is to convert anonymous function objects (also called lambdas) to named top-level functions, i.e. `define`-expressions. This is necessary when compiling languages that allow functions as first-class objects, such as LISP, to languages that do not have that property, such as C.

A first naive approach to solving this problem can be illustrated by the code snippet given in Figure 4.22.

```
1 def return_n_squarer():
2     return lambda x: x * x
```

Figure 4.22: A function that returns the square function

When called, the function `return_n_squarer` returns another function that takes one parameter `x` and returns x^2. In trivial cases such as this one, it is possible to give a name to the anonymous lambda and immediately lift it to the top level. The resulting code is given in Figure 4.23. It compiles and produces the same result as the original version.

```
1 def func_0(x):
2     x * x
3
4 def return_n_squarer():
5     return func_0
```

Figure 4.23: Moving the square function out of `return_n_squarer`

We will now consider a slightly more complex function that needs to be closure converted, shown in Figure 4.24.

```
1 def get_n_adder(n):
2     r = lambda x: x + n
3     return r
```

Figure 4.24: Another function that needs to be closure converted

This function takes a parameter `n` and returns a function that takes its own parameter `x` and returns $x + n$. `get_n_adder(1)` returns the successor function, `get_n_adder(7)` returns a function that adds 7 to its parameter and returns the sum.

If we try the naive approach that we used in the first example we encounter a problem that did not arise earlier. The code that we would obtain is shown in Figure 4.25.

This example shows that function objects consist of more than just the code they define. They can also access variables of the context within

```
1 def func_1(x):
2     x + n   # Error: 'n' is not defined
3
4 def get_n_adder(n):
5     r = func_1
6     return r
```

Figure 4.25: Trying to move the lambda out of `get_n_adder`

which they are defined. This combination of code and environment is commonly called a "closure". Separating the function from its context without taking care of these variable accesses will not produce valid code. Instead we need to make use of an idea that gives this stage its name: Closure conversion.

Before we explain closure conversion in detail, it is necessary to define two concepts that we will make use of in our explanation: free and bound variables. A variable used in a function is called free, when it is not declared inside that function but instead comes from the environment. Conversely, a variable inside a function is bound, when it is declared within the function. These ideas can be explain very concisely using the lambda-calculus. Consider the simple term $M = \lambda x.(x\ y)$. The variable x is bound in M, because it is a parameter to the abstraction. y, on the other hand, is a free variable. Figure 4.19 shows a function FV that returns the free variables of all LJSP expressions.

The goal of closure conversion is to turn all free variables in the function into bound variables. The way this is achieved is by adding an additional parameter to the lambda, usually an array or dictionary, that contains the values of all previously free variables. Before the body of the lambda is executed, these variables are then bound by assigning them to the value that the environment-variable gives for them. The equation that shows this transformation is given in Figure 4.20.

If we bring the lambda from the `get_n_adder` example given in Figure 4.24 into this form, we obtain the code shown in Figure 4.26.

```
1 lambda env_0, x:
2     n = env_0[0]   # make n bound
3     x + n
```

Figure 4.26: The lambda with all variables bound

Apart from the lambda itself, both the code that defines it and the code that calls it must be adjusted. We will first describe changes to be made to the definitions of lambdas before describing how calling them changes.

After closure conversion is complete, the code no longer defines plain

lambdas. Instead it defines closures, that are made up of lambdas (the code) and an environment variable (the data). The layout and creation of this data structure is implementation dependent and will be described in detail in the Implementation Chapter. In the example below, we will make use of two functions called `make_closure` and `make_lambda` that are implementation independent.

Using these functions and free variable assignment as shown in the last example, the `get_n_adder` example from the beginning of this chapter would get converted to the code given in Figure 4.27. The code creates a closure by explicitly naming and binding free variables, instead of implicitly using variables from the environment. The code of this closure can now be moved into a separate function, which is precisely what the next compilation stage does.

```
1 def get_n_adder(n):
2     r = make_closure(lambda env_0, x:
3                          n = env_0[0]
4                          x + n,
5                      make_env(n))
6     return r
```

Figure 4.27: Creating a closure

Whereas the return type of `get_n_adder` was a lambda that could be called directly, it is now a closure which requires slightly more work to be executed. Assuming the definition of `get_n_adder` given in the last example, Figure 4.28 shows how to call a closure. In Figure 4.21 we give two equations that show the transformations that have to be made to the code.

```
1 c = get_n_adder(1)
2
3 # To get successor of 5:
4 c.code(c.env, 5)
```

Figure 4.28: Calling a closure

In [9], Paul Graham draws an interesting parallel between closures and continuations:

> Continuations can be understood as a generalization of closures. A closure is a function plus pointers to the lexical variables visible at the time it was created. A continuation is a function plus a pointer to the whole stack pending at the time it was created. When a continuation is evaluated,

it returns a value using its own copy of the stack ignoring the current one.

4.2.7 Hoisting

This is again a relatively simple stage that works in conjunction with the last stage, closure conversion. Hoisting takes closure converted lambdas and converts them from anonymous functions to named, top-level functions. Because after closure conversion, all variables used in these anonymous functions are bound, this is a very simple process.

The last definition of `get_n_adder` given in the previous chapter would change to the code shown in Figure 4.29.

```
1 def func_0(env_0, x):
2     n = env_0[0]
3     x + n,
4
5 def get_n_adder(n):
6     r = make-hl(func_0, make_env(n))
7     return r
```

Figure 4.29: Hoisting the code out of the closure

Hoisting changes the name of the function used to create closures from `make-closure` to `make-hl`. The reason that a new function is needed is that the type of the first parameter changes from a lambda to the name of a function (`func_0` in the example giving in Figure 4.29).

4.2.8 Conversion to IR (Intermediate Representation)

This stage takes the LJSP Abstract Syntax Tree that the hoisting stage generates and converts it into the Intermediate Representation of the LJSP compiler. While not small in size, this stage is of limited complexity, as its work mainly consists of converting AST objects from LJSP AST classes to IR AST classes. One significant change this stage does make to the code is that it wraps the expression that LJSP programs can have besides function definitions in its own function called `expression`.

After passing through this stage, the code is in a sequential style consisting of a list of statements. This is opposed to the style of LJSP, where expressions are arranged as a tree and an expression contains all its sub-expressions. For example, let-expressions of the form `let i=v in ...` get converted to the imperative equivalent of `i=v; ...`.

4.2.9 Redundant Assignment Removal

This is an optimisation stage that removes instructions from the IR Abstract Syntax Tree without affecting the meaning of the program. It

is based on the observation that every variable only gets assigned to once. After being assigned a value, a variable can occur many times on the right hand side of an assignment, but never again on the left hand side.

This stage scans the code for simple assignments of the form $x = y$ that assign one variable to another without any additional computations. It then replaces every occurrence of x by y in the code following the assignment before removing the assignment entirely. To see that this does not change the result of the program being converted, consider the example shown in Figure 4.30.

```
1 x = 3
2 y = x
3 z = y * y
```

Figure 4.30: Example of a redundant assignment

After redundant assignment removal, the second statement will have been removed and the code will be changed to the code shown in Figure 4.31, which still evaluates to the same value as before.

```
1 x = 3
2 # Removed assignment was here
3 z = x * x   # y has been replaced by x
```

Figure 4.31: After removing the redundant assignment

4.2.10 Conversion to asm.js

This stage converts the IR code obtained after the intermediate stages to asm.js code. Because the Intermediate Representation (IR) of the LJSP compiler was designed to be similar enough to asm.js to be easily compilable, most of the work that used to be done in this stage could be moved to 'Conversion to IR'.

One of the main differences between IR and asm.js code is memory management, which is explicit in asm.js. The memory model of the asm.js code that the compiler generates will be described in detail in the Implementation Chapter.

This stage also makes a number of smaller changes to the code such as the addition of return statements at the end of functions as well as computing and adding function tables that make it possible to save functions in variables and call these variables later.

4.2.11 Conversion to C

Like 'Conversion to IR', this is again a stage that changes the types of the objects in the Abstract Syntax Tree. Unlike the previous stage, this is one of the most complex stages of the entire compiler. This is due to the fact that C is a statically typed language with explicit memory management, whereas the Intermediate Representation includes no type information or memory management. This, in turn, means that the 'Conversion to C' stage needs to derive the type of every variable that appears in the code.

It does so by examining the ways in which the variable gets used. Consider the IR code shown in Figure 4.32.

```
1 x = some_function()
2 y = x + 2.0
```

Figure 4.32: IR code without type information

Even if we do not know the type signature of `some_function`, it is clear from this snippet that variable `x` is of type double and can not hold a function pointer, as the addition `x + 2.0` is only defined for operands of type double. Similarly, in the example shown in Figure 4.33, the compiler can derive that variable `a` holds a function object.

```
1 function f(a) {
2     x = 3
3     a(x, 4)
4 }
```

Figure 4.33: The type of `a` can be derived as well

The types used in the C code that the LJSP compiler outputs will be described in detail in the Implementation Chapter.

4.2.12 Conversion to Emscripten C

This stage makes a number of manipulations to the C code generated by the previous stage to make it suitable to be processed and compiled by Emscripten. These changes include:

- Removing the `expression` function introduced by 'Conversion to IR'
- For all functions that end in `_copy`[5], this stage adds another copy with an additional suffix of `_call_by_value`. This change is due

[5]These are the ones created after CPS-translation

to the type of the function's parameters and will be explained in detail when covering types in the Implementation Chapter

- It changes memory allocation to use a custom memory manager. This topic will also be dealt with extensively in the next chapter.

4.2.13 Conversion to LLVM IR

This stage takes as its input the C AST generated by the 'Conversion to C' stage. The reason for this is that LLVM IR code is very close in structure to C. This stage makes use of the typing information derived in the last stage.

As LLVM IR is a low level language, this stage consists of expanding each C statement to multiple LLVM IR statements. It makes no structural change to the code, but introduces a large number of new, temporary variables.

The reason these temporary variables are necessary is because LLVM IR adds another layer of pointers to the code: Local variables in LLVM IR are mapped to registers and what were local variables in C become addresses in memory that need to be loaded when needed. Because the C code generated by the LJSP compiler already uses pointers extensively, this sometimes leads to triple pointers (`void***`) in the resulting LLVM IR.

The following small example illustrates this. Consider the line of C code shown in Figure 4.34.

```
1 a = (void**)b;   // b is of type void*
```

Figure 4.34: A type cast in C

This stage converts this line of C into the three line of LLVM IR given in Figure 4.35. In LLVM IR, local variables start with a `%` sign.

```
1 %t1 = load i8** %b
2 %t2 = bitcast i8* %t1 to i8**
3 store i8** %t2, i8*** %a
```

Figure 4.35: The same type cast in LLVM IR

Line 1 loads a pointer to b into variable `%t1`. Note that `%b` is of type `i8**` which is the equivalent of `void**` while in the C code it was of type `void*`. The reason for this is that the variable `%b` holds a pointer to the value of what, in C code, was simply `b`. This value is written to `%t1` This illustrates the additional layer of pointers mentioned above.

Line 2 casts `%t1` and saves the result in `%t2`. Typecasts in LLVM IR are accomplished with the `bitcast` instruction.

Line 3 stores the result of the typecast in the memory that variable `%a` points to. Again, compare `i8***` in LLVM IR to `void**` in C.

4.2.14 Conversion to Numbered LLVM IR

This last stage modifies the LLVM IR code generated by the previous stage so that variables introduced when converting to LLVM IR get renamed to sequential numbers, i.e. `%0`, `%1`, `%2`, ... This is the syntax that LLVM IR uses for unnamed temporaries that do not get stored in memory.

This stage exists for aesthetic reasons only, as the code it outputs is easier to read than the code that the previous stage generates, especially if optimisations are run on the LLVM IR with a command such as `opt -std-compile-opts -S <file>`. This is due to the fact that when rearranging and changing instructions during optimisation, LLVM can ignore unnamed temporaries and simply reintroduce a new set of sequential variables as necessary.

4.3 Design of the Ray Tracer

After giving a brief sketch of the idea behind ray tracing in the Background Chapter, this section will explain in detail the capabilities of the ray tracer included in LJSP and the mathematical theory behind the calculations necessary for ray tracing.

4.3.1 Features

The ray tracer used to test the LJSP compiler includes the following features:

- An arbitrary number of objects (planes and spheres) can be added to the 3D scene
- Objects can have an arbitrary colour
- Objects can be made reflective or diffuse
- The number of times rays get reflected by reflective objects is adjustable
- Both ambient light and spot lights illuminate the scene
- Shading[6] is computed based on the angle from the surface normal to the light source

[6] Varying intensities of light. Visible for example on the floor of the room in the example screen shot in Figure 4.36.

- The image gets antialiased by downsampling[7] the result

These features were chosen to make the ray tracer complex enough so that rendering an image would take a substantial amount of time. This was important to get reliable data when using the renderer as benchmark.

Figure 4.36 shows a rendering of a 3D scene that showcases these features.

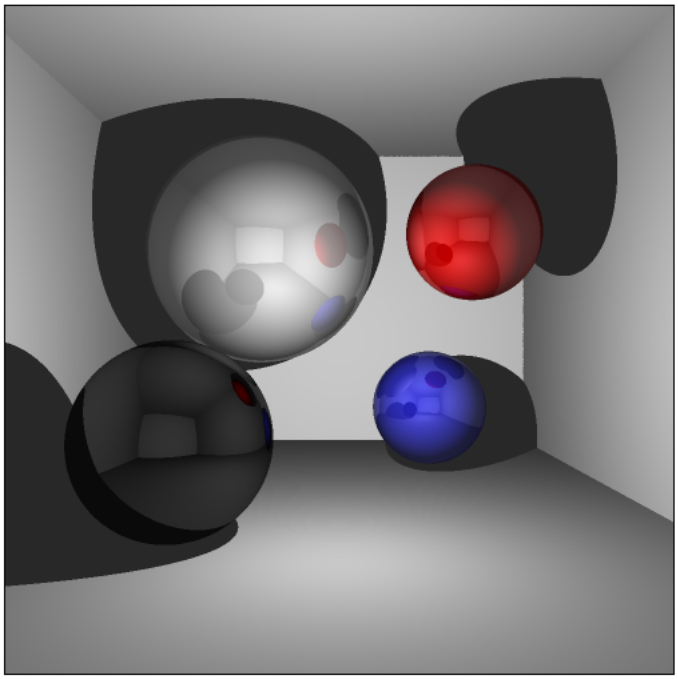

Figure 4.36: Ray traced 3D scene

4.3.2 Mathematical Underpinnings

The main objects of computation of the ray tracer are rays. A ray is given by a position and a normalised direction vector: $r = \langle \vec{p}, \vec{d} \rangle$. It is important to note that rays are unlike lines in that they start at position $\vec{p}$ and only extend in the direction given by $\vec{d}$ whereas lines extend to infinity in both directions.

The ray tracer further includes spheres given by a position and a radius: $s = \langle \vec{p}, r \rangle$. It also includes planes given by its normal and the distance from the origin $p = \langle \vec{n}, d \rangle$ and spotlights simply given by a position vector $l = \vec{p}$.

[7] This refers to smoothing rugged edges in the scene by first rendering at four times the size of the desired image (width*2 x height*2) before downsizing the result by a factor of four.

When tracing a ray r, the first step is to find the closest object that intersects the path of r, if such an object exists. To accomplish this it is necessary to compute the intersection point of r with all spheres and planes. We will first describe this computation for a ray and a plane before explaining the more complex computation for a ray and a sphere.

Given the ray $r = \langle \vec{p}, \vec{d_r} \rangle$ and the plane $p = \langle \vec{n}, d_p \rangle$, if the dot product a of the direction vector of the ray and the normal of the plane $\vec{d_r} \cdot \vec{n}$ is equal to 0, the ray runs parallel to the plane and does not intersect the plane[8]. If $a \neq 0$, we treat the ray as a line and calculate t, a scalar with the property that $\vec{p} + t * \vec{d_r}$ gives the intersection point of ray and plane. This scalar is given by $-d_p - (\vec{p} \cdot \vec{n})/(\vec{d_r} \cdot \vec{n})$. If $t < 0$, the plane lies behind the origin of the ray and there is no intersection point of r and p.

Calculating the intersection point and t of a ray $r = \langle \vec{p_r}, \vec{d_r} \rangle$ and a sphere $s = \langle \vec{p_s}, r_s \rangle$ is somewhat more complex. The calculation used in the LJSP ray tracer described in the following is mostly based on [18].

Because the calculations will be made in the object space of s, which is the 3D space with its origin at $\vec{p_s}$, it is necessary to first translate both r and s to this object space. This is very simple in the case of the sphere: Its position is simply set to $(0, 0, 0)$ as that is how the object space of s is defined. To translate r to the object space of s, it is necessary to subtract $\vec{p_s}$ from $\vec{p_r}$. The direction vector of the ray is independent of the position of the ray and does not need to be changed. To make the computation efficient, three temporary variables are now computed that are used repeatedly in the remaining computations:

$$
\begin{aligned}
A &= d_r \cdot d_r \\
B &= 2 \cdot (d_r \cdot p_r) \\
C &= p_r \cdot p_r - r_s \cdot r_s \\
D &= B \cdot B - 4 \cdot A \cdot C
\end{aligned}
$$

Calculating the intersection point means solving a quadratic equation of which the variable D is the discriminant. If $D < 0$, the equation has no solution and r and s do not intersect. If $D \geq 0$, two more variables are computed:

$$
t_0 = \frac{-B - \sqrt{D}}{2A} \qquad t_1 = \frac{-B + \sqrt{D}}{2A}
$$

If both t_0 and t_1 are less than 0, there is no intersection point. If one of these two values is less than 0, the other is returned as t. Otherwise the smaller one is returned.

Returning to the process of tracing a ray r, after calculating the closest intersection point of the ray with the objects in the scene, the

[8] Experimental results proved that it is not necessary for successful rendering to consider the case where r lies on the plane.

tracing function determines if this intersection point is illuminated by the spot light or if the light is obscured by some other object. It does so by constructing a new ray $r_n = \langle \vec{i}, \vec{d_l} \rangle$ with its position vector $\vec{i}$ being the intersection point and its direction vector $\vec{d_l}$ being a vector going from the intersection point to the light. Again, the closest intersection point of this ray with the objects in the scene is calculated. If there is none, or if it lies behind the light as seen from the ray, the original intersection point is visible from the light, otherwise it is not.

If the point is illuminated by the light, the tracing function next calculates proper shading. If the light from the spot light hits the object that r intersects with at an angle, the intensity of the light is less than when it hits the object head-on. The intensity of the light reaching the intersection point is calculated by forming the dot product of the objects normal at the intersection point and the direction to the light calculated earlier as $\vec{d_l}$. To imitate ambient light, this value is set to 0.2, if it is less than that.

If the object hit by the ray r is reflective, the tracing function recurses by tracing the reflection ray before returning a combination of the result of the recursion with the colour of the object. Otherwise it returns the objects colour illuminated with the light calculated in the last section.

5 Implementation

This chapter will describe the implementation details of the LJSP compiler. It builds on and assumes the previous chapter which describes the compiler from a theoretical perspective.

We will first give an overview of the code that the LJSP compiler generates, before explaining the structure of the compiler, the implementation of its stages and code emission in one section each. The reason for explaining the generated code first is that many of the internals of the compiler follow the requirements of the code that it generates.

Following this, we give a detailed overview of the implementation of the ray tracer.

5.1 Details of the Generated Code

This section will give a detailed description of the code that the LJSP compiler outputs. It will first describe the asm.js modules generated by the compiler before explaining the C and LLVM code it generates.

5.1.1 Asm.js Modules

Memory model

Asm.js offers three ways of saving values to memory: local variables, global variables and a heap in the form of an `ArrayBuffer`. In the modules generated by the compiler, all these variables are of type double and get casted to integers only inside expressions when necessary.

Local variables are variables local to the function they are declared in. All local variables in a function have to be declared at the top of the function. When the compiler converts an asm.js AST to code, it scans through all functions to find local variables that are used inside the function to create the appropriate declarations. Because all local variables are of type double, all these declarations are of the form `var n = 0.0;`, because the way to declare a variable as double in asm.js is to initialise it with a floating point value.

Every asm.js module generated by the LJSP compiler has exactly four global variables, shown in Figure 5.1.

```
1 var sqrt = stdlib.Math.sqrt;
2 var floor = stdlib.Math.floor;
3 var D32 = new stdlib.Float32Array(heap);
4 var mem_top = 0.0;
```

Figure 5.1: Global variables of asm.js modules

The first two global variables import two functions to the module: The square root function, which is used to implement the `sqrt` operator, and the floor function, which rounds a floating point number down to the previous integer. This function is used in the code to convert variables from float to integer.

Global variable `D32` holds a Float32 View to the heap as explained in the Background Chapter. Because all variables in LJSP are of type double, only this one view to the heap is necessary.

The last global variable, `mem_top` is used for memory management. This leads to the next section, the memory model used for managing the heap.

After Closure Conversion, the code includes calls to the `make-env` and `make-closure` functions that, in conjunction, are used to create closures. The IR function in the example shown in Figure 5.2 is taken from a program that computes fibonacci numbers (the code was edited slightly to make it more readable). Note how the closure created in lines 6 and 7 is not called right away, but instead given as parameters to a recursive call to fib in line 8.

In asm.js code, this data structure is implemented with arrays that are saved in the `ArrayBuffer` object. Every asm.js module generated

```
function fib(cont, n) {
    if (n < 2.0) {
        cont.code(cont.env, 1.0)
    } else {
        var_3 = n - 1.0
        env = make-env(n, cont)
        closure = make-hl(func_0, env)
        fib(closure, var_3)
    }
}
```

Figure 5.2: IR code for calling a closure taken from a real example

by the LJSP compiler includes a function called `alloc` shown in Figure 5.3. This function is used to allocate memory for new arrays.

```
function alloc(size) {
    size = +size;
    var current_mem_top = 0.0;

    current_mem_top = mem_top;
    mem_top = +(mem_top + size);

    return +current_mem_top;
}
```

Figure 5.3: The `alloc` function

The `alloc` function takes the number of needed elements and returns the index to the first element of a subarray of the heap of the required size. The variable `mem_top` always holds the next element of unallocated memory.

The closure data structure is constructed by the compiler using these arrays. It consists of a pointer to an array with two elements. The first element gives the index to the function in one of the function tables that contains the code of the closure. The second element contains a pointer to another array with a variable number of elements that contains the closure's environment. Figure 5.4 shows this structure.

Because `mem_top` gets increased by every memory allocation, it is necessary to free memory somewhere in the code, if we want to avoid memory overflows. The places where this happens are the functions that end in `_copy`, introduced by the CPS-translation stage. Because these functions are the entry points to the module, they reset `mem_top` to 0 before calling the function they were copied from. This is possible because in LJSP, no state is shared between function calls.

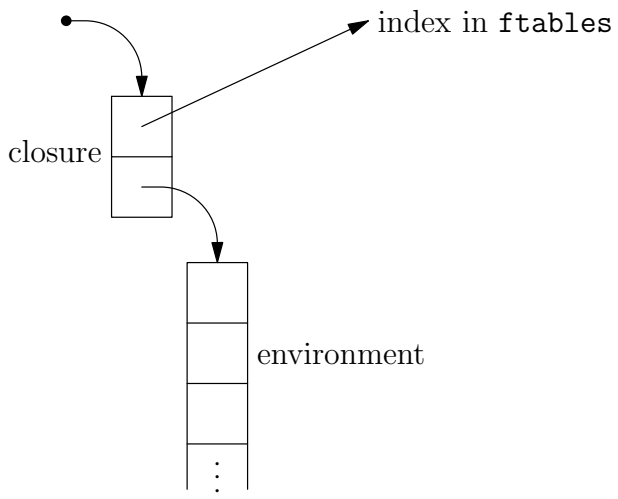

Figure 5.4: The closure data structure in asm.js code

The example shown in Figure 5.5 shows the asm.js code of the `_copy` function for the `fib` function from the example above. Detailed explanations of all statements will be given in the next section.

This example also illustrates why `_copy` functions are necessary. Memory management is internal to the module and can not be accessed by other code that calls into the asm.js module. It would therefore be impossible for the caller to create the continuation that the `fib` function takes as it first argument and that `fib_copy` creates in lines 11 to 14, because it can not call the `alloc` function to create the continuation.

Structure of the Module

Asm.js modules generated by the LJSP compiler are made up of the following elements:

- The four global variables and the `alloc` function that were described in the previous section
- A number of functions
- A number of function tables
- A return statement

An example return statement from the `gen_code.js` file included in the repository is given in Figure 5.6. Only the `_copy` functions are exported from the module to ensure that only these functions are used as entry points and the heap gets reset properly before every computation. They are, however, exported under their original name (without the suffix), which makes copied functions completely transparent to the user.

```
1  function fib_copy(n){
2  // Parameter n is of type double
3  n = +n;
4
5  // Declare vars
6  var closure = 0.0, env = 0.0;
7
8  // Reset memory
9  mem_top = 0.0;
10
11 // Create continuation closure
12 env = +alloc(+0);
13 closure = +alloc(+2);
14 D32[(~~+floor(+(closure + 0.0))|0) << 2 >> 2] =
       2.0;
15 D32[(~~+floor(+(closure + 1.0))|0) << 2 >> 2] =
       env;
16
17 // Call original function
18 return +fib(closure, n);
19 }
```

Figure 5.5: A `_copy` function

```
1  return {vectorsDotProduct: vectorsDotProduct_copy,
2      raySphereIntersectionPoint:
3      raySphereIntersectionPoint_copy};
```

Figure 5.6: The return statement of a real asm.js module

The function tables of the same program are shown in Figure 5.7. All functions in the module can be found in one of the function tables, even if they are always called by name. The name assigned to every table is made up of the string `ftable` followed by the number of parameters the functions in that table take. This is reflected in the fact that `_copy` functions can always be found in a function table with a parameter count one less than the table that contains the original function. In the example below, `vectorsDotProduct_copy` is in `ftable6` and `vectorsDotProduct` is in `ftable7`.

As mentioned in the Background Chapter, every function table has to have a size equal to a power of two. Because it is not always the case that this matches with the number of functions in the module, function tables with a length not equal to 2^x have their size increased to the next higher power of two by appending the first function in the table as many

```
1 var ftable6 = [vectorsDotProduct_copy];
2 var ftable7 = [vectorsDotProduct];
3 var ftable2 = [func_0, func_1, func_2, func_3,
4                func_4, func_0, func_0, func_0];
5 var ftable10 = [raySphereIntersectionPoint_copy];
6 var ftable11 = [raySphereIntersectionPoint];
```

Figure 5.7: Function tables in asm.js

times as necessary. An example of this is `ftable2` in the code, which had its first element copied three times to reach a size of 2^3. These copied functions are not used and only necessary to create valid asm.js code.

To conclude this section, we will describe some of the more complex statements that the compiler outputs.

While heap assignments work in principle as described above, in practice they require a number of casts and type coercions that makes them quite hard to read. The short example given in Figure 5.8 shows the creation of a closure. Except for line breaks, variable renaming and the removal of redundant parentheses, this is unedited asm.js code as generated by the compiler.

```
1 env = +alloc(+4);
2 D32[(~~+floor(+(env+0.0))|0) << 2 >> 2] = A;
3 D32[(~~+floor(+(env+(1.0))|0) << 2 >> 2] = cont_1;
4 D32[(~~+floor(+(env+2.0))|0) << 2 >> 2] = s_r;
5 D32[(~~+floor(+(env+(3.0))|0) << 2 >> 2] = var_28;
6 closure = +alloc(+2);
7 D32[(~~+floor(+(closure+0.0))|0) << 2 >> 2] = 2.0;
8 D32[(~~+floor(+(closure+1.0))|0) << 2 >> 2] = env;
```

Figure 5.8: Creating a closure in asm.js

Creating a closure entails first creating the environment and then creating the closure itself. In the code above, the environment is created in lines 1 to 5. Because it is made up of four variables, the first line allocates an array with four elements. The `+` signs in front of the call to `alloc` and many other expressions in the code are type coercions. These are explained in detail in the Background Chapter.

After allocating memory in line 1, the four variables that make up the environment (`A`, `cont_1`, `s_r` and `var_28`) are assigned to the four spaces in the array in lines 2 to 5. The computation of the index to `D32` consists of many different parts that we will now go through in detail.

The expression `... << 2 >> 2` on the very outside does not change the result of what comes before it, but is necessary because the asm.js

spec requires a right shift by log_2 4 when accessing the heap through a view with a datatype of length four bytes such as Float32Array. Because this right shift on its own would change the value of the index we compute, we add a left shift by the same number of bits in front of the right shift. This problem could also be solved by changing array index computation so that the right shift alone computes the correct result, but in practice, this back and forth shifting turned out to have a negligible impact on performance.

The remaining expression is a call to the floor function of the following, somewhat odd looking, form: `~~+floor(...)|0`. This is the asm.js way of casting a double to an integer. This is necessary, because all variables in the module are of type double, but array indices have to be of type int. This typecasting could be avoided by making parameters and variables that are used as indices to access the heap of type int instead of double. Because all functions in one function table have to have the same type signature, however, this would require complex changes to the compiler and is left as future work. Inside that cast, the variable holding the base of the environment (`env`) is added to offsets going from 0 to 3.

After the environment variable has been constructed, the rest of the example creates the closure itself. As mentioned above, closure variables are always arrays of length two. In line 7, the index to the function in its function table is assigned to the element at offset 0. Line 8 assigns the environment variable to the element at offset 1.

While the number assigned in line 7 seems, at first sight, to give insufficient information to identify the function to be called as there is more than one function table in the program, it does, in fact, suffice to uniquely determine the correct function table the index belongs to. The reason for this is that later in the code when this closure is called, the number of parameters it is called with, is known at compile time. The name of the function table is then given by the expression `"ftable" + (params.size + 1)`, because function tables are named after the number of parameters that the functions they contain take.

5.1.2 C Code

Memory model

Unlike asm.js, which uses global variables, the C code only uses local variables and memory on the heap allocated with `malloc`. While in asm.js modules, functions take parameters of type double, in the C code, all parameters are pointers of type `void*`. The reason for this is that both pointers to double values and arrays are passed to the function through its parameters. Based on the way the parameters are used inside the function, they are cast to the appropriate type and dereferenced.

This means that it is not possible to pass static double values as

arguments to function calls. Instead, it is necessary to allocate memory on the heap, write the value to that allocated space in memory and pass a pointer to it as parameter to the function call. An example of this is given in Figure 5.9.

```
1 const_0 = (double*)malloc(sizeof(double));
2 *const_0 = 1000000.0;
3 func_pointer_0(env_param_0, const_0);
```

Figure 5.9: Calling a function with a static value as parameter

To make it possible to pass local variables as arguments to functions, they are declared to be of type `double*`, assigned memory allocated with `malloc` and dereferenced when read from or written to. Simply passing a reference in a function call would lead to a crash as local variables are allocated on the stack and destroyed when the function that they are local to returns.

Creating closures is done similarly to the method used for asm.js. Figure 5.10 shows the C version of the code given in Figure 5.8.

```
1 env = (void**)malloc(sizeof(void*) * 4);
2 env[0] = A;
3 env[1] = cont_1;
4 env[2] = s_r;
5 env[3] = var_28;
6 closure = (void**)malloc(sizeof(void*) * 2);
7 closure[0] = &func_2;
8 closure[1] = env;
```

Figure 5.10: Creating a closure in C

The main difference between asm.js and C is that in the C code above, `closure[0]` is assigned a function pointer to the function to be called as there are no function tables used in the C code. Calling a closure is slightly more complex than creating it. Figure 5.11 shows how a closure is called in C.

Lines 11 to 15 are casts and assignments to obtain the function pointer and the environment variable from the closure. Line 17 is the function call itself.

Figure 5.12 shows the closure data structure used in the C code. It is quite similar to the asm.js data structure shown in Figure 5.4. The first difference between the two is that in the generated C code, the first element of the closure data structure points directly to a function as opposed to giving an index in a function table. The second difference is that the environment array is made up of pointers to double values, whereas in asm.js the environment array contains the values themselves.

```
1  // Variable declarations
2  void** casted_hl_var_0;
3  void* uncast_func_pointer_0;
4  void *(*func_pointer_0)(void*,void*);
5  void* env_param_0;
6  double* const_0;
7
8  ...
9
10 // Calling cont_1
11 casted_hl_var_0 = (void**)cont_1;
12 uncast_func_pointer_0 = casted_hl_var_0[0];
13 func_pointer_0 =
14     (void* (*)(void*,void*))uncast_func_pointer_0;
15 env_param_0 = casted_hl_var_0[1];
16 ...
17 ret_val_3 = func_pointer_0(env_param_0, const_0);
```

Figure 5.11: Calling a closure in C

Although this structure is the same for all closures used in the generated C code, it is always constructed ad-hoc with calls to `malloc`, casts and array accesses instead of using a C-style `struct`. While this could change in the future, readability of the code that the LJSP compiler generates was not a main focus during its development.

Before concluding this section, there is one remaining aspect of memory management in C worth mentioning. The C code generated by the 'Conversion to C' stage is designed to run as an executable and terminate after computing and outputting the result of the expression of the program. C code generated by 'Conversion to Emscripten C' however does not terminate and has its functions called many times by the ray tracer. This makes it necessary to free memory allocated by the functions if an overflow is to be avoided. The way this is achieved is by using a custom memory manager. This memory manager can be found in the file `jalloc.c` and includes two functions: `jalloc`, which replaces `malloc` and `free_all`, which frees all the memory allocated with `jalloc`. Like in the asm.js modules, the entry point functions ending in `_copy` reset the memory before calling the functions they were copied from.

`jalloc` is implemented using a linked list. Every block of memory requested is added as a node to the list before being returned. The `free_all` function goes through this list from head to tail and calls `free` on all the memory saved in it.

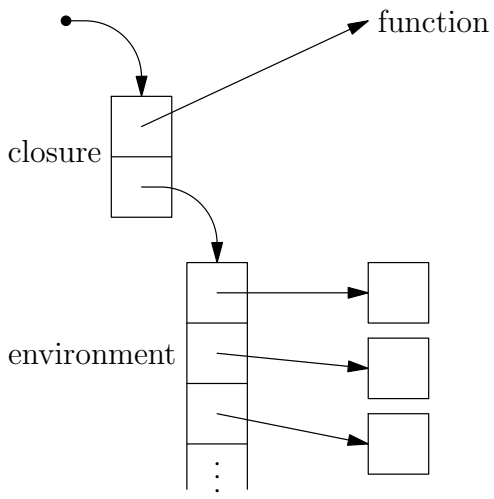

Figure 5.12: The closure data structure in C code

Structure of the Module

The C code generated by the LJSP compiler is made up of three sections:

- Preprocessor directives
- Function declarations
- Functions

The preprocessor directives are the same for all programs. They are shown in Figure 5.13.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 #define min(x,y) ((x)<(y)?(x):(y))
6 #define max(x,y) ((x)>(y)?(x):(y))
```

Figure 5.13: Preprocessor direcives used in compiled C code

`stdio.h` is included so that `printf` can be used to print the result of the program to the command line, `stdlib.h` provides the `malloc` function used to allocate memory and including `math.h` is necessary to be able to use the `sqrt` function that calculates square roots. The `min` and `max` functions are implemented using macros and the tertiary operator.

The function declarations include all functions in the file except `main`, so that they can call each other and do not need to be defined in any particular order.

The function block includes all the function that are present in IR code before the 'Conversion to C' stage. It further includes a `main` function, shown in Figure 5.14, which calls the `expression` function that contains the expression of the program and prints its result to the console.

```
1 int main(int argc, char **argv) {
2     double *r = expression();
3     printf("%f\n", *r);
4 }
```

Figure 5.14: The main function used in compiled C code

The functions themselves are split into a set of variable declarations and a list of statements. Generally, the statements are quite similar to IR code. One noteworthy exception are primitive operations which are much more complex, as these operations are performed on void pointers and the result must be written to a double pointer. The example given in Figure 5.15 shows the result of converting the IR statement `var_2 + var_4` to C:

```
1 prim_op_p_1 = (double*)var_2;
2 prim_op_v_1 = *prim_op_p_1;
3 prim_op_p_2 = (double*)var_4;
4 prim_op_v_2 = *prim_op_p_2;
5 var_1 = (double*)malloc(sizeof(double) * 1);
6 *var_1 = prim_op_v_1+prim_op_v_2;
```

Figure 5.15: A primitive operation in C

Both operands are first cast from `void*` to `double*`, before getting dereferenced. The result of these dereference operations, i.e. the values to be added are then saved in two variables (`prim_op_v_1` and `prim_op_v_2` in the example). Before the addition takes place, memory is allocated to hold the result and assigned to a `double*`. In the last line, the sum of the two operands is then written to the memory allocated in the previous line.

As mentioned in the section on the memory layout of the C code generated by the compiler, there are a few small differences between the code that the 'Conversion to C' stage and the 'Conversion to Emscripten C' stage emit. 'Conversion to Emscripten C' removes the `expression` and `main` functions. They are unnecessary, because Emscripten compiles the C code to an asm.js module which will have its functions called by

some other piece of code. 'Conversion to Emscripten C' further replaces calls to `malloc` by calls to `jalloc` and adds a call to `free_all` to `_copy` functions. It further and adds the line `#include "jalloc.c"` to the preprocessor block to make these two functions available in the code. The last change it makes is copying every function that ends in `_copy` to a new function with the suffix `_call_by_value`. These new functions take the same number of parameters as the original functions, but their parameters are of type `double` instead of `double*`. This is necessary for the same reason the `_copy` functions are necessary: The calling code has no access to the memory handling of the module that Emscripten generates and can not allocate any memory that the double pointers could point to. This is instead done by the new `_call_by_value` functions.

5.2 Structure of the Compiler

The LJSP compiler itself is written in Scala and spread out over a number of files. The code of the compiler follows a logical structure that separates the code into modules and makes it easy to find related modules. The necessity for a clear structure became evident quickly during the development of the compiler, as it was often necessary to edit multiple parts of the code simultaneously.

The code is structured as follows:

- The entry point of the compiler can be found in `main.scala`. This files contains functionality to parse the command-line arguments that the LJSP compiler gets called with. Based on these arguments, it calls the appropriate methods to achieve the desired results.

- All classes that make up the Abstract Syntax tree can be found in `AST.scala`. We decided against splitting up this file into a separate file for each language used in the LJSP compiler (LJSP, IR, C, etc.) as that would have lead to unnecessary fragmentation of the code.

- The compilation stages that make up the LJSP compiler can be found in files with file names starting with a number, e.g. `03_-cps_translation.scala`, optionally followed by a letter that indicates which stages precede and follow it. These files all contain exactly one object with a name equal to the name of the file without the number (`object cps_translation` in the example of CPS-translation) which in turn contains all the methods used in conversion.

- The functions that convert ASTs back to specific languages can be found in files that have file names starting with `code_generation_-`. There are five such files in total, one each for every programming

language that the LJSP compiler converts to or from: LJSP, IR, asm.js, C and LLVM IR.

- The file `run_tests.py` contains the testing framework described further down in the Testing Chapter. All other files related to testing can be found in the `test/` folder.
- The ray tracer and all files related to it reside in the `ray_tracer/` folder.
- `jalloc.c` contains the custom memory manager used in code generated by the 'Conversion to Emscripten C' stage.
- The file `util.scala` contains a small number of utility functions used throughout the rest of the code.

5.3 Implementation of Various Stages

This section will describe the implementation details of most stages. The stages not mentioned in this section follow the theoretical description given in the Design Chapter so closely that there is nothing that stands out enough to warrant inclusion in this chapter.

5.3.1 Parsing

Parsing is implemented using the Scala standard library. This makes the code terse and readable. Because LISP is a very easy language to parse, writing a custom parser for LJSP would not have yielded very interesting results. We therefore decided against implementing our own parsing library. The parser clearly reflects the grammar of LJSP in code form. The grammar rule for a `define` is given in the Specification Chapter as follows:

⟨*define*⟩ ::= '`(define (`' ⟨*ident*⟩ ⟨*params*⟩ '`)`' ⟨*expr*⟩ '`)`'

⟨*params*⟩ ::= ⟨*ident*⟩ ⟨*params*⟩ | ϵ

This rule is still visible in the code. Note that for the Scala parser combinators, `rep` has the meaning of "Zero or more", which made the additional *params* rule unnecessary in the code. The parsing rule for `define`s is shown in Figure 5.16.

5.3.2 CPS-Translation

The code for CPS-translation closely resembles the equations given in the Design Chapter. One peculiarity of the CPS-translation function is that it comes in two versions taking two different types of continuations.

```
def define: Parser[SDefine] =
"(" ~> "define" ~> "(" ~>
    identifier ~ rep(identifier) ~ ")" ~
    expression <~ ")" ^^ {
        case name~params~")"~e =>
            SDefine(name, params, e)
}
```

Figure 5.16: The parsing rule for `define`s

```
def cps_trans(e: SExp, k: SExp => SExp) : SExp
def cps_tail_trans(e: SExp, c: SExp) : SExp
```

Figure 5.17: The two functions that CPS-translate LJSP expressions

The first line of the definitions of both functions are shown in Figure 5.17.

While both functions take the expression to be CPS-translated as their first parameter, the first function takes a function that maps an `SExp` to an `SExp` as its second parameter while the second function takes the continuation in a simple `SExp`. In effect, the first function takes a part of the Abstract syntax tree that the result of CPS-Translating `e` is to be inserted into, while the second function takes an expression that contains the continuation in the context of the program to be compiled.

To further illustrate this concept, consider the translation of `SIf` statements used in earlier versions of the compiler shown in Figure 5.18.

```
case SIf(e1, e2, e3) => {
    val c = SIdn(fresh("var"))
    val p = SIdn(fresh("var"))

    cps_trans(e1, (ce1: SExp) =>
        SLet(c, SLambda(List(p), k(p)),
            SIf(ce1, cps_tail_trans(e2, c),
                     cps_tail_trans(e3, c))))}
```

Figure 5.18: `if`-tranlsation rule used in early versions

In the mathematical notation used in the Design Chapter, this can be expressed as:

$$\mathcal{K}[\![(\text{if } e_1\ e_2\ e_3)]\!]k \stackrel{\text{def}}{=} \mathcal{K}[\![e_1]\!]\lambda x.(\text{let } c\ =\ \lambda p.k(p) \text{ in } (\text{if } x\ \mathcal{K}[\![e_2]\!]c\ \mathcal{K}[\![e_3]\!]c)$$

This version of the CPS-translation of if-expressions creates a lambda ($\lambda p.k(p)$ in the equation above) and CPS-translates e_2 and e_3 with this lambda as their continuation. Because this is a lambda in the context of the program, not in the context of the compiler, it requires the additional `cps_tail_trans` function.

In contrast, the CPS-translation of if-expressions in the current version of the compiler is shown in Figure 5.19.

```
1 case SIf(e1, e2, e3) => {
2     cps_trans(e1, (ce1: SExp) =>
3         SIf(ce1, cps_trans(e2, k),
4                  cps_trans(e3, k)))}
```

Figure 5.19: `if`-tranlsation rule used in current versions

Or expressed in mathematical notation:

$$\mathcal{K}[\![(\text{if } e_1\ e_2\ e_3)]\!]k \stackrel{\text{def}}{=} \mathcal{K}[\![e_1]\!]\lambda x.(\text{if } x\ \mathcal{K}[\![e_2]\!]k\ \mathcal{K}[\![e_3]\!]k)$$

This version applies the continuation function k twice. This avoids the introduction of the `lambda` necessary in the previous version, but comes at the cost of code duplication: The AST subtree saved in k will be used in both branches of the if-expression. We will return to this trade off in the Evaluation Chapter where the reasoning behind the choice for the second version will be explained. It is, however, important to mention here that `cps_tail_trans` can not be avoided completely: When CPS-translating `SDefine` and `SLambda`-expressions that receive their continuations in a parameter, it is necessary to use `cps_tail_trans`, as these parameters are always on the level of the program, not on the one of the compiler.

5.3.3 Hoisting

The hoisting stage uses an internal class named `HoistedExpression`. This class is used to accumulate hoisted functions when recursively traversing the Abstract Syntax Tree.

In earlier versions of the compiler, this was implemented using Scala tuples, but using a dedicated class was found to improve readability.

5.3.4 Redundant Assignment Removal

This stage is implemented in a purely functional way. The core of redundant assignment removal is given in Scala-like pseudocode in Figure 5.20.

```
1  def rar(statements) = statements match {
2    case Nil => Nil
3    case s::sts => {
4      if s is of type var1=var2:
5        sts_ = rename var1 to var2 in sts
6        return rar(sts_)
7      else:
8        return s :: rar(sts)
9    }
10 }
```

Figure 5.20: Removing redundant assignments

This function calls itself recursively for each statement in the list, discarding it if it is a redundant assignment and renaming the variable on the left hand side in those cases.

5.3.5 Conversion to asm.js

Besides the rather simple task of converting IR AST classes to their equivalent in the asm.js AST class hierarchy (`IStaticValue` to `AStaticValue`, `IFunctionCallByName` to `AFunctionCallByName`, etc.), this stage makes a number of other major changes to the code.

The first is the generation of function tables. The part of the code that accomplishes this is given in Figure 5.21, again slightly edited.

```
1  val fs_map = functions.groupBy(_.params.size)
2
3  val fnames_map = fs_map.map{ case (size, fs) =>
4    ("ftable"+size.toString, fs.map{_.name})}
5
6  val fnames_map_pow_2 = fnames_map.map{
7    case (ftable, fnames) => {
8    val size_difference =
9      find_next_power_of_2(fnames.size) - fnames.
         size
10   (ftable, fnames ++
11     List.fill(size_difference)(fnames(0)))
12   }
13 }
```

Figure 5.21: Creating function tables

The statement in line 1 groups the functions in the `functions` list by the number of their parameters, creating a Map object that maps

parameter count to a list of function objects.

In line 3 and 4, this Map is turned into a second map, now mapping strings of the form `"ftable2"` to lists of function names.

Finally, the statement spanning from line 6 to 13 goes through every function table, represented by a key/value pair in the map, and calculates the difference in the count of function names to the next power of two. It then fills up the list with enough copies of the first function to reach that next power of two. It does so using the `fill` method of Scala's `List` class. This method takes a number `n` and an object `o` and creates a list with `n` copies of `o`.

After the function tables have been created in this way, converting a call to a variable holding the index to a function is a simple process. The code that does this is shown in Figure 5.22. Note how the correct function table name is found simply by adding the number of parameters (increased by one for the environment parameter) to the string `"ftable"`.

```
1  case IFunctionCallByVar(hl_var, params) => {
2    val converted_params = params.map{
3      convert_expression_to_asmjs(ftables, _)}
4
5    val ftable_name =
6      "ftable" + (params.size + 1).toString
7
8    AFunctionCallByIndex(ftable_name, hl_var,
9                         converted_params)
10 }
```

Figure 5.22: Converting calling a function variable to asm.js

The third part to calling functions using function tables in asm.js is finding the correct index to the function table determined in the code snipped above. This is done when expanding IR `make-hl`-expressions to asm.js code that allocates memory and creates the closure. The code that accomplishes this uses a function called `fname_to_ftable_index` that goes through all function tables and returns the index of the function in the function table it is in.

Finally, this stage introduces return statements to all functions. If the last statement in the function is an expression, that expression is returned. If, however, the last statement is an `if`-statement, a variable is created that is assigned the last expression of both if-branches and returned after the if statement[1]

[1]The simpler solution of returning inside the if-branches does not conform to the asm.js specification.

5.3.6 Conversion to C

This is one of the most complex stages in the LJSP compiler. This is reflected in its implementation, which is longer and more complex than that of other stages.

Unlike the other stages, which convert statements of one kind to statements of another kind, this stage converts IR statements to tuples of declarations and C statements. This is reflected in the type of the conversion function which returns a result of type `(List[CDeclareVar], List[CStatement])`. Deriving the correct type of the variables to be declared is one part of the work of this stage.

In the Intermediate Representation, it is not necessary to declare a variable before using it. Consider an IR statement such as `a = 3.0;`. In C, this statement gets expanded to the code shown in Figure 5.23.

```
1 double* a;
2 ...
3 a = (double*)malloc(sizeof(double) * 1);
4 *a = 3.0;
```

Figure 5.23: Converting a simple assignment to C

The implementation of the part of the code that accomplishes this transformation is given in Figure 5.24.

```
1 case IVarAssignment(idn, IStaticValue(d)) => {
2   (CDeclareVar(idn, CTDoublePointer) :: Nil,
3
4   CVarAssignment(idn,
5     Malloc(CTDoublePointer, CTDouble, 1)) ::
6   CDereferencedVarAssignment(idn,
7     CStaticValue(d)) :: Nil)
8 }
```

Figure 5.24: Converting assignments of static values to C

Line 2 of this example creates the first list in the tuple: a list with one element which contains the declaration of the variable assigned to. This variable is of type `CTDoublePointer` which is the compiler's representation of `double*`.

Lines 4 to 7 create the second list of the tuple consisting of the two statements that allocate memory and assign the static value `d` to the newly allocated memory by dereferencing the variable.

The declarations inferred from all available IR-statements are the following:

- If-statement, `if (cond) block1 else block2`: No declarations are added, but `block1` and `block2` are examined recursively
- Reading from an array, `v = a[i]`: `v` is declared to be of type `void*`
- Calling a named function, `fname(p1, ..., pn)`: All parameters are declared to be of type `double*`
- Calling a function variable, `var(p1, ..., pn)`: `var` is declared to be a function pointer with `n` parameters, all parameters are declared to be of type `double*`
- Assigning a static value, `v = 3`: `v` is declared to be of type `double*`
- Primitive operation, `v = a + b`: `idn` is declared to be of type `double*`
- Constructing an environment, `env = make-env(...)`: `env` is declared to be of type `void**`
- Constructing a closure, `hl = make-hl(...)`: `hl` is declared to be of type `void**`
- Assigning one variable to another, `v1 = v2`: `v1` is declared to be of type `double*`

5.3.7 Conversion to LLVM IR

Although the file containing this stage is the largest of all the files containing the compilation stages, this is is a simple stage in terms of the its transformation. Its main function expands every `CStatement` into a number of LLVM IR statements but it makes no structural changes to the Abstract Syntax Tree.

Many of these expansions are straightforward, such as using the LLVM IR `bitcast` instruction to cast variables from one type to another. Two, however, involve more complex expansions. The rest of this section lists and explains these two cases.

The first are `if`-statements that get expanded to the form they commonly take in assembly code that has explicit jumps. The result of expanding an `if`-statement to LLVM IR is shown in Figure 5.25. Line 1 makes a ne (not equal) comparison of the condition variable with 0 and writes the result to `%c`. Line 2 branches conditionally, jumping either to the next line or to `false_branch`. In LLVM IR, execution can not simply fall through a label. This makes it necessary to explicitly name both branches in the `br` instruction. It also makes the unconditional jump in line 8 required.

The second complex expansion is `min/max`-expansion. These expressions are expanded to `if`-statements as described above. Additionally, they make use of another instruction that can be used in conjunction

```
%c = icmp ne i32 %condition_var, 0
br i1 %c, label %true_branch, label %false_branch
true_branch:
...
br label %end_if
false_branch:
...
br label % end_if
end_if:
...
```

Figure 5.25: Structure of `if`-statements in LLVM IR

with conditional branching: the `phi` instruction. This instruction makes it possible to assign a value based on which branch of the `if`-statement was executed.

5.4 Code Generation

Generating or emitting code in text form is the reverse of parsing. It is the process of taking an Abstract Syntax Tree and converting it to the textual representation of the programming language. Because of this relationship between parsing and code generation, the latter is sometimes called "unparsing" [19].

In early versions of the LJSP compiler this was accomplished by simply adding a `toString` method to all classes that make up the AST. Because converting an expression or statement to a string involves converting all sub-expressions to strings, the `toString` method of the root object would recursively traverse the entire tree.

It later turned out to be a much better decision to move these code emitting functions to separate files, as described in the section "Structure of the Compiler". These files contain functions that work very similarly to the `toString` methods used previously in that they recursively traverse the AST, but they are separate from the definition of the class hierarchy in `AST.scala`.

5.5 Implementation Details of the Ray Tracer

This section will describe the internals of the ray tracer that is included with LJSP.

5.5.1 Files Related to Ray Tracing

All files related to ray tracing can be found in the `ray_tracer/` directory. They are:

- `ray_tracer.html`, this is the html file that loads all the JavaScript files and includes an interface that the user can use to interact with the code
- `ray_tracer.js`, this file contains the ray tracer itself
- `ljsp_code.scm`, this file contains an implementation of two functions: `raySphereIntersectionPoint` and `vectorsDotProduct` in LJSP
- The remaining JavaScript files contain different versions of the two functions defined in `ljsp_code.scm` and are explained in detail below

5.5.2 The HTML Interface

The user interacts with the ray tracer using the web page contained in `ray_tracer.html`. This html interface includes a canvas on which the rendered 3D scene will be drawn, four buttons that offer the user a choice of rendering methods as well as a way to display the time it took to render the image. Figure 5.26 shows this html interface.

Every rendering method consists of a different implementation of two of the main functions the renderer uses: `raySphereIntersectionPoint` and `vectorsDotProduct`. These functions compute the intersection point of a ray with a sphere and the dot product of two vectors. Comparing the running time of the different methods against each other offers a benchmark for each method.

There are four different rendering methods in total which correspond to the four buttons on the html page:

1. Standard JavaScript, the implementation of this method can be found in `ray_tracer.js`
2. Handwritten asm.js code, this is implemented in `handwritten_asm.js`
3. Asm.js code generated by the LJSP compiler by compiling `ljsp_code.scm`, this can be found in the `gen_code.js`
4. Asm.js code compiled by Emscripten from C code the LJSP compiler generated, implemented in `emcc_output.js`

The `ray_tracer.js` file contains a complete implementation of the ray tracer. The two functions mentioned above, both start with four `if`-statements that select one of the four implementations listed above.

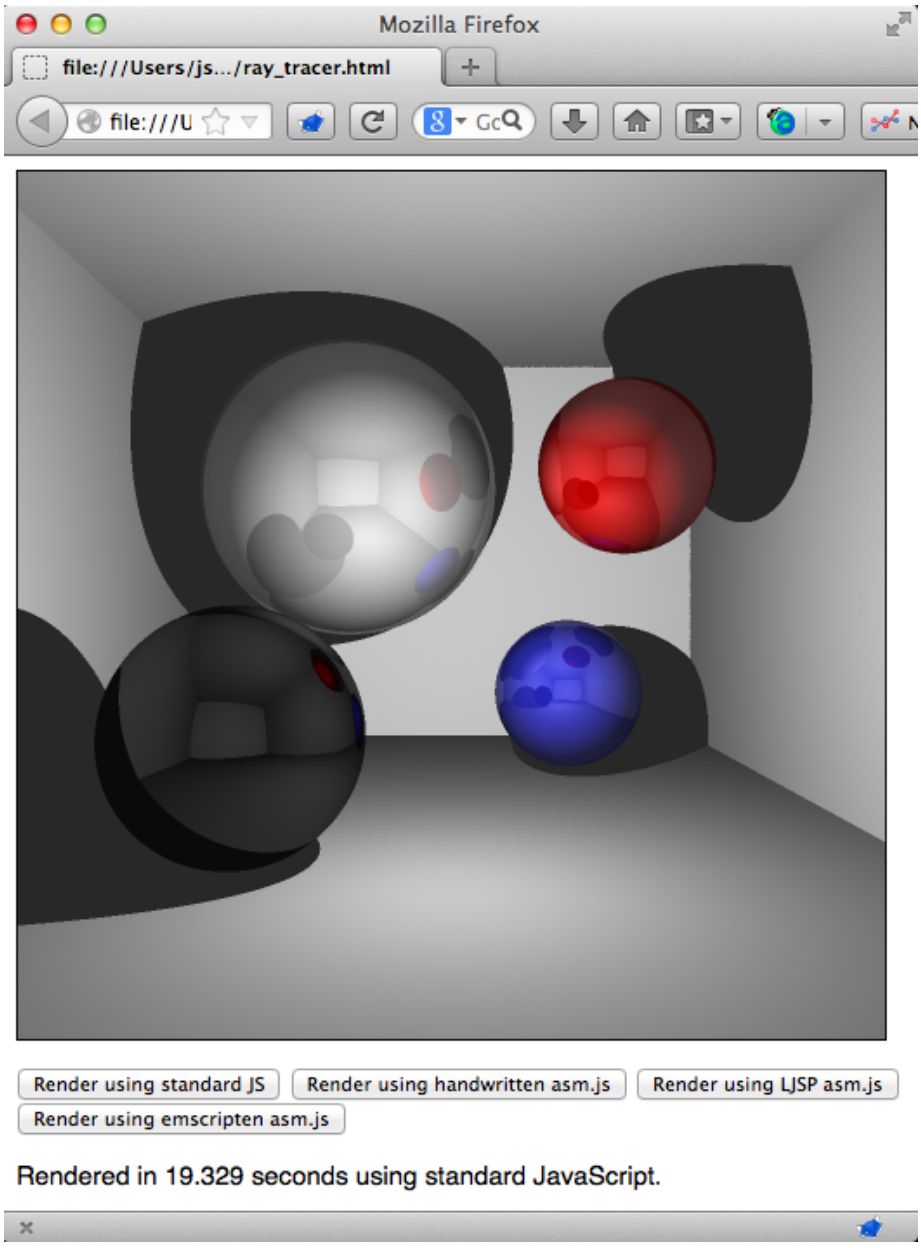

Figure 5.26: The interface of the ray tracer

5.5.3 Internals

The `ray_tracer.js` file

The internals of the ray tracer can be found in `ray_tracer.js`. This file consists of one function called `render`. The `render` function gets called with an integer value that determines which rendering method will be used and it writes the result of the rendering process to the canvas.

The `render` function is split into a number of sections. It starts with declarations of global variables that hold the objects making up the scene (`spheres`, `planes` and `lights`). The global variables section further contains the definitions of two variables that are used for floating point math (`hugeValue` and `tinyValue`).

The next section contains definitions for all the classes used in the ray tracer. It is followed by a number of sections that define functions on these classes, such as the dot product for the `Vector` class or colour mixing for the `Colour` class. None of these functions are particularly noteworthy, except for `vectorsDotProduct` which will be described below.

Following these small functions are the two intersection functions that calculate intersection points of rays with spheres and planes. These functions follow the mathematical description of the Design Chapter very closely. This section also contains the `closestIntersectionPoint` function that takes a ray as its parameter and loops over all objects in the scene, calling the two other functions in this section, depending on whether the current object is a sphere or a plane. An implementation detail not mentioned in the Design Chapter is that this function contains a safeguard inside the loop that loops over all the objects, shown in Figure 5.27.

```
1 if (k < tinyValue) {
2     continue;
3 }
```

Figure 5.27: Avoiding erroneous intersection points

`k` is the distance from the origin to the ray to the found intersection pont. When this value is very small, a reflected ray originating from the surface of an object was found to intersect that same object due to rounding errors. Because this is not the result we are looking for, the function skips this object and `continue`s.

The last function that makes up the main part of the ray tracer is the `trace` function. This function again follows the description given in the Design Chapter very closely.

The `render` function ends with a list of statements that make up the main method of the renderer. They create the objects that make

up the scene, initialise the canvas that the result is drawn on, call the `trace` function for every pixel and stop the time it takes to render. To achieve the antialiasing effect described in the Design Chapter, the `render` function creates two canvas objects, one that belongs to the canvas that the user sees and one, four times the size of the first one, that is hidden to the user. The renderer then renders the image on the second, large canvas and copies it to the first canvas at half scale.

Implementation of different rendering methods

The `render` function takes one parameter that determines the rendering method. The implementation of this is shown in the code snippet given in Figure 5.28, which shows an abridged version of the `raySphereIntersectionPoint` function.

```
1  function raySphereIntersectionPoint(r_original, s)
       {
2    if (renderType === 1) {
3      // Render using LJSP asm.js
4      return jModule.raySphereIntersectionPoint(...)
         ;
5    } else if (renderType === 2) {
6      // Render using Emscripten asm.js
7      ...
8    } else if (renderType === 3) {
9      // Render using handwritten asm.js
10     ...
11   } else if (renderType === 0) {
12     // Render using standard JavaScript
13     ...
14   }
15 }
```

Figure 5.28: Choosing one of the rendering methods

Depending on the value of renderType, one of the four rendering

methods is used.

6 Testing

This chapter describes the methods that were used to test the different subsystems of the LJSP compiler. While testing is imperative for all software development projects, it is especially important in compiler development. The reason for this is that it is usually impossible to determine whether the result generated by a compiler is correct simply by looking at the generated code. This is different to other areas of programming like web or user interface development, where a large number of bugs can be found simply by examining the program.

Just how essential good testing is for developing correct compilers was demonstrated by John Regehr et al. with their Csmith project [21]. By fuzzy-testing[1] C compilers it was possible to find around 325 bugs in both GCC and LLVM. Besides these two compilers, a third compiler for a limited subset of C described in [15] was tested. This compiler, called CompCert, was proven correct by its author using formal verification methods. Developing such proofs for critical parts of the LJSP compiler would be one possible starting point for future work.

Practically every time the testing framework of the LJSP compiler was extended to improve test coverage, some previously hidden bugs were found.

During the development of LJSP, two means of testing were employed:

- A custom testing framework written in Python that covers all of the front end stages and some of the back end stages.
- A ray tracer written in JavaScript that was used to test the generated asm.js code specifically.

The following two sections will describe both systems in detail.

6.1 Testing using `run_tests.py`

The `run_tests.py` file contains a custom testing framework that compares the results of different compilation stages with each other. It is based on the fact that no transformation should change the result of the program.

[1] Generating a large number of random but valid C programs.

`run_tests.py` was added to the project very early in the development of the compiler. It was initially used to test the front end stages of the LJSP compiler while they were being developed and it still fulfills that purpose. Additionally, it covers every back end stage except asm.js and Emscripten C conversion.

The framework contained in the `run_tests.py` comes bundled with a number of test cases which can be found in the `test/test_cases/` directory. These test cases are small LJSP programs that are designed for maximum coverage by testing different aspects of the LJSP language such as primitive operations, lambdas and recursion. It further uses the LJSP code contained in `test/test_lib.scm`, whose purpose and content will be described in the next section.

6.1.1 Implementation of the Testing Framework

Unlike the compiler itself, which was written in Scala, the testing framework was written in Python, because Python seemed a more adequate choice for writing the kind of "glue code" that the testing framework is is made up of.

The framework tests the different stages of the compiler by first evaluating the original LJSP source code using a Scheme interpreter. Subsequently, the program is compiled to all stages covered by the testing framework are the resulting code is evaluated. The result is then compared to the result of the original code. As mentioned in the Design Chapter, no transformation should affect the result that the program evaluates to. Therefore, if the two results do not match, the test fails.

The testing framework does not currently test intermediate stages. The Intermediate Representation (IR) is a language only used in the LJSP compiler, which means testing it would make it necessary to write a custom interpreter for it. Because the intermediate stages were a late addition to the compiler and are low in complexity, errors could so far be caught by tests covering the back end stages that follow the intermediate stages. Additionally, the IR conversion stage was created by combining duplicated code from the 'Conversion to asm.js' and 'Conversion to C' stages. This code had been tested in these earlier incarnations. Should development on the LJSP compiler continue, the addition of an IR interpreter and the test coverage of intermediate stages would be one of the next steps as we mention in the Conclusion Chapter.

6.1.2 Testing Front End Stages

As mentioned in the Design Chapter, all front end stages operate on LJSP code. Because the grammar of LJSP is almost a subset of the grammar of Scheme[2], this makes it possible to test these stages using

[2] As mentioned in the Specification Chapter, the only difference is the `neg` operator that LJSP uses for arithmetic negation. Scheme uses - for that.

a Scheme interpreter. The testing framework uses the Racket Scheme interpreter to evaluate LJSP code, but in principle, every Scheme interpreter could be used. For details on how to install Racket, please see the User Guide Chapter at the end of the report.

One thing to note is that while later front end stages still produce valid Scheme code, they make use of functions that are not part of plain Scheme. These functions are:

- `make-env` and `make-closure`, these functions are introduced in the closure conversion stage and are used to construct the closure data structure
- `get-env` and `get-proc`, these functions are also added during closure conversion and are the reverse of the two functions above, they are used to retrieve the environment and the code saved in a closure
- `nth`, this function, also used by the closure conversion stage, is used to access the values saved in environment variables
- `hoisted-lambda`, this function is used by the hoisting stage to construct closures that have a named, hoisted function as their code section

To still be able to test the LJSP code produced by all front end stages of the compiler, these functions, in addition to the `neg` operator, were implemented in Scheme. This code can be found in the `test/test_lib.scm` file. This file makes use of some more advanced features of Scheme, but it is heavily commented. To illustrate and explain the test library, the implementation of the `hoisted-lambda` function used by the hoisting stage is given in Figure 6.1.

```
1 (define (hoisted-lambda f env)
2   (lambda args (apply f (cons env args))))
```

Figure 6.1: Defining the `hoisted-lambda` function in Scheme

This function takes as arguments a function `f` and an expression `env` and returns a lambda. Calling the lambda returned by the function with a number of arguments causes function `f` to be called with `env` as its first argument and the arguments that the lambda is called with as its second to last arguments.

This is implemented making use of a feature of Scheme that allows a variable number of arguments. This is expressed in Scheme by leaving out the parentheses around the `args` following the `lambda`. The code further uses the `apply` function, which takes a function and a list and calls the function with the elements of the list as its parameters. Using

`apply` is necessary here because the number of arguments the function takes is not known in advance.

The content of the testing library is added in front of every piece of LJSP code generated by the front end stages that the testing framework evaluates.

6.1.3 Testing Back End Stages

Besides the front end stages of the compiler, the testing framework also tests three of its back end stages: 'Conversion to C', 'Conversion to LLVM IR' and 'Conversion to Numbered LLVM IR'.

These stages are tested by creating temporary files in a directory that is temporarily created and deleted after all back end stages succeed.

'Conversion to C' is the first back end stage the testing framework tests. Testing it requires three steps: First the LJSP compiler has to compile the original source file to a .c file that gets saved in the temporary directory. Then, the C-compiler of the system is used to compile this .c file to an executable. Finally, the executable is run and the result compared to the result of the original version of the program.

Testing LLVM IR code in both forms is somewhat simpler: The .s file generated by the LJSP compiler is interpreted using the `lli` command that comes as part of the LLVM distribution. `lli` is an interpreter for the LLVM Intermediate Representation. The result is again compared with the result of the original program.

Should any of these tests fail, the temporary directory does not get deleted, so that the generated files can be inspected and searched for bugs.

6.2 Testing using the Ray Tracer

The ray tracer included in the project was mostly used for benchmarking, but it was also a useful asset when testing the asm.js code generated by the compiler. Testing asm.js proved to be very difficult. The reason for this is that the only publicly available validator for asm.js is part of Firefox and can not be used separately from it. This meant that programmatically testing the output of the compiler, as was done for the other stages, was not practical. Testing of asm.js was therefore accomplished by compiling the complex function included with the ray tracer and validating it by manually opening it in Firefox.

One advantageous feature of asm.js that made debugging asm.js code somewhat easier was mentioned in the Background Chapter: asm.js code evaluates to the same result regardless of whether it gets compiled to machine code or whether it is interpreted as normal JavaScript. This made it possible during debugging to add statements such as `console.log`, that violate the specification of asm.js. While this would cause Firefox

to reject the module as valid asm.js, the rest of the code still produced the same result as it would have, had it been compiled to machine code.

7 Evaluation

This chapter will evaluate the results and findings of the project and put them into the context of what we set out to examine.

7.1 Benchmarks

The overall goal of this project was to evaluate asm.js and the promises about performance that were made during its launch. To achieve this, we compare four different implementations of the same function used in a ray tracer: one in plain JavaScript and three in asm.js. The results of this benchmark are given in table 7.1. It gives the amount of time in seconds it took to render a 3D scene using each of the four different implementations of `raySphereIntersectionPoint` five times. It also gives the average of the five running times of each method. Figure 7.2 shows shows the average running times of the four rendering methods in a graph.

	JavaScript	Handwritten	LJSP asm.js	Emscripten
	17.248	22.04	28.657	121.927
	17.505	21.753	29.243	121.737
	17.573	21.667	29.224	122.799
	17.358	21.699	29.354	122.357
	17.514	21.748	28.879	120.782
∅	17.4396	21.7814	29.0714	121.9204

Figure 7.1: Rendering times for different versions of the ray tracer in seconds

The first remarkable discovery we made using the ray tracer as benchmarking tool is that using the handwritten and fairly optimal asm.js version turned out to be 24.9% slower than using plain JavaScript to render the scene. This was a surprising result that went very much against our expectations.

The asm.js code generated by our compiler is 33.5% slower than the handwritten asm.js version. This is a result that we are very happy with, especially considering that the first working version of the LJSP compiler

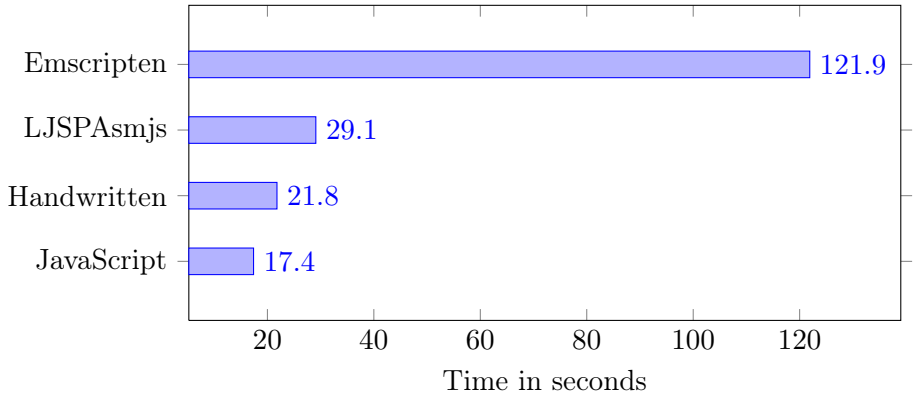

Figure 7.2: Rendering times of the four rendering methods

produced code that made the rendering take more than 60 seconds. With more and better optimisation stages, we are confident that the generated code could get even closer to the handwritten, optimal version. It is not likely, however, to reach the performance of plain JavaScript, as not even the optimal version is as fast.

The code generated by Emscripten from the C code the LJSP compiler generates is much slower than the other three versions, taking over two minutes on average to render the scene. It is 319.4% slower than the asm.js code generated by the LJSP compiler directly and 459.7% slower than the handwritten version. We strongly suspect that this is because Emscripten comes with a runtime that comes with a big overhead compared to the simple asm.js code the other two asm.js files are written in. This is reflected in the size of the files: The file that Emscripten generates is 4840 lines in total, whereas the handwritten version fits into 74 lines. If the benchmark was a more complex program, Emscripten's performance would surely improve relative to the other implementations.

We suspect that the overhead generated by repeatedly calling asm.js code from JavaScript is one of the reasons the version written in plain JavaScript is faster than even the optimal asm.js version. Additionally, current JavaScript engines already produce very efficient code for functions like `raySphereIntersectionPoint` that consist of simple mathematical operations.

Based on these discoveries and conjectures, we assume that the true strength of asm.js lies in being a compilation target for complex apps, such as the gaming engine compiled in the example that Mozilla gives on their webpage or interpreters for languages such as Python or Ruby, which are also sometimes used to demonstrate asm.js. Breaking down complex code, such as object oriented C++, into simple asm.js during compilation and executing the resulting machine code is much faster than translating the C++ code to high level JavaScript as the benchmarks by Mozilla and others show [1].

It is still surprising to us, however, that asm.js performs so badly in our use case. The fact that asm.js code is currently limited to arithmetic operations and read/write access to an array is one of the sources for criticism of it [8, 17]. We suspect that this is also why, despite the large initial interest, it has not yet achieved any serious level of adoption. Outside of a relatively small niche of applications that have little interaction with the DOM or other JavaScript code, such as gaming engines, interpreters and possibly cryptography, asm.js is simply not expressive enough. It is not possible, for example, to use asm.js for string manipulation. The developers of asm.js seem to be aware of this and David Herman, one of its developers, addressed this problem with a list of planned additions to the specification at [10] such as garbage collection and binary data handling in early 2013.

7.2 Evaluating the Compiler

While originally only conceived to compile LJSP to asm.js, the compiler was later extended to compile LJSP to C and LLVM IR as well. This decision was made after seeing the first benchmarks and was motivated by a desire to get a fuller picture of asm.js running time. Because Emscripten includes heavy optimisations, we thought it might produce ray tracing code that runs faster than the code generated by the LJSP, which turned out not to be the case.

The reason both C and LLVM IR are included as back ends is a remnant of the development process of the compiler. When developing the LLVM IR back end, we chose to first output C code, which is easier to generate because it is closer to the Intermediate Representation. We then compiled the C code this back end generated to LLVM IR using clang[1]. Using the output of clang as a reference, we added a second back end for LLVM IR to the LJSP compiler that imitated the output that clang generates. Because the LJSP compiler performs no optimisations when compiling from C to LLVM IR, the code these two back ends generate is equivalent in the operations it performs with the generated LLVM IR being more low-level and explicit. When compiling to asm.js using Emscripten, we used the output generated by the C back end and the `emcc` binary that comes with the Emscripten distribution. While we could have compiled the LLVM IR code generated by the LJSP compiler, both would yield the same result and C turned out to be somewhat easier to work with.

Projects that grow beyond their initial scope on often turn into an unorganised patchwork of additions. We are generally happy with how the compiler turned out after all extensions. It was, however, necessary to rewrite and refactor significant parts of the program at various points during the project to accommodate for new requirements. An example

[1] clang is the C compiler of the LLVM project.

of this is the Intermediate Representation and the intermediate stages that were not part of the first version of the compiler.

Using Scala to implement the compiler turned out to generally have been a good choice. Scala's pattern matching expression is very well suited for compiler writing and is used throughout the LJSP compiler. The possibility to mix functional and imperative programming styles was helpful and often resulted in expressive and elegant code. The downside of using Scala was the slow execution speed of the Scala compiler. This became a bigger and bigger problem as the project grew in size. Compiling the LJSP project in its current form takes almost 40 seconds on our machine, which made it no longer possible to make small incremental changes to the compiler and instead required us to batch together a number of changes before compiling. Additionally, on some occurrences, the Scala compiler segfaulted for unknown reasons, which was fixed by rerunning it with the same arguments.

The biggest design mistake made during the development of the LJSP compiler was to make all parameters to C functions to be of type `void*` instead of deriving their types from the way they are used inside the function body. Doing so would have made the C code easier to generate, less complex and faster. It would have avoided the need for much of the complex pointer handling and some memory management. It also would not have been very difficult to implement, because type derivation is already done for all local variables and could easily have been extended to include parameters as well. The reason we decided to make all parameters of the same type was because that is how the asm.js conversion stage, that had been implemented earlier, was designed. It worked well for asm.js, but it was not an optimal choice for C. Implementing type derivation for parameters, removing most of the calls to `jalloc` and compiling the code again with Emscripten would yield an interesting additional data point.

Another aspect of the compiler that could be improved is related to the fact that there is a substantial number of stages that only change expressions of a small number of types while leaving the remaining expressions unaltered. Examples for this are Prim op reduction, which only touches prim ops and closure conversion, which only changes lambdas and applications. To traverse the entire Abstract Syntax Tree, these stages need to handle every kind of expression, however, so that subexpressions can be converted. Closure converting the expression `(if` e_1 e_2 e_3`)`, for example, leaves the if untouched but causes recursive calls for all three subexpressions e_1, e_2 and e_3. Separating the AST-traversing algorithm from the changes that every stage makes early on would have made the code much more concise. Sarkar, Waddell, and Dybvig describe [19] a mechanism they call "pass expander" as part of their nano-pass compiler that automatically generates code to handle expressions in each stage that do not change.

A further mistake that was made in the beginning and was later corrected was the unnecessary introduction of `lambda`s to the code during CPS-translation. This seemed to be a good idea from a theoretical perspective looking at the CPS-translation equations, but in practice, it turned out to be very slow. Specifically in the case of `if`-expressions, described in detail in the CPS-translation section of the Implementation Chapter, it turned out that the trade-off between speed and code duplication did not exist in reality. Introducing additional `lambda`s made the code not just slower, but eventually also longer, because creating them requires lengthy construction of closures.

Much of the initial difficulty in developing the compiler was in understanding the concept of continuation-passing style and a lot of time was spent on finishing the three main front end stages: CPS-translation, closure conversion and hoisting. After that part of the project was completed, reading and understanding the asm.js specification turned out to be another challenge as the material on asm.js is limited and quite terse.

In conclusion, we are satisfied with the results that the project has achieved. Our initial requirements were met and the question with which we set out was answered. The results the project generated could be developed further in ways that we will describe in the next chapter.

8 Conclusions

This chapter will recapitulate the report and summarise the scope and findings of the project. It will also give an overview of possible future work on the LJSP project.

8.1 What we have done

With this project, we set out to evaluate asm.js as a compilation target for a LISP-dialect called LJSP. We wrote a compiler for LJSP that compiles to asm.js and also implemented a ray tracer in JavaScript to have a benchmark that would allow us to compare different versions of the code.

The initial result of asm.js code being much slower than plain JavaScript were surprising and going against our expectations. Although we were able to much improve the asm.js code generated by our compiler, this performance gap did not go away.

To get another data point, we extended our compiler to also be able to generate C and LLVM IR code from LJSP so that we could then pass this output on to the Emscripten compiler. The results this generated again surprised us by being much slower than any other version of the ray tracer the project includes.

Re-implementing the code in asm.js by hand revealed that even an optimal asm.js version was still slower than the version that used standard JavaScript only.

Our hypothesis is that asm.js is not suited for the kind of purpose we used it for: small functions that compute some relatively simple arithmetic operations, called repeatedly from standard JavaScript.

8.2 Future Work

The next step in evaluating asm.js would be to extend LJSP's grammar so that the entire ray tracer could be written in LJSP. This would allow us to compile code with a more complex structure to asm.js. It would also reduce the interaction between asm.js and non-asm.js JavaScript. The first step to reach this goal would be to implement lists.

Another addition to the project that would have to be made soon, should development on it continue, is an interpreter for the Intermediate Representation. This would then make it possible to include the intermediate stages in the stages covered by the testing framework.

Finally, additional intermediate stages could be added that optimise the generated code more heavily such as tail call elimination. The intermediate representation that is already part of the compiler offers a suitable framework for adding such stages. This would depend on the addition proposed in the previous paragraph, as these new stages would have to be tested to be reliable. Especially optimisation stages are prone to contain errors and keeping these new stages as small as possible, as described in [14] would make testing and debugging them easier.

9 User Guide

This chapter will describe the commands necessary to achieve the result presented in the previous, and other earlier chapters.

9.1 Compiler

The first step before the compiler can be used is to compile the compiler itself. This is done by calling `scalac -deprecation ./*.scala` from the root directory of the repository. We have included an executable version that was compiled with Scala version 2.9.1-1. Compiling the compiler will take some time (around 37 seconds on a 1.6 GHz MacBook Air).

After it has been compiled, the LJSP compiler can be called by entering `scala -cp ljsp/ ljsp.Ljsp` on the command line. Executing that command, without any switches, prints the usage of the compiler that shows all switches that are available.

These available command line arguments are:

- Either `-i` followed by a file name that contains LJSP code or code in a string, e.g. `scala -cp ljsp/ ljsp.Ljsp "(define (add x y) (+ x y))(add 1 2)"` but not both.
- One compilation target. The full list of possible targets can be viewed by calling the compiler without arguments and corresponds to the list of stages in the compiler. If no target is given, the output of all stages is printed to the command line.
- Optionally, `-o` followed by an output file name that the result of the compilation should be written to.

9.2 Ray Tracer

The ray tracer can be used by opening the `ray_tracer/ray_tracer.html` file in a recent version of Firefox[1]. Clicking on one of the four buttons will render a 3D scene with the rendering method selected. Please note that the renderer will not show anything until rendering has finished.

The HTML file depends on a number of JavaScript files, all of which are included in the repository. Two of these files can, however, be generated by compiling the `ray_tracer/ljsp_code.scm` file.

The first of these files is the JavaScript file containing the asm.js code for rendering the scene using asm.js code generated by the LJSP compiler (the rendering method activated by the third of the four buttons). This file is called `ray_tracer/gen_code.js` and can be generated by executing the following command from the root of the repository: `scala -cp ljsp/ ljsp.Ljsp --asmjs -i ray_tracer/ljsp_code.scm -o ray_tracer/gen_code.js`. Please note that the LJSP compiler must have been successfully compiled, before it can be used. See the first section of this chapter for details on how to do this.

[1] During the development of the ray tracer, Firefox 27 was used.

The second file contains the asm.js code generated by Emscripten (the forth of the four buttons renders the scene using this implementation). To generate this file, an Emscripten compatible C version of the code in `ljsp_code.scm` has to be generated using the LJSP compiler by running `scala -cp ljsp/ ljsp.Ljsp --emC -i ray_tracer/ljsp_code.scm -o rtc.c` from the root of the repository. This generates a file called `rtc.c`, also included with the repository. To compile this file and save the result in `ray_tracer/emcc_output.js`, which is where the ray tracer expects the Emscripten code, call Emscripten with the command shown in Figure 9.1. For this command to work, Emscripten must be installed and `emcc` in the directories of the system's `PATH`. For instructions on how to install Emscripten, please see the Emscripten tutorial[2].

```
1 emcc -O2 -s EXPORTED_FUNCTIONS="['
     _raySphereIntersectionPoint_copy_call_by_value
     ', '_vectorsDotProduct_copy_call_by_value']" -s
      NO_EXIT_RUNTIME=1 rtc.c -o ray_tracer/
     emcc_output.js
```

Figure 9.1: Running Emscripten on the generated code

9.3 Testing

Running tests can be achieved by executing `python run_tests.py` on the command line. This requires Python to be installed. The testing framework further depends on Racket and LLVM. `run_tests.py` contains two constants `RACKET_PATH` and `LLI_PATH` that have to point to the executables of racket[3] and lli[4]. These constants need to be adjusted to contain the correct paths.

[2] https://github.com/kripken/emscripten/wiki/Tutorial

[3] Racket can be downloaded at http://download.racket-lang.org/

[4] lli is the LLVM IR interpreter that comes with the LLVM project which can be downloaded at http://llvm.org/releases/download.html

10 Bibliography

[1] Are we fast yet, asm.js benchmarks and comparisons. `http://arewefastyet.com/#machine=11&view=breakdown&suite=asmjs-apps`. Accessed on 18. March 2014.

[2] Github archive. `http://www.githubarchive.org/`. Accessed on 18. March 2014.

[3] Hacker news asm.js release thread. `https://news.ycombinator.com/item?id=5227274`, February 2013. Accessed on 18. March 2014.

[4] Mozilla is unlocking the power of the web as a platform for gaming. `https://blog.mozilla.org/blog/2013/03/27/mozilla-is-unlocking-the-power-of-the-web-as-a-platform-for-gaming/`, March 2013. Accessed on 18. March 2014.

[5] Top github languages for 2013 (so far). `http://adambard.com/blog/top-github-languages-for-2013-so-far/`, August 2013. Accessed on 18. March 2014.

[6] Andrew W. Appel. *Compiling with Continuations (corr. version)*. Cambridge University Press, 2006.

[7] Douglas Crockford. *JavaScript - the good parts: unearthing the excellence in JavaScript*. O'Reilly, 2008.

[8] Vyacheslav Egorov. Why asm.js bothers me. `http://mrale.ph/blog/2013/03/28/why-asmjs-bothers-me.html`, March 2013. Accessed on 18. March 2014.

[9] Paul Graham. *On Lisp*. Prentice Hall, 1st edition, 1993.

[10] Dave Herman. David herman on future plans for asm.js. `https://news.ycombinator.com/item?id=5228042`, February 2013. Accessed on 18. March 2014.

[11] Dave Herman, Luke Wagner, and Alon Zakai. asm.js: a high performance subset of javascript. `https://github.com/dherman/asm.js/blob/master/tex/def.pdf`, January 2013. Accessed on 18. March 2014.

[12] David Herman, Luke Wagner, and Alon Zakai. asm.js, working draft. `http://asmjs.org/spec/latest/`, December 2013. Accessed on 18. March 2014.

[13] Tom Hughes-Croucher and Mike Wilson. *Node - Up and Running: Scalable Server-Side Code with JavaScript.* O'Reilly, 2012. Foreword by Brendan Eich.

[14] Andrew W. Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. *SIGPLAN Not.*, 48(9):343–350, September 2013.

[15] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[16] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, May 1999.

[17] Wladimir Palant. Thoughts on using asm.js for performance bottlenecks in browser extensions. `https://adblockplus.org/blog/thoghts-on-using-asmjs-for-performance-bottlenecks-in-browser-extensions`, June 2013. Accessed on 18. March 2014.

[18] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010. pp. 116-118.

[19] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass infrastructure for compiler education. *SIGPLAN Not.*, 39(9):201–212, September 2004.

[20] Charles Severance. Javascript: Designing a language in 10 days. *Computer*, 45(2):7–8, 2012.

[21] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.

[22] Alon Zakai. Emscripten: An llvm-to-javascript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, SPLASH '11, pages 301–312, New York, NY, USA, 2011. ACM.

[23] Alon Zakai. Big web app? compile it! `http://kripken.github.com/mloc_emscripten_talk/#/28`, February 2013. Accessed on 18. March 2014.