Bournemouth University

Computing
Final Year Project
3c - A JIT Compiler using LLVM

Edward Barrett
Supervisor: Laurence Tratt
May 21, 2009
Revision: 272

Word Count (excluding back matter)

```
FILE: 3c-compiler-Edd-Barrett.tex
Words in text: 14844
Words in headers: 217
Words in float captions: 451
Number of headers: 70
Number of floats: 27
Number of math inlines: 20
Number of math displayed: 0
```

Abstract

In the past, implementing virtual machines has either been a custom process or an endeavour into interfacing an existing virtual machine using (relatively) low level programming languages like C. Recently there has been a boom in high level scripting languages, which claim to make a programmer more productive, yet the field of compiler design is still rooted firmly with low level languages. It remains to be seen why language processors are not implemented in high level scripting languages.

The following report presents an investigation into designing and implementing computer languages using a modern compiler construction tool-kit called the "Low Level Virtual Machine" (LLVM), in combination with a modern scripting language. The report covers in detail traditional approaches to compiler construction, parsing and virtual machine theory. Comparisons are made between traditional approaches and the modern approach offered by LLVM, via an experimental object oriented language called 3c, developed using LLVM, the Aperiot parser, Python and an incremental development methodology.

Acknowledgements

This report is the result of several months of non-stop coding, typesetting and coffee fuelled late nights in the UNIX lab. This daunting, soul crushing, RSI inducing task would not have been possible had the following individuals not inspired, supported or otherwise enabled me:

Laurence Tratt For the project idea and his ongoing support and dry sense of humour during the months of development.

My Parents For the moral and financial support throughout the entirety of my University life.

Peter Knaggs For introducing me to language design and allowing me to attend his compiler design course alongside students of the year above.

Ruth Pitman For looking after me and caring in times of need.

#llvm on irc.oftc.net For helping me learn about LLVM and for a source of interesting discussion regarding compiler design.

Shaun Bendall For allowing me to skip work to attend Knaggs' compiler design course.

Didi Hoffman and Dave Hazell For their shared research and insight into language design.

and. . . Chris Taylor For keeping me insane.

Thanks to you all. We will all have to get a beer if this gets a good mark!

I would also like to thank the following software projects for their existence: LLVM, llvm-py, Aperiot, Python, TeX Live, OpenBSD, Vim, TeXworks, Inkscape, Xfig, Subversion, Trac, Graphviz, Rsync, OpenSSH and any others I forgot. Thanks!

ISC Software License

University Rights to Distribution

This document is submitted in partial fulfilment of the requirements for an honours degree at the University of Bournemouth. The author declares that this report is their own work and that it does not contravene any academic offence as specified in the University's regulations. Permission is hereby granted to the University to reproduce and distribute copies of this report in whole or in part.

Table of Contents

1 Introduction **1**

2 Background Study **2**
2.1 What is a Compiler? . . . 2
2.2 Compiler Sub-systems . . . 2
2.2.1 Tokenisation . . . 3
2.2.2 Syntax Analysis . . . 3
2.2.3 Semantic Analysis Stage . . . 4
2.2.4 The Synthesis Stage . . . 4
2.3 Type Systems . . . 5
2.4 Traditional Compiler Development . . . 5
2.5 What is a Virtual Machine? . . . 7
2.5.1 *System* Virtual Machines . . . 7
2.5.2 *Process* Virtual Machines . . . 7
2.6 The Low Level Virtual Machine . . . 8
2.6.1 Introducing LLVM Assembler . . . 8
2.6.2 The Structure of an LLVM IR Program . . . 9
2.6.3 Tight Integration with the Operating System Libraries . . . 10
2.6.4 LLVM APIs . . . 10
2.6.5 The LLVM Optimiser and Profiler . . . 12
2.7 Comparing LLVM to other Virtual Machines . . . 12
2.7.1 The Java Virtual Machine . . . 12
2.7.2 The Lua Virtual Machine . . . 14
2.7.3 Comparison to LLVM . . . 15

3 Software Engineering Technique **17**
3.1 Development Model . . . 17
3.2 Software Engineering Tools . . . 18

4 3c Design and Implementation **19**
4.1 System Overview . . . 19
4.2 3c Source Code Design . . . 19
4.3 Aperiot as a Tokeniser and Parser . . . 20

4.3.1 LL(1) Limitations . . . 23
4.4 The 3c Mid-Layer . . . 25
4.5 3c Object Hierarchy . . . 26
4.6 Basic Functionality . . . 27
4.6.1 Constructing Built-in Types . . . 27
4.6.2 Printing Values . . . 28
4.6.3 Variable Assignment . . . 28
4.6.4 Conditionals and Looping . . . 28
4.6.5 Functions . . . 30
4.7 IR Tables . . . 31
4.7.1 The Type-Sym Table . . . 31
4.7.2 The Virtual Function Table . . . 32
4.7.3 Polymorphic Operator Tables . . . 32
4.7.4 Example Table Usage . . . 34
4.8 Optimisation . . . 35
4.9 JIT . . . 36
4.9.1 Run-Time Errors . . . 36

5 3c in Practice - System Testing and Evaluation 37
5.1 Test Cases . . . 37
5.1.1 Boundary Value Analysis Tests . . . 37
5.1.2 Fibonacci Sequence . . . 38
5.1.3 Nesting Test . . . 38
5.2 Evaluation . . . 38
5.2.1 Evaluation of Development Technique . . . 39
5.2.2 Evaluation of Design and Implementation . . . 40
5.2.3 Evaluation of Testing . . . 42
5.3 Future Improvements . . . 43
5.3.1 Critical Improvements . . . 44
5.3.2 Non-Critical Improvements . . . 44
5.3.3 Enhancements . . . 46
5.4 Conclusion . . . 46

A References 48

B 3c Documentation 52
B.1 Installation Instructions . . . 52
B.2 Manual Page . . . 52
B.3 3c Syntax Reference . . . 54

C Testing Materials 55
C.1 Test Environment . . . 55
C.2 Boundary Value Analysis Tests . . . 58

C.2.1 Equality Operator Test 58
C.2.2 Less Than Operator Test 60
C.2.3 Greater Than Operator Test 60
C.2.4 Less Than or Equal Operator Test 62
C.2.5 Greater Than or Equal Operator Test 62
C.2.6 Inequality Operator Test 64
C.3 Fibonacci Tests 64
C.3.1 Program Listings 64
C.3.2 Results 68
C.4 Nesting Test 70
C.5 Contrived Optimiser Test 71

1. Introduction

The *Low Level Virtual Machine* is a modern compiler construction kit aiming to make building programming languages easier [LLV, 2009a]. The aim of the this report is to explore the possibilities offered by LLVM through the implementation of a programming language called *3c*. The development of 3c was experimental and heavily research driven. For this reason there were initially very few requirements set in stone in order to allow the project to adopt features based upon what LLVM can facilitate. Some very basic requirements were drafted:

- 3c must be free and open.
  The project must be written using open-source technology and released under a liberal license. This allows others to take the project source code and learn from it and adapt it.
- 3c must be simple for any programmer to use.
  The syntax of 3c must be conventional and understandable to other programmers of, for example: C, C++, Java and Python.
- 3c must be portable.
  3c must be portable across different operating platforms and computer architectures.
- 3c must implement common basic language features.
  3c needs to support at-least conditionals, loops, variables, functions and an integer representation.

Further requirements were adopted as the LLVM compiler infrastructure was explored and new possibilities became known. The author was particularly interested as to whether an object hierarchy could be implemented and if advanced OO concepts like polymorphism were feasible.

2. Background Study

Before any design or implementation took place, some research was performed. In this section, the field of compilers and virtual machines is studied, followed by a study into the Low Level Virtual Machine and it's key concepts.

2.1 What is a Compiler?

A *compiler* is one of the software systems falling under the category of *language processors* [Aho et al., 2007]. Broadly speaking, a compiler is a program which transforms one language into another. Typically, the input language is program source code and the output language is machine code, later to be executed by the host operating system, but other types of compiler exist:

- The Java compiler, outputs byte-code, which will be interpreted by the Java Virtual Machine (JVM) at a later date.
- The LaTeX compiler, outputs DVI (DeVice Independent) documents from LaTeX source code.
- The yacc "compiler compiler"[1], takes in a grammar specification and outputs C source code. Yacc is investigated in further depth in section 2.4.

When examined closely, a compiler is constructed of several sub-systems, which will be studied in further detail in section 2.2.

2.2 Compiler Sub-systems

As previously mentioned, there are several sub-systems of compilation. Typically these are a tokeniser, parser, semantic analyser and code synthesiser. Some compilers include various optimiser sub-systems, but such components are entirely optional.

The tokeniser. Scans the input, collecting groups of characters (*tokens*) which have semantic meaning.

The parser. Analyses the tokens and generates a *syntax tree* representation of the input.

The semantic analyser. Checks the tree makes sense semantically using static code analysis techniques.

The code synthesiser. Generates the output.

[1] No typographic error. It actually stands for *Yet Another Compiler Compiler*.

2.2.1 Tokenisation

During the tokenisation (or *lexical analysis*) stage, input is scanned and groups of characters with semantic meaning are identified as *tokens* which will be used by the parser for syntax analysis purposes. Take for example a language where a variable declaration is written as shown in figure 2.1. Non literal elements are marked in italic. Figure 2.2 shows how the input `let a = 1` might be tokenised for such a construct[2]. The resulting tokens are supplied to the semantic analysis sub-system.

```
let varname = number
```

Figure 2.1: An examplary variable assignment construct.

```
    let          a        =         1
     ↓           ↓        ↓         ↓
<id : "let"> <id : "a"> <=> <number : "1">
```

Figure 2.2: A tokenisation example.

2.2.2 Syntax Analysis

Following tokenisation, the parser will proceed to analyse the sequence of tokens, for which a *grammar specification* is required. To introduce the concept of parser grammars, this paper uses a commonly used notation, *Backus Naur Form* (*BNF*) [Aho et al., 2007]. Figure 2.3 shows an example of a simple *context-free* grammar expressed in BNF. A context-free grammar is one where rules have only one item on their left hand sides, the opposite of which being a *context-sensitive* grammar [Grune and Jacobs, 2008].

```
EXPR  →  EXPR OPER number
      |  number
OPER  →  +
      |  -
      |  *
      |  /
```

Figure 2.3: A simple BNF grammar specification.

Context-free BNF grammar specifications have a *production rule* name on the left hand side and at-least one list of tokens on the right. The lists define sequences of tokens which constitute a valid instance of that production rule. Once a grammar is defined, a set of input tokens can be applied and the result will be either a parse tree or a syntax error (in the case that the input tokens could not be applied to the grammar). A parser starts at the *initial rule* of the grammar. What happens next largely depends upon which parsing algorithm is used. A commonly used algorithm in computer languages is the LR algorithm[3], whereby tokens are read from **L**eft to right, using the **R**ight-most derivation (bottom-up). This continues until either an error is encountered and parsing

[2]Token types vary between parser implementations.

[3]Other algorithms such as LL(k) and LALR also exist. Parsing is a vast topic and an in depth study is out of the scope of this paper. For further reading on parser algorithms see Aho et al. [2007] and Grune and Jacobs [2008].

is aborted or until the sequence of tokens is exhausted, in which case parsing was successful and a parse tree can been visualised. As a means of example the input `1 + 2 + 3` is applied to the above example grammar (fig. 2.3), starting at `EXPR`. Figure 2.4 shows the resulting parse tree. Some parsers will go as far to optimise the parse tree, by removing ineffectual and intermediate modes, resulting in an *abstract syntax tree* (*AST*). Such optimisations have no bearing upon the functionality of the resulting output and are performed purely to improve the performance of the compiler [Discher and Richard J. LeBlanc, 1991]. In the remainder of this report the term *syntax tree* will be used to refer to either a parse tree or an abstract syntax tree, as they are handled identically in stages following syntax analysis.

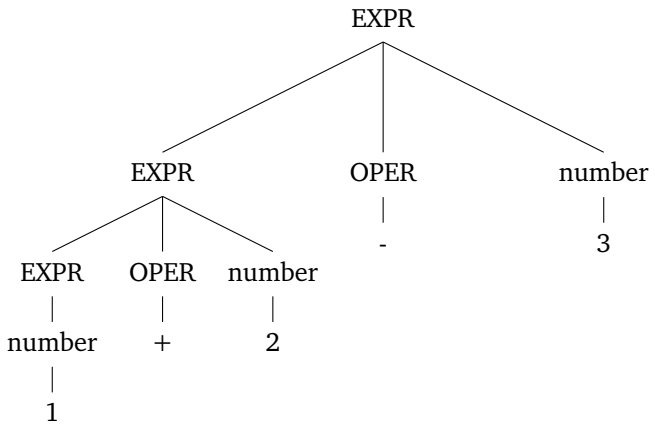


Figure 2.4: A parse tree derived by applying `1 + 2 - 3`, to the grammar defined in figure 2.3.

One can clearly see the input tokens reading horizontally from left to right on the leaf nodes of the tree. These tokens are said to be *terminal*, where-as all others are said to be *non-terminal*. One can follow the nodes up from the leaves and see how the terminal tokens were derived by the production rules of the grammar, until ultimately arriving at the initial rule.

2.2.3 Semantic Analysis Stage

Once the compiler has obtained a syntax tree for the given input, it is able to do some *semantic analysis*. At this stage the compiler will do some checks on the validity of the input which can not be realised from grammar analysis alone. A good example is type checking in statically typed languages such as C, where the compiler will check the types of certain constructs, checking they make sense semantically. If an array index were to be specified as a float in the C programming language, the compiler should (and does) abort, informing the user of a semantic error [Aho et al., 2007]. Platform independent optimisations may occur at this stage, depending upon the compiler implementation.

2.2.4 The Synthesis Stage

The final stage of any compilation process is the *code synthesis* stage (often shortened to just *code-gen*). This is where the syntax tree is converted into whichever output format the compiler is designed to fabricate. Also optionally, output target specific optimisations may take place at this time. Traditionally, executable code would be written to a file on disk, but as discussed before, the output may be of any format. The output may even be in the same format as the input, just transformed in some manner.

2.3 Type Systems

All computer languages need a method by which to infer typing information. The sub-system responsible for this task is the *type system*, of which there are 2 mainstream kinds [Rushton and College, 2004]:

Static Typing A statically typed system is one where all of the types in the language are explicitly encoded into the source language.

Dynamic Typing A dynamically typed system infers types at run-time and embeds little typing information within the source language.

Static typing allows a vast majority of type checking to occur at compile time because all types are known from the source language. It is also possible to have a much faster run-time with static typing because type checking occurs only once at compile-time, whereas in a dynamic system, the types are checked as the program is executed. Static typing has been criticised with claims that it restricts the expressiveness and flexibility of the language by imposing typing contracts. Examples of statically typed systems are: C, C++ and Java.

The dynamic typing approach allows the programmer to be far more flexible in their programming style, but there are some downsides. As discussed before, type checking is less efficient because type checking happens upon each execution of the program, but also fewer errors will be detected prior to runtime, meaning that testing will need to be more extensive. Certain object oriented concepts such as member function overloading can be complicated to achieve with dynamic typing, as such mechanisms traditionally rely heavily upon knowing argument types up-front. Examples of dynamically typed systems are: Python, Ruby and Lua.

It seems typing systems are largely subject to personal taste. Rushton and College [2004] suggests that no one type system is well applied to all problems and that a suitable typing method should be derived from the nature of the problem domain.

2.4 Traditional Compiler Development

One popular approach to implementing a compiler is to use a lexical analyser and parser generator tool-kit such as *lex* and *yacc*[4]. The programmer will then add their own code in order to synthesise output based upon the output from the parser. Figure 2.5 shows the structure of such a system.

Many widely used general purpose computer languages have adopted this method in part or full[5]. It is also worth noting that these tools are not solely used in the language development context, as many projects are using tokeniser/parser generator tool-kits for other purposes. Examples of applications using such tool-kits include:

- The Perl scripting language.
  Uses BSD yacc for parsing grammar [Wall, 2009].

- The TCL scripting language.
  Uses BSD yacc or GNU bison for parsing [TCL, 2009].

- The Solaris Operating Environment's `zonecfg` command.
  Uses implementations of lex and yacc to parse commands entered by the user to configure zone virtualisation [ZON, 2009].

[4] Or alternatives such as *flex* and *bison*.

[5] Some projects using yacc or bison use their own tokenisers.

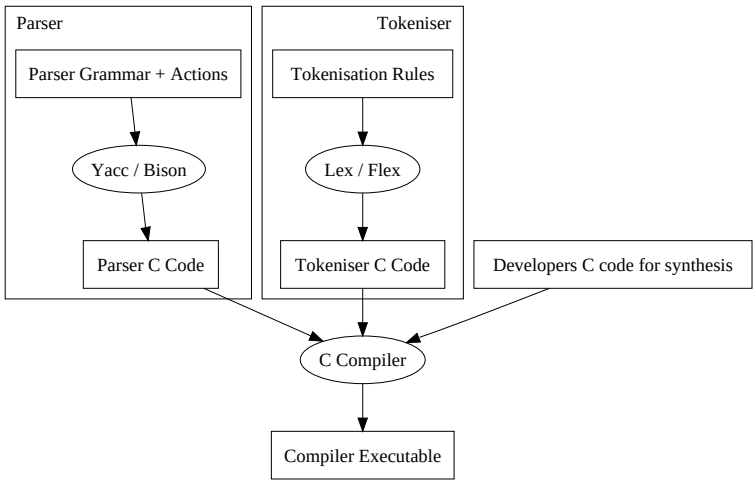


Figure 2.5: Diagram showing typical utilisation of lex and yacc in compiler development.

- The Portable C Compiler (PCC).
  Uses both BSD lex and yacc for it's C preprocessor [Anders Magnusson, based upon works of Stephen C. Johnson of Bell Labs, 2009].
- The Ruby scripting language.
  Uses GNU bison in combination with a custom tokeniser [RUB, 2009].
- The PHP scripting language.
  Uses GNU bison in combination with the Zend language scanner [PHP, 2009].
- The Calm Window Manager (CWM).
  Uses BSD yacc and a custom tokeniser to parse it's configuration file [CWM, 2009]
- Etc...

The output of both lex and yacc is C source code, which can be compiled into object files and linked using a compile time linker. The programmer is free to link any object files in at this stage, making adding parsers and tokenisers to C programs easy and flexible. All the programmer need do in his application is call the functions the tokeniser and parser objects files define[6]. The resulting executable is the compiler itself, which is ready to take source code input and (in a single process) perform the compilation stages discussed in section 2.2. The reason for the popularity of this approach is clear:

- The tool kit is proven in the wild in real and successful software projects.
- The tools are free and open-source [FLE, 2009][YAC, 2009][BIS, 2009][7].
- The tools are multi-platform, as they are written in and generate portable C code.
- The tools are default on a wide number of systems and can are easily added if not [CYG, 2009].

[6] Usually `yylex()` and `yyparse()`.
[7] Including the C compiler in some cases [GCC, 2009]

2.5 What is a Virtual Machine?

A *Virtual Machine* (VM) is a software layer that provides the user with a pretence of having a machine other than the actual hardware in use [Rowledge, 2001]. There are currently two different interpretations of the term, the meanings of which, although similar, are not the same.

2.5.1 *System* Virtual Machines

A *system* virtual machine is one which emulates (in software) a real system, one which exists as a piece of hardware. Such virtual machines emulate closely the CPU, registers and memory of the real-life implementation as closely as possible. Such set-ups have been used for various reasons. Currently it is believed that by using (system) virtual machines as a replacement for real systems, companies may reduce the cost of running a data-centre and therefore the total cost of ownership [BIG, 2007]. The reasoning behind this is that running fewer machines costs less in power for cooling and for powering the otherwise physical machines themselves. The other main uses for system virtual machines are:

- To run different operating systems, be that different versions of the same system, or entirely different systems altogether.
- For privilege separation uses, for example where administrators require many users to have administrator privileges on their own dedicated systems.

This paper is not concerned with system virtual machines and one should assume that the term "virtual machine" refers to a *process* virtual machine from this point onward.

2.5.2 *Process* Virtual Machines

The second type of virtual machine and the one most relevant to this paper is the *process* virtual machine. The process virtual machine provides an execution environment for computer program code. Such systems use the concept of an intermediate representation *byte-code* (or *bit-code*), which is executed within the *sandbox* of the machine.

A process virtual machine is usually implemented in one of two ways: *stack-based* or *register-based*, stack-based being the most common. In a stack-based VM, an *argument stack* is present [Ierusalimschy, 2003][8]. When an operation is executed, the number of required arguments is popped from this stack and the result of the operation is pushed back on to the stack after it has been computed. In a *register-based* machine however, there is no argument stack and data is stored in named *registers* instead. It has long been debated as to which architecture is the best for a process virtual machine. A register based approach can reduce significantly the number of instructions needed, but argument look-up is thought to be less efficient, as the registers must be resolved, leading to larger code. In a stack based system, many more instructions are required to push the data on to the stack initially, but it is quicker to retrieve arguments later, as they will be a known offset to a *stack pointer* [Shi et al., 2005].

There are several execution strategies for virtual machines, the simplest of which is *interpretation*, which means the byte-code is executed statement-by-statement on the fly. Execution in such a way tends to be slower, but uses considerably less memory resources [JAZ, 2009]. Another way to execute code in a VM is with *Just In Time* compilation (or JIT compilation). JIT compilation aims to improve the performance of execution by compiling blocks of the code path to native code as they are encountered at run time. This approach can help improve overall performance of code,

[8] which should not be confused with a stack of frames.

but often leads to slow start-up times whilst many code paths are encountered and compiled for the first time. Such an approach also uses more memory than direct interpretation [JAZ, 2009].

2.6 The Low Level Virtual Machine

The Low Level Virtual Machine(LLVM)[LLV, 2009a] is a register-based compiler framework which aims to provide a standardised tool-kit for mid-level and front-end language development. Like many virtual machines it works with an *intermediate representation* language. Although this concept is not new [Discher and Richard J. LeBlanc, 1991], LLVM aims to further extend and refine the process. Figure 2.6[9] shows some of the ways a developer might choose to implement LLVM. Some of the sub-systems in LLVM are synonymous to those of the "traditional compiler" described in section 2.4, but fundamentally LLVM is different. In-fact LLVM differs from even the mainstream definition of a *virtual machine*:

- Provides a standard API and compiler back-end for many different compilers.
- Is not just an execution environment, although a JIT engine is provided, should it be needed.
- Ships with tools to make platform specific assembler from on-disk byte-code, this can be assembled and linked in order to make an executable binary[10].
- Has very strong type checking. Far stronger than C.
- Has a comprehensive optimiser framework for many specific platforms.
- Has a number of profiling utilities.
- Can be used solely as an interpreter, using byte-code from disk.
- Can be interfaced by a number of language bindings, allowing parsers and tokenisers to be implemented in high level languages.

2.6.1 Introducing LLVM Assembler

As stated before, LLVM uses an intermediate representation format which the user can express as LLVM assembler. This assembler code can then be assembled into *bit-code*, which is an in-memory data structure representation. Once in memory the bit-code may be executed, dumped to an IR assembler file or converted into CPU instructions as platform specific assembler code. It is important that one makes a clear distinction between *bit-code* and *byte-code*, as although in general the terms are interchangeable, in the context of LLVM they are *not* the same. Both formats are derived from assembled LLVM assembler source code, but *byte-code* is strictly an on-disk format, whereas *bit-code* is an in-memory data structure for internal use only.

Listing 2.1: "Hello World" in LLVM IR.

```
1 ; ModuleID = 'mod'
2
3 define i32 @main() {
4 entry:
5         %0 = alloca [4 x i8]            ; <[4 x i8]*> [#uses=2]
6         store [4 x i8] c"%s\0A\00", [4 x i8]* %0
```

[9] LLVM is aiming in the future to remove the dependency upon GCC to make native binaries.

[10] Ultimately LLVM will be able to emit executable binaries directly, but these features are still under heavy development.

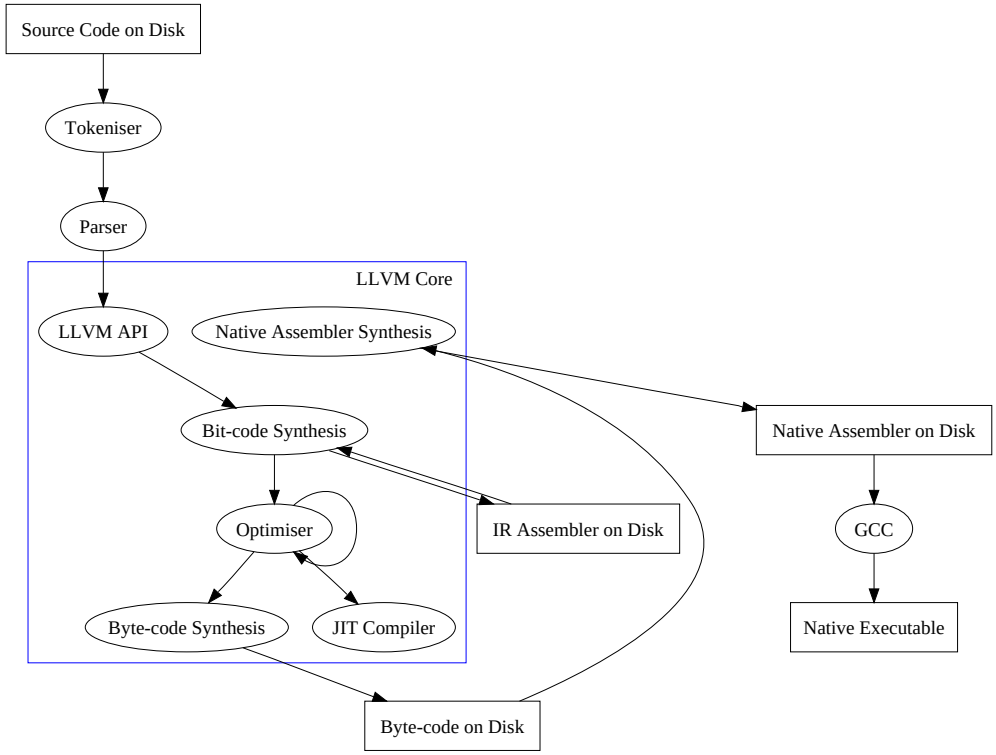


Figure 2.6: A diagram showing various ways in which the user can interact with LLVM.

```
7         %1 = getelementptr [4 x i8]* %0, i32 0, i32 0           ; <i8*> [#
              uses=1]
8         %2 = alloca [13 x i8]           ; <[13 x i8]*> [#uses=2]
9         store [13 x i8] c"Hello World!\00", [13 x i8]* %2
10        %3 = getelementptr [13 x i8]* %2, i32 0, i32 0          ; <i8*> [#
              uses=1]
11        %4 = call i32 (i8*, ...)* @printf(i8* %1, i8* %3)               ;
              <i32> [#uses=0]
12        ret i32 0
13 }
14
15 declare i32 @printf(i8*, ...)
```

Listing 2.1 shows the compulsory “Hello World” program in LLVM assembler. At a glance it looks similar to microprocessor assembler code, but there are some important differences which will be highlighted in the following sections.

2.6.2 The Structure of an LLVM IR Program

The top level component of any LLVM assembler program is the *module*, which acts like a container for one or more functions and perhaps some global variables. Unlike most assembler implementations, LLVM assembler supports the concept of functions. Each function has it’s own stack frame

(and therefore it's own variable scope). A function may take a number of arguments[11] and may return one value. Arguments and return values must be one of the so called *first class* types: integer, floating point number, pointer, vector, structure, array or label. [LLV, 2009b]. Each function may have 0 or more blocks. A function with no block is an *external* function, for example a function in libc, like `printf(3)`. A *block* is a container for assembler instructions. Each block must be terminated by a `return` or a `branch` to another block. Branching may be selective, therefore allowing looping and conditional constructs.

2.6.3 Tight Integration with the Operating System Libraries

LLVM has the ability to use the underlying operating environment's system calls and native shared object libraries, directly from within byte-code. A system call is a call to a C function of the operating system which requests a kernel facility [Stevens, 1992]. By providing this interface, LLVM has very tight integration with the file-system, network stack and memory management features of the system. In-fact once one realises that LLVM can call the `dlopen(3)` system-call, a whole new universe of possibilities opens. `dlopen(3)` is used to import shared libraries at runtime, meaning that during JIT execution, the program could do some quite weird and wonderful things, like drawing graphical user-interfaces, or interacting with relational databases databases. With other languages, this typically requires extending the VM with a plug-in shared object written in C [The Python Development Team, 2009] [Jung and Brown, 2007], but with LLVM any C function may be called with no need to modify the VM or write plug-ins.

2.6.4 LLVM APIs

Although one could write LLVM assembler code by hand, it would be very cumbersome and error prone. Assembler code is easier to fabricate in an auto-generated fashion, by either an LLVM language binding or by a third party compiler. The LLVM distribution provides a C++ API, which is well documented on the LLVM web-page, however other third party bindings are being developed. The "Hello World" assembler code show in figure 2.1 was generated using the Python bindings [R Mahadevan, 2009]. Listing 2.2 shows the Python source code that was used. Currently you may interface LLVM via C, C++, Ruby [LLV, 2009c], Python [R Mahadevan, 2009] and Haskell [O'Sullivan, 2009].

[11] which may be of variable length (*varags*).

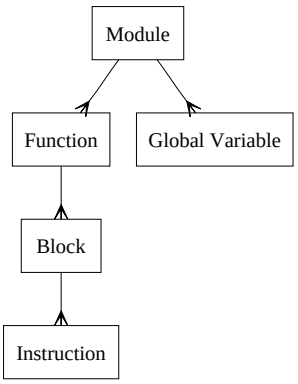


Figure 2.7: Diagram showing the multiplicity of the elements of an LLVM module.

Listing 2.2: The Python script used to generate “Hello World”.

```
#!/usr/bin/env python
# $Id: world.py 154 2009-04-26 13:31:43Z edd $

from llvm import *
from llvm.core import *
from llvm.ee import *

int_t = Type.int(32)
i8_p = Type.pointer(Type.int(8))
zero = Constant.int(int_t, 0)

mod = Module.new('mod')

main_sig = Type.function(int_t, [])
main = mod.add_function(main_sig, "main")

printf_sig = Type.function(int_t, [ i8_p ], True)
printf = mod.add_function(printf_sig, "printf")

block = main.append_basic_block("entry")
b = Builder.new(block)

# ------ main

# make printf format string
fmt_c = Constant.stringz("%s\n")
fmt_p = b.alloca(fmt_c.type)
b.store(fmt_c, fmt_p)
fmt_i8_p = b.gep(fmt_p, [ zero, zero ])

# make output string
str_c = Constant.stringz("Hello World!")
str_p = b.alloca(str_c.type)
b.store(str_c, str_p)
str_i8_p = b.gep(str_p, [ zero, zero ])

b.call(printf, [ fmt_i8_p, str_i8_p ])

b.ret(Constant.int(int_t, 0))

# /----- main

print mod

```

```
45 # jit
46 mp = ModuleProvider.new(mod)
47 ee = ExecutionEngine.new(mp)
48 retval = ee.run_function(main, [])
```

2.6.5 The LLVM Optimiser and Profiler

One feature of LLVM is that it is able to optimise at several stages of the compile/execute life-cycle: compile-time, link-time and run-time [Lattner and Adve, 2004]. LLVM defines a set of *optimiser passes*, which the user may turn on individually, according to their needs. A typical optimiser pass will transform the bit-code representation of a program, arriving at a new, functionally identical program, which when applied properly, can improve the performance or size of a program. Some optimiser passes do not transform the program at-all. These passes are *profiler* passes. Instead of altering the program, such passes only analyse it, allowing the developer to spot possible bottlenecks and shortcomings in their programs. At the time of writing, there are 63 optimiser passes defined in LLVM [Spencer, 2009].

2.7 Comparing LLVM to other Virtual Machines

As previously mentioned, the concept of the virtual machine is not new. Several other virtual machine implementations were developed prior to the birth of LLVM. In this section two other VM implementations are investigated and contrasted to LLVM.

2.7.1 The Java Virtual Machine

The Java Virtual Machine (JVM) is a product of Sun Microsystems and is the stack-based JIT engine underlying the Java programming language [Lindholm and Yellin, 1999]. The JVM is designed to provide platform independence and validity of byte-code programs. The JVM's byte-code format is the *class* file format. A class file is a binary format which is ready for JIT execution in the JVM. It is easy to make the assumption that a class file has a one to one relationship with a Java class definition, however this is untrue. The byte-code format is completely disjointed from the Java programming language, but does lend itself to object oriented representations [Lindholm and Yellin, 1999].

Hotspot Technology

Traditional JIT compilers, do not build the entire byte-code into native instructions, but instead compile the code-path directly ahead at runtime. This could happen on a per method basis for example. The JVM builds further on this idea, by profiling code as it is running, then compiling and transforming the code path based upon "hot spots". Initially the JVM will act as an interpreter, executing statements one by one and gathering profiling data. If a certain code path is executed frequently enough to make compilation beneficial, the JVM will compile that section of the byte-code to native instructions. Once a code-path has been compiled, strictly the JVM is no longer an interpreter, but a hybrid interpreter/JIT engine.

Validity Checking of Class Files

A popular use for Java on the internet is for *applets* and application servers [TOM, 2009]. A Java applet allows the user to run a JVM instance inside a web browser to deliver dynamic content. Byte-code originating from an unknown source in a networked environment has security and validity implications. For this reason, the JVM does some checks on the byte-code it is presented prior to it's execution, which happens in several stages:

1. Byte-code structure checks.
2. Data flow independent checks.
3. Data flow analysis checks.
4. Symbol Checking.

Initially the candidate byte-code is checked for structural integrity. The JVM checks for "magic bits" at the beginning of the file (`0xCAFEBABE`) and that the byte-code version is compatible with the VM implementation[12]. The file is checked for correct termination with no extra bytes at the end etc. Next some checks on the basic semantics of the code, which do not require data flow analysis are done. During this pass the JVM checks all symbols have been given valid names and that *final* classes have not been super-classed for example. Following this data flow analysis based checks are performed, for example "define/use pairs" (D/U pairs) are analysed to check no variable is a accessed before it is defined. During this phase of verification, each instruction that is checked has it's *changed* bit set, in order that it not be checked twice. In the final stage reference types are checked and various referenced class attributes and methods are checked for existence.

The validation process described above is one preventative measure against foreign, possibly malicious byte-code. Another design feature which greatly improves the security of a class file is it's proactive memory management. In a non-memory-managed language such as C, the user must have knowledge of pointer types and memory allocation routines, such as `malloc(3)`. For low level hardware programming this is ideal, as the programmer will want true flexibility with how he/she deals with data structures in memory and on disk, possibly optimising routines with pointer arithmetic. This makes it very easy to forge bad pointers, either accidentally or with malicious intent.

Figure 2.8 shows a C program, which demonstrates a memory management mistake. The programmer has accidentally freed up the memory buffer storing the variable `a`, before a call to `printf(3)`, where the pointer (to the now free memory buffer) is dereferenced. As far as the C compiler is concerned, this is valid, as the pointer to `a` remains on the stack after the call to `free(3)`. Only at run-time can the user *potentially* detect this error. Unfortunately in many environments this program will succeed, despite the programming error. Later on the operating system may allocate the same memory `a` points to for a different type of data structure, causing subsequent uses of `a` to demonstrate undefined behaviour. It was (and still is) programming errors similar to this that malicious users exploited in order to execute arbitrary code with harmful intent. The classic example would be using a stack overflow to over-write the return address of a function, in order to execute malicious code. Although some tools can help identify memory management errors [OBS, 2009][VAL, 2009][Perens, 2009], the result largely depends upon the host operating environment. Additionally not all developers will adopt such tools.

The JVM byte-code format pro-actively prevents memory management errors, by handing off memory management to the JVM directly, leaving the programmer unconcerned with such tasks. Because the class file format has no concept of pointers, it makes it more difficult to forge references to unallocated memory [Gosling and McGilton, 1995]. This safety feature of the JVM comes at the price of reduced flexibility.

[12]Not only are there multiple version numbers of Java, but also different implementations from different vendors.

```c
#include <stdio.h>

int
main(void)
{
        int *a = (int *) malloc(sizeof(int));

        *a = 666;
        free(a);

        printf("a is now %d", *a);

        return 0;
}
```

Figure 2.8: A contrived memory management programming fault in C.

2.7.2 The Lua Virtual Machine

The Lua scripting language is a small lightweight scripting language mostly used as a plug-in language, providing applications with scripting support. Lua byte-code, as of version 5, executes in a register based virtual machine and was one of the first of this kind to be adopted in the mainstream [Ierusalimschy et al.]. The Lua VM has has been designed from the ground up with several specific goals in sight. Lua was meant to be small, fast, portable, embeddable and under a license suitable for industry[13].

Lua 4 Versus Lua 5 - Stack based to Register Based

The Lua virtual machine used to be a stack based virtual machine, but as of release 5 is a register based VM. The authors of Lua have justified this decision in depth [Ierusalimschy et al.], reasoning that in a stack based VM, some operations require values on the stack to be moved and swapped, which not only defeats the point of a stack, but is subject to an excessive instruction count and requires repeated use of the expensive `copy` instruction. Secondly, although register machines *do* have a larger instruction size, due to having to specify operands explicitly, Ierusalimschy et al. argue that the overhead is exaggerated as register operands can be resolved using short and cheap CPU instructions, where-as stack operands often require large instruction operands which can not be despatched (portably) in one CPU cycle. The JVM's branching implementation is given as an example of such a case. Lastly it was thought that by using a larger number of registers (than physical machines), many more local variables may be stored directly in registers (as opposed to globals), making local variable access very fast [de Figueiredo et al., 2008]. These rather bold claims were backed up with solid evidence that the register based Lua 5 out-performed Lua 4 in lab tests by an average of 70% (fig. 2.9).

Listing 2.3: "Lua 4 (stack-based instructions)"

```
GETLOCAL        0       ; a
GETLOCAL        1       ; b
```

[13] Lua is distributed under the terms of the MIT open source license.

```
3 ADD                       ; +
4 SETLOCAL          2       ; c =
```

Listing 2.4: ”Lua 5 (register-based instructions)”

```
1 ADD               3       1       2; c = a + b
```

2.7.3 Comparison to LLVM

Having studied the JVM alongside LLVM, it is clear that the projects, although in the same software category, have different goals. The JVM aims to hide the underlying machine and to allow safe execution of a program via JIT compilation regardless of the underlying hardware platform. LLVM is more limited than the JVM in terms of dynamic execution, and *can* JIT compile code, but has nothing as intricate as a hotspot compiler. Having said that the JVM does not ship with any tools to help make executable binaries. LLVM on the other hand has a tool called *llc*, which converts byte-code to platform specific assembler code.The hotspot compiler is a novel touch, but it is important to remember that the constant profiling of the code-path will cost CPU cycles and memory capacity itself. Further research in this area would be interesting.

LLVM has much of validity checking the JVM has, but it is only enabled optionally through an API call (`verify()`). Additionally, memory management in LLVM is manual and exposes pointer types to the language implementer. As with the C example just presented, this makes buffer over-runs and other pointer based mis-adventures easy, but does provide the programmer with more low level flexibility.

Lua shares it's register-based design with LLVM, but again is a very different system. The authors of Lua have managed to harness the benefits of a register-based machine in a portable nature, however it is yet to be seen if other stack-based VM's which aim to be less portable can out-perform Lua. The Sun's Hotspot Java VM implementation, for example is only targeted at x86/64 and SPARC based machines. It is likely that at the cost of portability, the Hotspot compiler can generate some CPU specific instructions in order to improve the speed of execution. LLVM aims to have the best of both worlds, by having knowledge of a diverse range of computer architectures, optimising at the instruction level for a specific target, but whilst also remaining incredibly portable across a variety of UNIX, Windows and other platforms. The obvious outcome of this is that the development time (of LLVM) will be many more man hours, which seems to stand true, as the code-gen and interpreter sub-systems of LLVM are mostly incomplete since the year 2000. Another unfortunate side-effect to LLVM's portable development strategy is that it has exposed

Program	**Lua 4.0**	**Lua 5.0**	**%**
sum (2e7)	1.23	0.54	44
fibo (30)	0.95	0.69	73
ack (8)	1.00	0.88	88
random (1e6)	1.04	0.96	92
sieve (100)	0.93	0.57	61
heapsort (5e4)	1.08	0.70	65
matrix (50)	0.84	0.59	70
average			70

Figure 2.9: Lua speed improvements [Ierusalimschy et al.]

some odd bugs in certain versions on C compilers (namely GCC) on certain platforms [The LLVM Development Team, 2009], however this is expected and is not the fault of LLVM.

Really, the name “The Low Level Virtual Machine”, is a somewhat misleading name as it is much more of a compiler construction kit than a VM. Other VM’s like Lua and Java are much more conventional in the sense that they are *only* execution environments in a sandbox. LLVM does not aim to be a sandbox and offers the implementer access to operating system calls directly. There is a JIT engine available, but the implementer need not use it if he/she does not want or need to. It is low level and flexible. It includes few high level compiler technologies, but provides the facilities for a developer to implement them. Features such as garbage collection and run-time profiling are achievable if required. In a nutshell, LLVM is a re-usable back-end, which is re-targetable and optimised at multiple stages for a wide range of computer architectures, yet hiding the specifics from the compiler implementer.

3. Software Engineering Technique

Having researched compiler technology, planning of the 3c compiler started. A suitable development strategy is an important consideration for software projects. In this section software engineering techniques are explored and the proposed development strategy for the 3c compiler is presented.

3.1 Development Model

There are several well documented approaches that a software engineer may adopt for a project, the most classic example of which being the *waterfall* methodology. In the waterfall methodology distinct milestones for requirements engineering, design, implementation, testing and maintenance are identified. Development then moves from one milestone to the next sequentially. Building upon this the *V* methodology works in much the same way, but goes further, identifying deliverables and their dependencies with a strong emphasis on testing [Forsberg and Mooz, 1994]. Conservative methods like the *waterfall* and *V* methodologies allow little opportunity to re-work the design at a later date, which is somewhat unrealistic in the world of software engineering. Iterative approaches on the other hand, allow the key stages of software development to be tackled in part in a number of small iterations, meaning that there is plenty of space for the project design to be adapted mid-development.

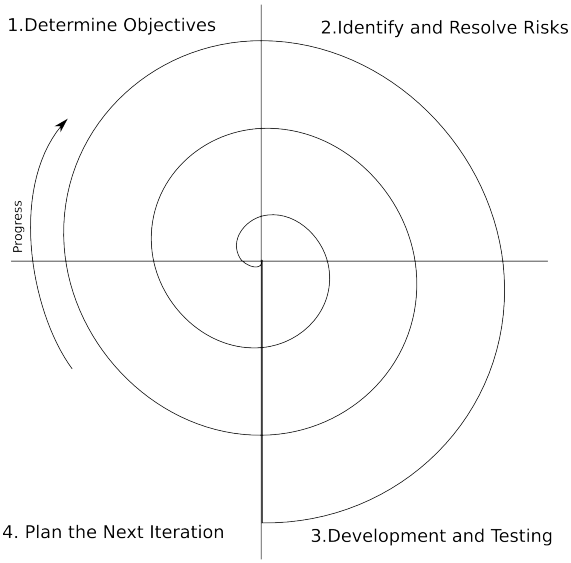


Figure 3.1: The spiral development model [Forsberg and Mooz, 1994]

The development model chosen for the 3c project was the *spiral* technique [Boehm, 1988] (fig. 3.1), mainly for it's iterative qualities. The waterfall and V methods were disregarded due to their

need for mostly concrete requirements and design prior to any implementation. It was thought (at the time) that the author had too little understanding and insight of the capabilities of the tools for full requirements and design to be developed prior to at least a partial implementation. It is also worth noting that at the time of writing, little of the supporting technology 3c is based upon was mature: llvm-py was not fully complete and largely undocumented in places, LLVM itself was unfinished etc. By adopting an iterative methodology, design and specification changes could be made dynamically to respond to tool-chain shortcomings or to the discovery of undocumented new features. Another iterative methodology, *extreme programming* (XP) was also considered, but it was decided 3c would not suite well. XP is largely based upon distinct roles within a mid-sized development team, fed by feedback from an external end-user [Cockburn, 2007]. The development of 3c is by a sole software engineer and is mostly driven by curiosity and by expectations of a programming language in the author's experience, deeming XP inappropriate.

3.2 Software Engineering Tools

Some tools (other than programming tools) will be used to help the software development life-cycle, the keystone of which is *subversion* (sometimes called *SVN*) [SVN, 2009]. Subversion is a source code management system which has been heavily adopted in the open-source community due to its liberal license and in many cases has replaced the *concurrent versions system* entirely. The function of subversion is to track changes to source code over time and provide information about when and what changed as well as where in the source code and by whom. This involves a SVN server, holding a code *repository* and allows developers to "check out" project code, edit the code and then check the code back in. This allows developers to track development progress and helps to locate the cause of any recently introduced software faults. Subversion also supports the concept of code *branches*, which are commonly used to store different versions of a source tree. One branch was used for every cycle of the spiral software development life-cycle. Additionally a bug tracker which plugs-in to subversion [TRA, 2009] was used in order to keep a list of bugs. Each bug was assigned (amongst other information) a severity, category and detailed description.

It is important to plan for disaster in any software development exercise. To ensure the success of the development of 3c, some measures were taken to avoid data loss. This was implemented in the form of 3 backup strategies. The first was inherent in the design of subversion; the code checked out by clients is effectively a backup. The second was a copy of the entire subversion repository data to another machine located in the same building as where primary development of 3c occurred. This was achieved with a piece of software called *rsync* [Tridgell and Mackerras, 2009], which efficiently makes a copy of data over a network, by only sending parts of the backup which have changed. This backup alone was not deemed enough because a fire or flood could still cause permanent data loss (if all aforementioned backups are in the same geographical area). To eliminate this single point of failure, a third automated backup was scheduled to compress and upload (via the secure shell [SSH, 2009] utility *scp*) the subversion repository to an off-site location each day.

4. 3c Design and Implementation

Having identified the initial direction of the project, 3c was designed and implemented using an iterative approach. This section describes the resulting final design of 3c. Any drastic divergences or compromises made relating to the design (at any iteration) are explained and are noted in the margin for emphasis.

4.1 System Overview

The 3c programming language is a pure object oriented, dynamically typed language which uses LLVM (version 2.4) for mid-level and back-end JIT functionality. The compiler front-end is written in the Python scripting language (version 2.6.1) and utilises the Python bindings for LLVM (version 0.5). A scripting language was chosen for the front-end due to time constraints, as implementing the same compiler in C or C++ would take significantly longer. The Python scripting language was specifically chosen because at the time of writing it had the most mature LLVM bindings (of the scripting languages) [R Mahadevan, 2009]. Instead of writing a custom tokeniser and parser, an existing open-source project was used, called Aperiot [Posse, 2009]. Figures 4.1 and 4.2 show the 3c sub-systems and class layout at a glance.

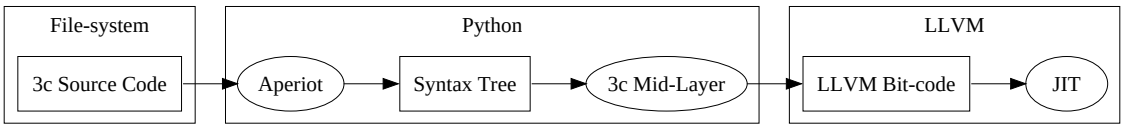


Figure 4.1: Diagram showing the sub-systems of 3c.

4.2 3c Source Code Design

3c source code is the input format of the 3c compiler. 3c source files are simple ASCII text files formatted to the rules of the 3c language grammar. It is conventional to name 3c source files with a `.3c` suffix, however it is not required. Section B.3 shows an overview of the 3c syntax. Each line may contain at most 1 statement terminated by a UNIX line feed. Lines whose first non-white-space character is a `#` are ignored (as comments). Indentation may be used, but is purely cosmetic, as all leading white-space is stripped prior to tokenisation.

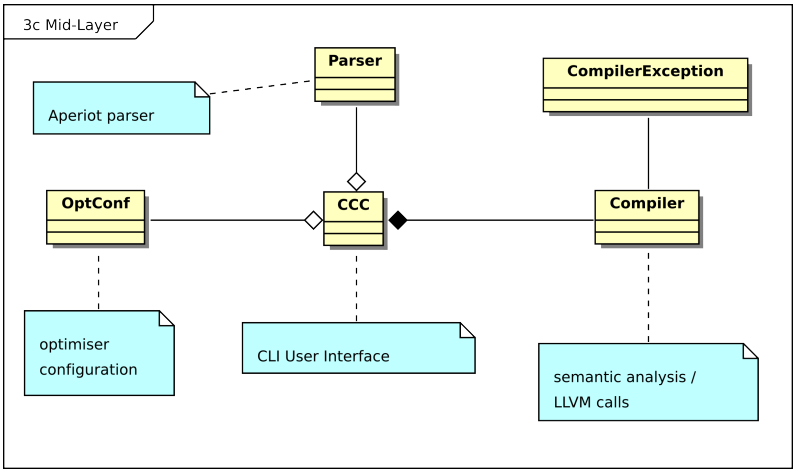


Figure 4.2: Finalised 3c mid-layer class diagram (Methods not shown for brevity).

4.3 Aperiot as a Tokeniser and Parser

In order to accelerate development a third party component was used for the purpose of tokenising and parsing. The *Aperiot* [Posse, 2009] parser can perform both of these tasks using a simple BNF-like syntax and the resulting output is imported natively into the user's Python code. Aperiot uses the LL(1) parsing algorithm, which is a top-down parsing method which uses the left-most derivation. One token of lookahead is used and no back-tracking is permitted (traditionally). Unfortunately the LL(1) parsing algorithm, can only parse a subset of context-free grammars. This limitation *did* impact the development of 3c and explains the "wordyness" of the 3c syntax. The Aperiot input for the 3c language is shown in listing 4.1.

Listing 4.1: Aperiot grammar for 3c

```
1  # 3c grammar
2  # $Id: ccc_parser.apr 210 2009-05-09 17:41:31Z edd $
3
4  #/=====================================================================
5  #Copyright (c) 2008-2009, Edd Barrett <eddbarrett@googlemail.com>
6  #
7  #Permission to use, copy, modify, and/or distribute this software for any
8  #purpose with or without fee is hereby granted, provided that the above
9  #copyright notice and this permission notice appear in all copies.
10 #
11 #THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
12 #WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
13 #MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
14 #ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
15 #WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
16 #ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
17 #OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
18 #\=====================================================================
19
```

```
20 import
21     glue
22     llvm.core
23
24 identifiers
25     varname
26     funcname
27     argname
28     classname
29
30 numbers
31     number
32
33 operators
34     plus  "+"           # polyop +
35     times "*"           # polyop *
36     minus "-"           # polyop -
37     div   "/"           # polyop /
38     asn   "="           # polyop =
39
40     mbracc "->"         # member accessor
41
42     cond_eq  "=="       # equality test
43     cond_neq "!="       # not equality test
44     cond_lt  "<"        # less than test
45     cond_gt  ">"        # greater than test
46     cond_lte "<="       # less than or equal test
47     cond_gte ">="       # greater than or equal test
48
49 brackets
50     lpar  "("
51     rpar  ")"
52
53 keywords
54     print        "print"
55     let          "let"
56     func         "func"
57     func_done    "func_done"
58     ret          "ret"
59     call         "call"
60     pass         "pass"
61     new          "new"
62     if           "if"
63     else         "else"
64     if_done      "if_done"
```

```
65      while        "while"
66      while_done   "while_done"
67
68  strings
69     string
70
71  start
72      INIT
73
74  rules
75
76  # start here
77  INIT -> pass         : "" # nothing
78        | print EXPR   : "comp.hook_print($2)"
79        | let ASSIGNABLE asn EXPR : "comp.hook_var_assign($2, $4)"
80        | func funcname lpar FUNCARGLIST rpar : "comp.hook_func_def($2)"
81        | ret EXPR     : "comp.hook_ret($2)"
82        | func_done    : "comp.hook_end_func()"
83        | if EXPR COND_OP EXPR    : "comp.hook_cond($2, $3, $4)"
84        | else         : "comp.hook_else()"
85        | if_done      : "comp.hook_end_cond()"
86        | while EXPR COND_OP EXPR : "comp.hook_while($2, $3, $4)"
87        | while_done   : "comp.hook_end_while()"
88        | EXPR         : "$1"
89
90  # expressions
91  # {
92  EXPR -> TERM plus EXPR   : "comp.hook_polyop($2, $1, $3)"
93        | TERM minus EXPR  : "comp.hook_polyop($2, $1, $3)"
94        | new classname lpar CALLARGLIST rpar : "comp.hook_class_inst($2)"
95        | TERM             : "$1"
96
97  TERM -> FACT             : "$1"
98        | FACT times TERM  : "comp.hook_polyop($2, $1, $3)"
99        | FACT div TERM    : "comp.hook_polyop($2, $1, $3)"
100       | CALL             : "$1"
101
102 FACT -> plus FACT        : "$2"
103       | minus FACT       : "comp.hook_negate($2)"
104       | lpar EXPR rpar   : "$2"
105       | PRIM             : "$1"
106 #}
107
108 CALL -> call funcname lpar CALLARGLIST rpar : "comp.hook_func_call($2)"
109       | call varname mbracc funcname lpar CALLARGLIST rpar : "comp.
```

```
          hook_member_call($4, $2)"
110
111 # argument lists
112 # annoyingly cant have expressions in call statements
113 # due to LL(1). conflicts with FUNCARGLIST
114 CALLARGLIST -> PRIM CALLARGLIST : "comp.hook_arg_push($1)"
115              | empty            : ""
116
117 # def needs its own rule that only takes argument name tokens
118 # for example "func my_func(1)" is invalid
119 FUNCARGLIST -> argname FUNCARGLIST  : "comp.hook_arg_push($1)"
120             | empty                 : ""
121
122 # primitive tokens
123 PRIM -> number           : "comp.hook_int($1)"
124       | varname          : "comp.hook_var_get($1)"
125       | string           : "comp.hook_string($1)"
126
127 # lhs of a 'let X = Y'
128 ASSIGNABLE -> varname        : "comp.hook_assignable($1)"
129 #             | member varname : "comp.hook_member_get($2)"
130
131 # conditional operator for if statements
132 COND_OP -> cond_eq    : "$1"
133          | cond_neq   : "$1"
134          | cond_lt    : "$1"
135          | cond_gt    : "$1"
136          | cond_lte   : "$1"
137          | cond_gte   : "$1"
```

4.3.1 LL(1) Limitations

One limitation of the LL(1) parsing algorithm is that it can not deal with left-recursion [Aho et al., 2007]. This posed little problem, as one can re-factor the grammar in such a way so as to eliminate the left-recursion, as shown in figure 4.3a.

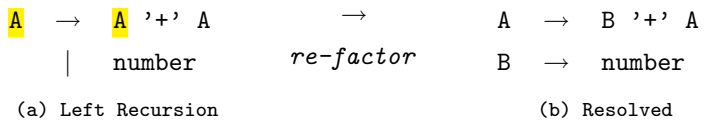


Figure 4.3: Re-factoring a left recursion grammar into one LL(1) can parse.

However, consider the grammar in figure 4.4a. An LL(1) parser can not parse this because it only looks at the first token, when deciding which rule to apply [Posse, 2007], hence the name LL(1);

The 1 is referred to as the look-ahead (the k) of the class of LL(k) parsers[1] . A technique called left factoring may be used to transform the grammar into a functionally identical grammar, which may be parsed by LL(1). Sometimes left-factoring will need be to applied several times before a grammar becomes LL(1) compatible.

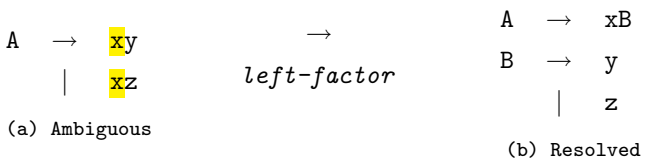


Figure 4.4: A grammar a conventional LL(1) parser can not parse, resolved with left-factoring.

Another way in which grammars can be transformed so as to conform to the limitations of LL(1), is by the process of *in-lining*. Using this technique, rules with only one candidate (*unit rules*), are made redundant, by using the single candidate directly. Figure 4.5a shows a new grammar which again, the LL(1) can not parse. By in-lining rules B and C, then left-factoring once, the grammar is transformed into a grammar LL(1) is happy to parse.

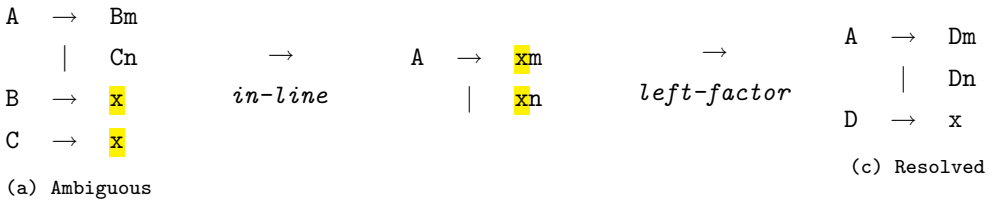


Figure 4.5: Using in-lining to resolve LL(1) conflicts.

Usually the grammar author is expected to manually make these adjustments. Aperiot does the transformations automatically in memory at run-time. As stated previously there is a fundamental issue with the LL(1) algorithm, which stems from grammars which simply can not be made non-ambiguous, even when using left-factoring and in-lining. Figure 4.6a shows such a grammar and an attempt to re-factor it, before reaching a final ambiguous grammar (x is ambiguous). LL(1) parsers can not parse ambiguous grammar [Aho et al., 2007].

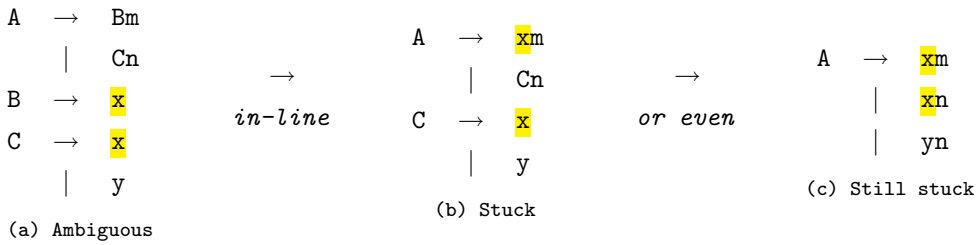


Figure 4.6: A grammar which can not be re-factored to adhere to LL(1).

[1] LL(k), with a $k >= 2$ would have no problem parsing this example grammar, as it would look at more than just the first token when making it's decision.

LL(1) Limitation Workaround

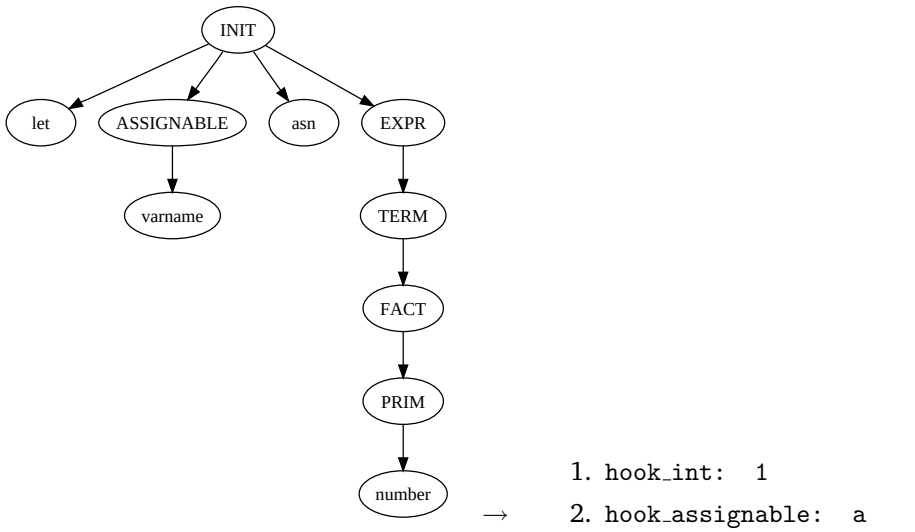

Design change

Having identified this as a problem with the initial grammar design for 3c, a simple but effective solution was devised. By inserting a unique string literal as the first token in composite rules, one can guarantee the grammar to be LL(1) compatible. So a variable declaration such as `a = 1`, became `let a = 1`, so that it would not conflict with the singleton statement `a` (which is a valid 3c expression).

4.4 The 3c Mid-Layer

The 3c mid-layer is the sub-system between the parser and the LLVM Python bindings, which does semantic analysis, state tracking and triggers bit-code synthesis for 3c. Like many parsers, Aperiot allows the implementer to associate grammar constructs with *parser actions*. A parser action is basically a small block of code, which is executed as the syntax tree is traversed after parsing is complete. The 3c compiler grammar uses this mechanism to assign calls to the methods of the mid-layer. Such methods in the context of 3c were named *parser hooks*. A parser hook deals specifically with one language construct and has a method name prefixed `hook_`, for example `hook_member_call()` deals with 3c member function calls.

A simple example is beneficial at this stage of discussion. Consider the simple example shown in figure 4.7. This figure shows the parse tree for the program `let a = 1` and the sequence of method calls upon the compiler mid-layer.



(a) Parse Tree

(b) Mid-layer Calls

Figure 4.7: A sequence of parser hook calls.

Each call to a parser hook may alter the *state* of the compiler, which may alter the behaviour of the compiler at a later date. The 3c compiler holds more state information than other compilers due to the decision to make the 3c parser a per-line based parser. This decision was made because an attempt to make an interactive shell like that of Python and Ruby was planned, not completed. Unfortunately this overcomplicated certain aspects of the compiler, particularly in dealing with conditionals and loops (more information in section 4.6.4). At the time of writing 3c holds the following state information:

A list of stack frames For each function call, a new stack frame is opened which will hold information such as the function name, a pointer to the function, argument names, local variable names and pointers to what is held in the variables. When a function return node is met, the stack frame is removed. Stack frames are essential to proper variable scoping. Without such mechanisms all variables will be global, making proper recursion very difficult or impossible.

An argument stack. Upon reaching argument nodes for a function argument list, the objects referenced are cached in the argument stack for use later. The reason for this is that the length of an argument list is never know prior to compilation, meaning one parser hook would be needed for each length of argument list. Needless to say this is both impractical and illogical. So instead when function call node is reached, the correct objects are be retrieved from the argument stack prior to synthesising the function call instructions. The argument stack is also used when synthesising function definition code, so as to copy the arguments onto the stack of the local function with the correct symbol table name.

A conditional/loop stack. Due to limitations in the line-by-line parsing approach of 3c, a stack of conditional and loop statements is maintained (the *C/L* stack). Each record in the stack contains pointers to each block within. For example a loop construct has a *condition check*, *loop body* and *loop exit* block. Each of these blocks will be needed later, for example when the end of a loop is reached and the compiler must direct program execution back to the *loop check* block to see if the loop runs again.

A type-sym table. A list of class definitions is held with information relating to the class attributes and methods, but most importantly a *type-sym*. A type-sym is a unique identifier integer for every type in 3c. Currently parser hooks do not manipulate this table, as user defined classes are not yet implemented.

3c does not alter the parse tree, before traversing it and applying actions, therefore no AST is involved in the compilation process. This was mainly due to time constraints. 3c does however perform limited semantic analysis and error trapping. 3c will abort compilation if the user attempts to instantiate a non-existent class or if an undefined variable is referenced. There are some other semantic-based error traps, which are implemented at run-time instead of at the mid-layer, due to various complications (See section 4.9.1).

4.5 3c Object Hierarchy

3c is a pure object oriented language, meaning that the user is only ever concerned with objects and there is no such thing as a "primitive type". There are 3 built-in classes: `Object`, `Integer` and `String`. The base class for the entire system is the `Object` class, which is the most generic type, holding only one field donating the object's type. Figure 4.8 shows the class hierarchy in 3c. One might ask why a type is held, given that the object is of type `Object`. This is because it is necessary to define a "lowest common denominator" interface to function calls within the 3c byte-code. This is achieved by casting higher level types down to generic `Object` when calling methods and functions. This is required if inheritance is to work properly.

Consider for example, an imaginary class `A`, which is sub-classed by another imaginary class `B`. Now `A` defines a method, which is to be inherited by `B`. The first argument of any method internally is a pointer to the object which is being operated on. In 3c this detail is hidden from the user, but in some languages such as Python, it is not[2]. Although 3c appears to be dynamically typed, LLVM assembler is *not*. This means that if `B` is to inherit this method, `A` must internally provide a version of the method which accepts the type of `B` as the pointer type to it's self. Such a solution is not a good one, as this would mean `A` would need to have prior knowledge of every class which might

[2] `def my_method(self, arg1, ...)`

sub-class it. Instead, all arguments are casted down down to `Object` and within the function casted back up to either a known type, or the type indicated by the type field of the `Object` class.

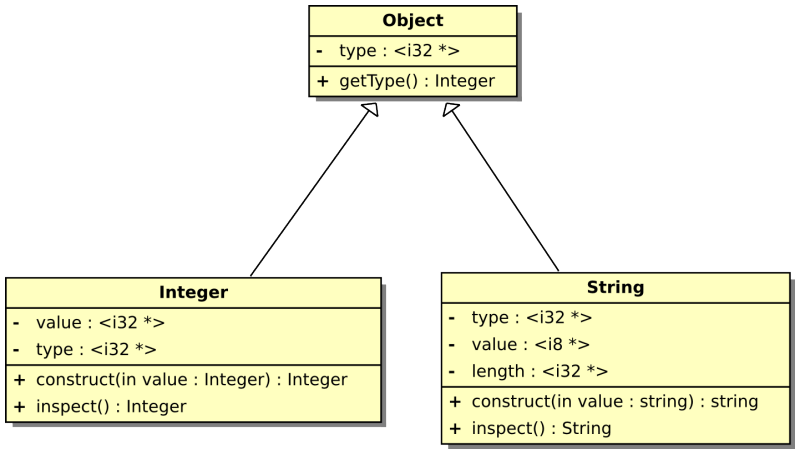


Figure 4.8: 3c Class hierarchy (internal methods are hidden).

All references to object instantiations are stored internally as "double pointered" LLVM structure types[3]. There is a reason for using two pointers, which is discussed in section 4.6.4. Each field within an instantiation is a pointer to some data. In the case of `Object`, (as mentioned previously) there is just a pointer to an integer indicating the type of the object. Figure 4.9 shows this structure visually.

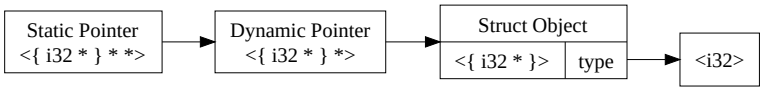


Figure 4.9: An `Object` instantiation

4.6 Basic Functionality

In this section each language construct is studied in detail, identifying how each works internally, how they alter the compiler state and any potential problems which were encountered during development.

4.6.1 Constructing Built-in Types

The user may instantiate the built-in classes in 2 ways:

1. Using the `new` keyword, eg. `let a = new Integer(1).`
2. By using literals in 3c source code, eg. `let a = 1` (strings and integers only).

The two forms are functionally equivalent and allocate instances on the heap, placing a reference within the local function scope. The first form is somewhat less efficient, as two instances are created, one by the literal as the constructor argument, and one created by the copy constructor.

[3]Pointer to pointer to structure.

The generic `Object` may be instantiated, but is not functionally useful. It may be used as a void type.

4.6.2 Printing Values

The `print` statement is used to print an instance. In the case of `Integer` and `String` instances, the *value* field of the object is printed to standard output, followed by a UNIX line feed. Generic `Object` may not be printed. Internally the print statement calls the `__print` method of the specified instance via the virtual function table and calls libc `printf(3)`.

4.6.3 Variable Assignment

The `let` statement is used to assign variables, eg. `let a = 1`. If the variable is undefined, it is automatically defined in the symbol table of the current stack frame, before being assigned. Assignment comprises of copying the dynamic pointer of the value into that of the variable being assigned.

4.6.4 Conditionals and Looping

The user may conditionally execute blocks of code using the `if` statement. This statement comes in two forms; with a single block or with two mutually exclusive blocks. Conditionals statements may only compare operands of the same type. Internally conditional constructs are implemented using IR *conditional branching*, much similar to microprocessor assembler code. `while` loops may be used in 3c (fig. 4.12). The loop condition is checked prior to entering the loop.

```
if <expression> <conditional_operator> <expression>
...
if_done
```

Figure 4.10: Conditional - Single block form.

```
if <expression> <conditional_operator> <expression>
...
else
...
if_done
```

Figure 4.11: Conditional - 2 Block form.

```
while <expression> <comparison_operator> <expression>
...
while_done
```

Figure 4.12: While loop construct.

Looping caused considerable complications due to the decision to the way variables were referenced and assigned. Initially all object instances were *not* referenced via a double pointer, but instead by a single pointer. Later it was noted that such an approach would lead to infinite loops, just as long as the loop body was entered once. Consider the variable x in the program shown in listing 4.2. Using the flawed representation, the variable is initially stored as a single pointer reference $x \rightarrow 10$. The loop body would enter, x is printed, and then execution arrives at the re-assignment of x. During bit-code synthesis, the mid-layer had over-written it's symbol table reference to x with the new instance resulting from *x - 1*. Upon reaching the end of the loop body the JIT engine branched back to the loop check and checked the *old* instance instead of the *new* instance, leading to an infinite loop. The crux of the problem is that the loop check reference can only be a static pointer value.

Design change

In order to overcome this a "double pointered" variable referencing approach was adopted. Consider instead the variable x is initially represented as $x_s \rightarrow x_d \rightarrow 10$, where x_s is the *static variable pointer* and x_d is the *dynamic variable pointer*, which is over-written upon re-assignment of the variable. Variable look-ups are then modified to dereference 2 pointers before retrieving instance fields. Via this mechanism, it no longer matters that the loop check construct dereferences a static pointer, as the following pointer is dynamic, allowing, in this example a different integer value to be compared upon each iteration of the loop. The loop then exits correctly after 10 iterations.

Listing 4.2: Loop case study program.

```
1 let x = 10
2 while x > 0
3     print x
4     let x = x - 1
5 while_done
```

Design change

A second complication of looping was related to the line-by-line parsing approach of 3c. This approached prevented parser hooks from receiving code-blocks as operands and instead conditional and loop constructs must be cached in the mid layer, holding pointers to each block, correctly switching IR insertion and terminating nested loops and conditionals properly.

Consider a conditional statement `if a == 1 ... else ... if_done`. This is represented in IR as shown by in listing 4.3. This simple example has 4 blocks: one checks which branch of the loop to jump to (`cond_check`), one for each branch body (`cond_true` and `cond_false`) and finally an exit block (`cond_exit`), which is jumped to at the end of both branches of the conditional. In the initial implementation the mid-layer simply updated it's `__current_builder` state attribute upon reaching an `if`, `else` or `if_done` statement, effectively resuming subsequent IR generation at the right place in the construct, terminating the relevant labels as it goes.

Listing 4.3: A simple conditional IR representation

```
1  cond_check:
2          %1 = icmp eq %a, i32 1
3          br %1, label %cond_true, label %cond_false
4  cond_true:
5          ...
6          br label %cond_exit
7  cond_false:
8          ...
9          br label %cond_exit
10 cond_exit:
```

This seems simple, but watch what happens if no further logic is implemented and a while loop (`while b < 100`) is nested inside the true branch of the a conditional (listing 4.4). This is invalid IR because: a) The `cond_true` block is double terminated, b) the `loop_exit` block is not terminated. If this code were to be validated before execution, LLVM would abort due to this. It becomes clear from the previous example, that a loop or conditional block needs knowledge of nested (child) loops and conditionals in order to correctly place branching statements. The separate conditional and loop stacks were merged into a single one called the *conditional/loop stack* (or the *C/L stack*)[4]. No conditional branches are terminated until the entire construct is complete, and terminators are only added if the branches were not previously terminated by nested conditionals or loops. Logic is then added to continue parent IR generation at the last child's `exit` block.

Listing 4.4: A broken nested loop representation.

```
cond_check:
        %1 = icmp eq %a, i32 1
        br %1, label %true, label %false
cond_true:
        br label %loop_check
        br label %cond_exit
loop_check:
        %2 = icmp slt, %b, i32 100
        br %2, label %loop_body, label %loop_exit
loop_body:
        ...
        br label %check
loop_exit:
cond_false:
        ...
        br label %exit
cond_exit:
```

4.6.5 Functions

3c supports the use of functions, which come in two forms: *plain* functions and *member* functions (or methods). The user may not define member functions, as no interface for creating user classes is yet implemented. Plain functions are defined using the `func` statement as demonstrated in listing 4.5. An argument list in parentheses is required, but may be empty. Arguments within an argument list are separated by a space character. Arguments are *copied* on to the local function stack and are scoped locally to the function body, meaning that duplicate variable names may be used in different scopes, with no fear of them conflicting. All functions *must* be terminated by a `ret` statement. Failing to return from a function will cause compilation to be aborted by the mid-layer, which performs static code analysis to detect such errors.

Listing 4.5: A sample function declaration.

```
func my_function(my_arg)
        print "the arg is" + my_arg
        ret 0
```

[4] In the sources `__cl_stack` in the `Compiler` class.

```
4 func_done
```

The calling of functions is achieved through the `call` statement, which takes two forms, one for plain functions and one for member functions (listing 4.6). Plain functions are directly called, whereas member function calls are despatched via the use of a *virtual function table* (or *v-table*) in order to achieve polymorphism. This table is explained in depth in section 4.7.

Listing 4.6: Calling a plain function and a member function (method).

```
1 call my_function(arg)
2 call my_object->my_method()
```

4.7 IR Tables

A lot of the functionality of 3c was implemented in the Python mid-layer because the outcome of the operation is known at compile time, for example when the user requests a new `Integer`, the type is known and code can be easily statically synthesised to call the correct constructor. In other parts of compilation on the other hand, the outcome will greatly depend upon details which are unknown until run-time. For example when you call a member function of a variable, which implementation of the member function should be executed? It depends upon what type the variable is representing, which will not be known in the mid-layer.

In the early stages of 3c development, it became very clear that a lot of information about classes and methods would need to be available at run-time during JIT. Because LLVM is programmed in an assembler language, such information must be accessible via pointer arithmetic (via the `gep` and `load` instructions). The most suitable way to achieve this was to store a number of tables at the bit-code level, along with some routines to manipulate and search them. 3 types of table were devised: the *type-sym table*, the *virtual function table* (or *v-table*) and the *polymorphic operator table* (or *polyop table*). Throughout the rest of this document, the short notations will be used to refer to these tables.

4.7.1 The Type-Sym Table

The type-sym table is a cache of the class types present in the compiler and pointers to other per-class tables. There is one type-sym table global to the entire compiler, which in the source code is referred to as the `__type_symtab`. Figure 4.13 shows the structure of a record of the type-sym table. Recall that each type in 3c has two identifiers. One is an English name which is easy for the programmer to remember and another is an integer, called a *type-sym*. The position in the table implies the type-sym, so the first type in the table will have a type-sym of zero. Pointers to the type's name, v-table, v-table length and polyop tables are held.

< [i8 x 64] * > Ptr to type name	< [16 x { [i8 x 128], i8 *}] * > Ptr to v-table	< i32 * > Ptr to v-table Length	< [4 x [16 x i8 *]] * > Ptr to polyop tabs

Figure 4.13: The structure of a type-sym table record.

4.7.2 The Virtual Function Table

The v-table allows the compiler to provide a facility which selects and despatches the correct implementation of a member function, with respect to the rules of inheritance. This is required because the type of an object, which is having a call placed upon it, is not known until run-time. Each class has one v-table. Figure 4.14 shows the structure of a v-table record.

During compiler initialisation, upon processing each member function declaration within a class, a record is added to the v-table for the type in question. A record consists of a mangled function name and a void function pointer, which points to the newly defined method. The mangled function name is the all important part of this sub-system, as it has two purposes:

1. Identifying the correct IR function.
   Upon calling a member function of an instance, the function name being called is mangled by encoding the number of arguments into the function name. A function called `inspect` with 2 outfacing arguments would be mangled to `inspect[2]`. An IR routine is then called[5] which loops over the length of the table attempting to find a record of the same name, before either despatching the function, or returning an error. Bear in mind that in a dynamically typed language such as 3c, overloading of functions with the same number of arguments is impossible, as argument types are not disclosed in the function declaration.

2. Re-casting the function pointer.
   A void pointer alone is not callable, because the return type, argument type(s) and number of arguments are not known. Before 3c can call the desired function, the void pointer[6] must be *bit-casted* back up to it's original implementation. Because 3c uses a generic `Object` during member function despatch the correct function signature can easily be derived. Each function call will always return `Object` and will have the same number of arguments (also as `Objects`) as in the square brackets of the mangled name. Once the void pointer has been casted to the correct pointer type, the function may be called. This is cumbersome but necessary as each v-table record must be of identical type (void pointer) to be valid for storage.

< [128 x i8] >	< i8 * >
Mangled Function Name	Void Function Ptr

Figure 4.14: The Virtual Function Table Record Structure.

4.7.3 Polymorphic Operator Tables

3c uses a polymorphism concept called *operator overloading* for any expression which contains addition, subtraction, multiplication or division operators. In 3c these operators are called *polymorphic operators*. Such an approach allows a user-definable way of applying mathematical operators to objects of different types. For example, what should happen when the user executes a statement such as `print "my age is: " + 2`? Logically this is impossible in many computer languages, including LLVM assembler. However by using a polyop table, 3c achieves more user-expectable results from adding objects of different types.

The operator overloading system of 3c was heavily influenced by that of the C++ programming language. Listing 4.7 shows a C++ program which implements operator overloading. The way

[5] __vtab_lookup().

[6] < i8 * > is a void pointer in LLVM IR.

C++ achieves this is via methods named specially with an `operator` prefix followed by an operator symbol. The argument type then denotes the type which is being added to the type of the current class. 3c closely emulates the C++ naming convention for operator overloading, but due to differences in typing systems is unable to use argument types as an identifier. C++ is a statically typed programming language, whereas 3c is dynamically typed and so the compiler will never know the types of the arguments of a member function until run-time. To overcome this, the member function name has further information encoded inside it, as shown in figure 4.15[7].

```
int A::operator+(int other)
```
(a) C++ Operator Overloading

→

```
func oper+int[1](other)
```
(b) 3c Operator Overloading

Figure 4.15: The 3c polyop encoding scheme.

Listing 4.7: A C++ program demonstrating operator overloading.

```
#include <iostream>

class A {
        public:
                A(int);
                int operator+(int);
                int num;
};

A::A(int newNum) {
        num = newNum;
}

int A::operator+(int other) {
        int i = num + other;
        return i;
}

int main(void) {
        A a1 = A(666);
        int result = a1 + 1;

        std::cout << result << std::endl;

        return 0;
}
```

The mid-layer of 3c builds an IR routine on the fly (prior to JIT execution), which is called at the beginning of the `main` IR function at run-time (prior to any user-program execution). This routine loads any 3c methods which appear to be operator overloading semantics into a a polyop table structure. An example polyop table is shown in figure 4.16. Each class in the system has one

[7]This is a direct translation, and the types do not exist in 3c.

polyop table, each comprising of 4 rows, one for each operator and a number of columns, one for each type in the system, indexed by type-sym. The table is essentially a matrix of void function pointers to member functions which specialise in performing various operator overloading tasks. The pointer may be re-casted (in much the same way as in the v-table) and called. The returned object will contain the result of the operation. There is a special case where the pointer may be null, indicating that the operation is invalid, like in the case of subtracting a string from a number. Such an operation makes no sense, so compilation is aborted. The routine that performs the lookup is the `__polyop()` method of the generic `Object` class, which is therefore inherited by every class in the system.

	type-sym 0	type-sym 1	type-sym 2	...
polyop +	< i8 * >	< i8 * >	< i8 * >	...
polyop -	< i8 * >	< i8 * >	< i8 * >	...
polyop *	< i8 * >	< i8 * >	< i8 * >	...
polyop /	< i8 * >	< i8 * >	< i8 * >	...

Figure 4.16: The structure of a polymorphic operator table.

The built-in classes use this mechanism thoroughly, even for operating upon instances of the same type and although user-classes are not implemented yet, when (and if) they are, the existing codebase will dynamically accommodate this feature. All the user would need to do is define member functions adhering to the naming convention discussed above.

4.7.4 Example Table Usage

To clarify the function of the IR tables, a simple example can be presented which makes use of all three tables. Take the 3c program shown in listing 4.8 as a case study.

Listing 4.8: Table test program.

```
1 let a = 42 + " is the meaning of life"
2 call a->inspect()
```

The flow of execution is as follows:

1. The `Integer` class constructor is called in order to create an instance of `42`.
2. The `String` class constructor is called in order to create an instance of `"is the meaning of life"`.
3. The `__polyop()` method of `42` (inherited from `Object`) is called, the first argument is an operator identifier of `<i32 0>` (for plus[8]) and the second argument is the operand instance of `" is the meaning of life"`.
4. The type-sym attribute of the instance being operated on (`42`) is extracted. For an integer this is 1.
5. The `Integer` type-sym record is extracted from record number 1 of the type-sym table, as determined by the last step.
6. The pointer to the polyop table for this class (`Integer`) is extracted from the type-sym record.

[8] 0: +, 1: -, 2: *, 3: /

7. The record corresponding to the operator offset is extracted from the type polyop table, in this case 0 for plus.
8. The type-sym attribute is of the operand instance (`" is the meaning of life"`) is extracted. For a `String`, the type-sym is 2.
9. Void function pointer number 2 of the polyop record (from stage 7) is extracted. This will be a pointer to the `oper+String[1]()` member function of the `Integer` class. The pointer was placed there under instruction of the mid-layer by an IR routine called `__add_builtin_vtabrecs()`, which was called in 3c's `main()` prior to any user code.
10. The pointer is re-casted and called.
11. Execution is handed off to the `oper+String[1]()` member function of the `Integer` class, where a new string is constructed and returned: `"42 is the meaning of life"`. Internally this is achieved by call to libc `snprintf()`.
12. In turn `__polyop()` passes directly back the new string instance to the caller, therefore completing the operator overloading section of execution.
13. The result of the polyop operation is assigned to the variable `a`.
14. The name of the member function `inspect()` is mangled to `inspect[0]` so that it conforms the v-table encoding scheme.
15. The IR routine `__vtab_lookup()` is called with the mangled name and class' type-sym as arguments.
16. Execution is handed to `__vtab_lookup()` and the type-sym attribute of the instance being called upon (the new concatenated `string`) is extracted. For a string this is 2.
17. The type-sym table is consulted again, and the record at index 2 (for `String`) is extracted.
18. The pointer to the v-table for this class is extracted from the type-sym record.
19. The v-table for the class is now searched for the record corresponding to the requested method name in mangled form.
20. The correct record is found and the void function pointer (in this case to the `String` class' `inspect()` function) is extracted and returned.
21. Finally, the void function pointer is re-casted and called.

4.8 Optimisation

The 3c compiler may optionally apply LLVM optimiser passes to the internal representation of the program prior to JIT execution. Optimiser passes are applied within the 3c mid-layer under the instruction of an *optimiser configuration*. If any passes are enabled, they are applied one-by-one on the in memory bit-code. LLVM has the option to apply optimisations on a per-function basis, but this feature was not used in 3c and the whole IR module is transformed by the optimiser instead.

The configuration file is stored in a hidden file in the user's home directory (*~/.3crc*). If this file is non-existent when 3c executes, a blank configuration with no passes enabled is created. This configuration may be edited using either a text editor like *vi* or via 3c's built-in optimisation configuration editor. The built-in editor is invoked by running 3c with the `-c` switch. Using the built-in editor has the advantage that the user need not look up the LLVM pass number, as symbolic names are displayed during editing. The file itself is a simple list of pass numbers, one per line.

4.9 JIT

The last thing the mid-layer has the option to do is execute the completed IR module via JIT compilation. A "dry run" may be performed with the `-d` flag, which causes 3c to only parse and synthesise before aborting instead of executing the module. Another useful flag `-b` causes 3c to dis-assemble and dump it's bit-code into an LLVM assembler file on the disk. Unless the `-o flag` is used with `-d`, the user is prompted for the file name of the IR dump prior to JIT. More information on these flags is available in section B.2.

4.9.1 Run-Time Errors

Although compilers attempt to detect as many errors as they can, some errors can not be detected prior to execution. This statement especially holds true for dynamically typed languages such as 3c, where little is known about the type of variables at compile time. Due to this a number of routines are implemented in IR, which check for semantic errors during JIT execution. In such cases, 3c is able to detect and abort execution, preventing undefined behaviour. The following cases cause execution abortion:

Bad comparisons. 3c can only compare instances of the same type, so comparing a `String` instance to an `Integer` instance will result in an error.

Inapplicable polymorphic operations. If semantics for a polymorphism operation are not implemented, for example, `666 - "some string"`, the compiler will exit.

Division by zero. Division by zero is an impossible mathematical operation. Attempting to perform this would have caused LLVM to segmentation fault, so a suitable error message is displayed and compilation is aborted instead.

Calls to invalid member functions. If no v-table record exists for a member function, the user is requesting a non-existent member function, so 3c aborts.

5. 3c in Practice - System Testing and Evaluation

Exhaustive testing is impossible. This is an assumption made. because: A) Developers can never test a piece of software on every end user's software and hardware configuration. B) Often it is very difficult to test every sequence of user input and C) Computer programs are not always deterministic [Barrett, 2009]. For this reason software engineers have developed testing techniques which aim to capture a large subset of software faults in a shorter time as possible. Derived from these techniques, a set of system test cases were written in order to validate the behaviour of the compiler is correct. Quick tests were also run ad-hoc after each iteration of the spiral model, checking that the new functionality of each iteration had not broken any previous functionality.

5.1 Test Cases

5.1.1 Boundary Value Analysis Tests

Boundary value analysis is a form of black-box testing which builds upon the concept of *equivalence partitioning* [Myers, 1979] [Roper, 1994]. A *partition* can be defined as a range of inputs where the output is *likely* to be the same. By testing upper and lower the boundaries of a partition, the tester assumes that all other values of the partition have also been tested. Although the approach is limited, as combinational inputs may invalidate the assumption, it is the author's opinion that it does lend it's self particularly well to testing conditional statements. A set of boundary value analysis tests were devised in order to validate the behaviour of conditionals within the 3c source code implementation. The test cases and their outcomes are shown in appendix C.2. The same logic is used for loops, so it was assumed (due to time constraints) that the tests need not be repeated.

Every test apart from one succeeded, revealing a fault in the logic of the inequality operator. After inspection, a trivial error was found, which related to the change to the double-pointered instance representation. A `bitcast` instruction was casting a pointer to the old single-pointered instance representation type. A fix was devised (listing 5.1) and regression tests were executed, to ensure the fix had not introduced further software faults. All regression tests passed.

Listing 5.1: Unified diff showing the inequality operator bug fix.

```
1 Index: ccc_compiler.py
2 ===================================================================
3 --- ccc_compiler.py     (revision 235)
4 +++ ccc_compiler.py     (working copy)
5 @@ -2104,7 +2104,7 @@
6                 eq = m.get_function_named(mang)
```

```
7
8                   ret = b.call(eq, [ func.args[0], func.args[1] ])
9   -               ret_i = b.bitcast(ret, Type.pointer( \
10  +               ret_i = b.bitcast(b.load(ret), Type.pointer( \
11                                  self.__mk_object_struct("Integer")))
12
13                  # get the int out
```

5.1.2 Fibonacci Sequence

The Fibonacci sequence is a good test for compilers for two reasons. Firstly it confirms that the compiler is able to synthesise recursive code properly without variables of the same name clashing between stack frames. Secondly running Fibonacci on large numbers provides a suitable length of execution time to be used as a performance benchmark. A series of tests were run to confirm that stack frames and recursion were properly implemented by the 3c compiler. The fib program shown in listing C.2 was run 10 times, both with and without optimisation enabled. The optimised tests had a number of LLVM transforms applied, which unrolled loops, in-lined functions, combined statements, marked functions internal and eliminated dead code (see optimiser configuration in section C.1). Out of curiosity, byte-code and native code were dumped to disk using the `-d` option of 3c and tested in a similar manner. The byte-code was planned to be executed in a purely interpreted (no JIT) fashion using the `lli` utility, but unfortunately the interpreter functionality of LLVM-2.4 is incomplete and was unable to interpret the code[1]. The binary was made by using the `llc` utility following by using GCC (with the optimiser *off*) to assemble a binary. The size of the code generated by the program and the execution times were also noted. Similar programs were implemented in Lua (listing C.4) and Java (listing C.3) so as to provide a performance comparison. The results are shown in tabular and graphical form in figure C.3.2. The test passed, but there appears to be some performance issues, which are discussed in section 5.2.3.

5.1.3 Nesting Test

As described earlier, the CL (conditional/loop) stack had become somewhat over-complicated and it was identified as an area likely to contain software faults. For this reason, a nesting test was devised (listing C.5). This test nests loops and conditionals in a function at varying depths. No software faults were detected with this test.

5.2 Evaluation

At this stage, 3c has been developed from an idea, through 15 iterations of a spiral development methodology, before becoming mature enough for evaluation. In this section, iteration 15 of the 3c compiler is critically evaluated, highlighting both good aspects and shortcomings of the 3c implementation and development process.

[1] ERROR: Constant unimplemented for type: [7 x i8]

5.2.1 Evaluation of Development Technique

As expected, the iterative development life-cycle offered by the spiral model fitted the nature of the project very well. In total 15 iterations occurred (fig. 5.1), each adding a single feature or set of related features. Iterations 2 and 11 were not finished, as the development priorities were re-prioritised. Phase 2 involved adding a second pass of the parse tree, which would have allowed a program to call a function not yet defined until later in the 3c source code, for example. This feature was buggy and is purely a nicety, so it was delayed with the intention of re-visiting it later. Iteration 11 was user-classes and was also aborted due to time constraints. Further development would have a) allowed less time for system testing and documentation and b) possibly introduced software faults, which may not have been detected in the (now shorter) testing time allocation. It is the author's view that it is better to release a bug-free product than one including *new feature x*, which has not been tested properly and probably contains software faults. The spiral methodology provided enough flexibility to re-work the development cycle, whereas other methodologies which required a solid design decision would not have accommodated this development flow. For this reason the spiral model would certainly be used again for a research based project. The disaster recovery measures described in section 3.2 were not required, as no data loss was incurred. The bug tracking system was used fairly minimally, as most bugs were fixed as soon as they were encountered.

1. Basic numeric calculations.
2. Second pass on the parse tree (disabled).
3. Variables with global scope.
4. Functions (with no arguments).
5. Function stack frames and local variables.
6. Function arguments.
7. Object hierarchy and v-table.
8. User constructors and Integer Class replaces i32.
9. String Class.
10. Operator overloading polymorphism.
11. User classes (aborted).
12. Looping and conditional statements.
13. Double-pointered object references.
14. Re-integrate flat functions.
15. Optimiser functionality.

Figure 5.1: Iterations of 3c.

The subversion source code management system provided an invaluable record of changes to the 3c source-base. As expected *branching* accommodated the iterative nature of the project. Inspection of the `phases` folder within the project distribution will confirm this. Some iterations have intermediate branches, which were used when the task at hand was becoming large. On several occasions changes had introduced bugs and a question arose; What has changed which could be affecting this? In such a case the *diff* function of subversion was able to show which parts of the program had changed and in most cases the bug could be traced quickly.

5.2.2 Evaluation of Design and Implementation

Most of the design choices made throughout the development cycle of the project were well thought through and proven in practise, however as mentioned previously a couple of bad design choices made in early iterations stunted development in the late stages of implementation.

Starting with good aspects of the system, it is believed that Python greatly accelerated the development of the project. The list and dictionary data structures of Python were particularly well suited to representing the various stacks and their corresponding records within the mid-layer. By comparison, the Python data structures provide much higher level operations than the C++ `vector` class does, which would have been used otherwise. Python is also in a state of version limbo. Python 3.0 has been released and breaks backward compatibility of Python 2.*x*. Software projects have either quickly adopted version 3, or are holding off until more projects start using version 3. Luckily all Python components used in 3c, were of the latter disposition and no incompatibility issues were encountered. The only minor quibble with using Python and llvm-py for 3c was that the host system will require LLVM, Python and the bindings at all times in order to run 3c programs. If C++ were used, a standalone static binary could be built, which has no runtime dependencies.

LLVM lived up to it's expectations, providing a JIT execution environment, strict typing and good bit-code verification. This allowed a large amount of programming faults to be discovered prior to JIT execution. The developed system, although untested on platforms other than OpenBSD, should be very portable thanks to LLVM's development model. There were however a few minor criticisms, which should be noted. When developing a LLVM module, the author advises the developer to be very careful with the `bitcast` instruction. This instruction casts one pointer type to another and was used for type conversion within 3c. This instruction is quite dangerous, but necessary, as it will do *exactly* as it is told. If LLVM is told to bitcast a string pointer to an integer pointer, it will do so, possibly resulting in erroneous behaviour. This was a common method of introducing faults during 3c development. It is advised `bitcast` operations are carefully checked. It is also questionable as to whether the typing system of LLVM is too strict. Do arrays of different lengths *really* need to be seen as of different type? In many cases strings (character arrays), were casted to different lengths, purely to satisfy LLVM's type checker. Whilst this is good in a way because it causes the programmer to think carefully about the design of his/her types, it also encourages further use of the perilous `bitcast` operation.

LLVM's static analysis of bit-code appears to be very capable. It was noted that a 3c was unable to return an instance from a function, which is always defined, but within either the true or false branch of a conditional statement (and not outside the conditional). By commenting line 6 of the Fibonacci test, one can reproduce this behaviour (listing 5.2). The instance `result` is always defined, but the LLVM verifier throws an error (listing 5.3). This is because if the mid-layer symbol table does not know about the `result` variable prior to the conditional statement, then it can not know about the different registers which `result` is assigned to in each branch of the conditional statement. The first version of `result` is the one returned and LLVM successfully realises that the second can not be returned at-all, aborting compilation.

Listing 5.2: Commenting line 6 of the Fibonacci test

```
1 #!/usr/bin/env 3c
2 # Fibonacci number generator
3 # $Id: fib2-sa.3c 264 2009-05-20 14:21:56Z edd $
4
5 func fib(n)
6 #       let result = 0
7
```

```
8         if n <= 1
9                 let result = n
10        else
11                let n1 = n - 1
12                let n2 = n - 2
13                let result = call fib(n1) + call fib(n2)
14        if_done
15
16        ret result
17 func_done
18
19 let loop = 0
20 let out = ""
21
22 while loop < 25
23        let out = out + call fib(loop) + " "
24        let loop = loop + 1
25 while_done
26
27 print "Your Fibonacci numbers:"
28 print out
```

Listing 5.3: Error code resulting from commenting line 6 of the Fibonacci test

```
1  Traceback (most recent call last):
2    File "/home/edd/proj/3c/phases/final/3c", line 276, in <module>
3      ccc.start()
4    File "/home/edd/proj/3c/phases/final/3c", line 161, in start
5      self.__jit()
6    File "/home/edd/proj/3c/phases/final/3c", line 177, in __jit
7      self.__dump_bc, self.__dump_file)
8    File "/home/edd/proj/3c/phases/phase-15/ccc_compiler.py", line 690, in
       execute
9      self.__mod.verify()
10   File "/home/edd/proj/python-2.6.1/lib/python2.6/site-packages/llvm/core.
       py", line 947, in verify
11     raise llvm.LLVMException, ret
12 llvm.LLVMException: Instruction does not dominate all uses!
13        %46 = malloc { i32* }*          ; <{ i32* }**> [#uses=2]
14        store { i32* }* %49, { i32* }** %46
15 Instruction does not dominate all uses!
16        %46 = malloc { i32* }*          ; <{ i32* }**> [#uses=2]
17        ret { i32* }** %46
18 Broken module found, compilation terminated.
```

The first *serious* design flaw of 3c, was the decision of parser. The Aperiot parser was chosen because it was easy to integrate with the other components of the project. Aperiot is written in

pure Python and so was easy to install, it has a grammar description syntax not unfamiliar to the well understood and documented yacc, meaning that there was little learning needed to use it as a part of 3c. In hindsight, more time should have been spent inspecting parsing algorithms. Perhaps the serious limitations of the LL(1) method would have become apparent *before* the parser was integrated. As a result, the grammar of 3c was slightly modified, but not too drastically, still clear and simple, just somewhat wordy. Having criticised Aperiot, it is worth mentioning that as far as LL(1) parsers go, it is a very good implementation and it probably would be used again, perhaps just for parsing simple configuration files and not programming languages.

The "double pointered" instance representation was over-looked. This was due a C programming concept which was wrongly assumed to be present in LLVM assembler. In C the `&` operator may be used to get the memory address of a variable (listing 5.4). If such a function existed in LLVM, then a single pointered instance representation would have sufficed, however there is an important realisation which was not made as to why this can not be possible in LLVM; the memory address of a register can not be obtained. Such an operation is just inherently impossible in the design of a virtual machine (and in a real machine). There is a workaround and it is the same one which was described earlier in this document (section 4.6.4), which is to provide a pointer with a known address, within which storing a pointer which can change therefore making a double pointer arrangement. Modifying the whole compiler to reflect this change was both time consuming and introduced bugs, one of which went un-noticed until system testing (section 5.1.1), so this was a lesson well learned.

Listing 5.4: Use of the ampersand operator in C.

```
1 #include <stdio.h>
2
3 int main(void) {
4         int a =1;
5         printf("'a' is stored at 0x%x\n", &a);
6
7         return 0;
8 }
```

5.2.3 Evaluation of Testing

The testing strategy was not as formalised as it should have been, but was present and effective. As planned the end of each iteration included various tests, which confirmed that the work in the most recent implementation had not broken a previous iteration's features. It was not uncommon to find breakage in the code base at this stage, in which case the faults were investigated and resolved, before regression testing was performed. These tests could and should have been implemented in software in the form of unit tests or via an external test framework like DejaGnu [DEJ, 2009].

The documented system testing revealed that as expected, JIT execution slower than native execution. Unfortunately no comparison to interpreted 3c byte-codes could be obtained, but it is thought that it would be much slower than with JIT. Oddly the optimised version of the fib program was marginally slower than the non-optimised version, which was thought to be because the optimiser itself takes a little time to run, but this does not account for why the optimised native binary is slower, as optimisation took place at compilation time not run-time. More research is needed in this area.

The performance of 3c seems rather sluggish in terms of both start-up time and in execution time. Lua and Java were found to outperform a JIT executed, speed optimised 3c `fib(25)` program

by an average of 27 times. The start-up time of 3c is related to the JIT engine building newly encountered code paths for the first time. This can be confirmed by writing a program which immediately uses the `print` statement and then timing program execution using both JIT and native execution. When run using JIT, the program will take about 6 seconds to execute the initial `print` statement, whereas when running a native binary made from an assembler dump, the `print` statement is executed almost immediately.

Start-up time can not account for why the Fibonacci test took as long as it did for 3c. Less the 6 second start-up time of the JIT engine, an optimised `fib(25)` program takes on average 11.206 seconds. The initial thought regarding this was that the string comparisons of v-table lookups were slow and inefficient. A quick experiment was run in an attempt to confirm this. The vast majority of the v-table look-ups of the fib program are for the `<=` comparison operator and this only occurs in the fib program for integers. The `member_call()` method within the mid-layer was modified to directly call the `__comp_lte()` IR method of the `Integer` class, therefore completely bypassing the v-table altogether and with any luck eliminating some of the execution time associated with v-table look-ups. By doing so about one second of execution time can be shaved off, which does suggest that the v-table could be optimised (by using an intermediate cache?), but does not explain what is causing execution time to be so slow.

Next the optimisation passes were placed under scrutiny and the LLVM assembler dumps of an optimised and an unoptimised `fib(25)` program were compared. Indeed the code had been transformed, adding 651 lines of assembler code through what was assumed to be in-lining, however further inspection revealed that there were 37 function declarations in both versions of the assembler code. The dead code elimination pass was verified to be enabled too, meaning that LLVM failed to in-line *any* IR functions at-all. Following this a simple contrived LLVM program was developed (independent of 3c) which was clearly not optimal, then the same LLVM optimiser passes were applied. Section C.5 shows both the un-optimised and optimised IR code. The code had been reduced from 58 to 18 lines and JIT execution time had fallen from an average[2] of 30.37 seconds to just 4.5 seconds. This confirms that the optimiser passes are working properly. It would be interesting to discover why LLVM is not able to optimise 3c code as well as this, but unfortunately once again time restrictions disallow this.

The most frustrating aspect of testing 3c, is that it was impossible to get a larger test sample from the Fibonacci test, due to 3c not having any garbage collection or stack frame clearing. It is common in general purpose programming languages for a function, upon returning to clear the memory it allocated for it's arguments, local variables and constants. This is what is meant by "stack clearing" and garbage collection refers to the act of automatically freeing instances which are no longer referenced at run-time. These features never made it into 3c, but were lined up to be the next 2 iterations of development. As a consequence 3c uses vast amounts of memory. In the test environment (listing C.1) the operating system (OpenBSD) aborts a `fib(26)` program because it exhausts the default maximum heap space of 512MB. Increasing the maximum heap space causes the system to thrash (constant paging between physical memory and disk virtual memory), therefore invalidating any performance comparisons with other language implementations, as disk IO is much slower than memory IO. Java and Lua can do much larger Fibonacci functions in memory without paging.

5.3 Future Improvements

Like any software product, 3c could be improved. This section highlights some ideas which could improve 3c.

[2] An average of 3 runs.

5.3.1 Critical Improvements

The memory usage of 3c is unacceptable and stack frame clearing and garbage collection should be implemented as soon as possible. Stack frame clearing should be trivial, but as for garbage collection, there are two ways this could be achieved. The first approach would be to maintain a table of allocated instances along with a count of how many references there are to each. At a regular interval, records in the table with no references can be freed. The second method would be to attempt to use an automated garbage collector like *Boehm* [Boehm, 2009], which may require custom Python and/or LLVM builds linking a special library. Implementing either of these techniques will cost run-time CPU cycles, but should hugely improve the memory consumption of 3c. The LLVM project has some information about implementing garbage collection on their web-page [Lattner and Henriksen, 2009].

The performance of 3c has a lot of room for improvement. By fixing the memory management of 3c, a performance gain could result. If 3c causes fewer memory allocation system calls, pages of memory may not fragment as much, meaning the operating system will spend less time defragmenting small non-contiguous memory pages. Failing this, a different approach to improving performance should be considered, perhaps by profiling 3c in some way and finding which aspects of the system take longest to execute. The LLVM `opt` utility has some profiling functionality [Spencer, 2009] which could help.

5.3.2 Non-Critical Improvements

The line-by-line parsing approach of 3c works, but as explained earlier, has introduced some complications in the handling of the C/L stack in the 3c mid-layer. Annoyingly feature requiring this awkward parsing approach (an interactive shell) was not implemented due to lack of time, meaning the system suffered for no real reason. 3c could be improved by parsing the code between conditional and loop constructs as an atomic *block*. This approach means that pointers to the IR assembler labels would not have to maintained and perhaps the whole C/L stack would become redundant. Using blocks in this way is common in programming and scripting languages, for example in the Ruby scripting language. Figure 5.5 shows a (multi-line) block in Ruby source code and the corresponding parser grammar is shown in figure 5.6, where `opt_block_param` is the token representing a block.

Listing 5.5: Example Ruby block.

```
for i in (1..10) do
        print i # <- block content
end
```

Listing 5.6: Block parser grammar for Ruby [RBP, 2009].

```
brace_block     : keyword_do
  {
      /*%%%*/
      dyna_push();
      $<num>$ = ruby_sourceline;
      /*%
      %*/
  }
  opt_block_param
```

```
10  compstmt keyword_end
11  {
12      /*%%%*/
13      $$ = NEW_ITER($3,$4);
14      nd_set_line($$, $<num>2);
15      dyna_pop();
16      /*%
17      $$ = dispatch2(do_block, escape_Qundef($3), $4);
18      %*/
19  }
```

Currently 3c is a single pass compiler, meaning that the parse tree is traversed only once, which is at the time the parser actions are applied. 3c could be improved by adding another pass prior to the existing pass. Firstly this would allow ineffectual nodes to be removed from the parse tree, resulting in an AST. Secondly it would allow for example, a function to be called at the top of a source file which is defined at the bottom of the file, but most importantly, it could be used to inject code to define variables at the start of a function (possibly using a *PHI* node [Lattner, 2007]), so that the programmer can have confidence that the ”Instruction does not dominate all uses” error can not be achieved. Work did start on this in iteration 2 of development, but was found to be buggy. As this was not essential to the function of 3c it was postponed allowing features of higher priority to develop instead.

Some minor extra syntactical sugar could be added to the 3c syntax, such as *and* and *or* statements in conditionals and *for* and *do..while* loops. Such statements can be re-factored into already existing 3c syntax, however some programmers would expect these constructs to exist. It would also be nice to be able to use expressions as function call arguments. For now only literals and variables may be used as arguments to functions because of LL(1) shortcomings. Perhaps a different parser should be used.

File	Lines	%
ccc.py	277	6.62
ccc_compiler.py	3659	87.34
ccc_opt_conf.py	227	5.421
glue.py	24	0.57
total	4187	100.00

Figure 5.2: Distribution of lines of code in the 3c mid-layer.

Unfortunately the `Compiler` class of the 3c mid-layer has become large and contains a vast collection of instance variables. In software engineering lingo the code is said to exhibit the “large class code smell” [Fowler, 1999]. Figure 5.2 shows the distribution of the lines of code through-out the Python source code of 3c, revealing that the `Compiler` class accounts for 87% of the code. Most of the code in this class is structure and member function definitions for built-in types. In the interest if code clarity and maintainability, this class should be re-factored [Atwood, 2006]. A proposed re-factoring is shown in figure 5.3, where 3c class and member function definitions have been separated from the compiler class.

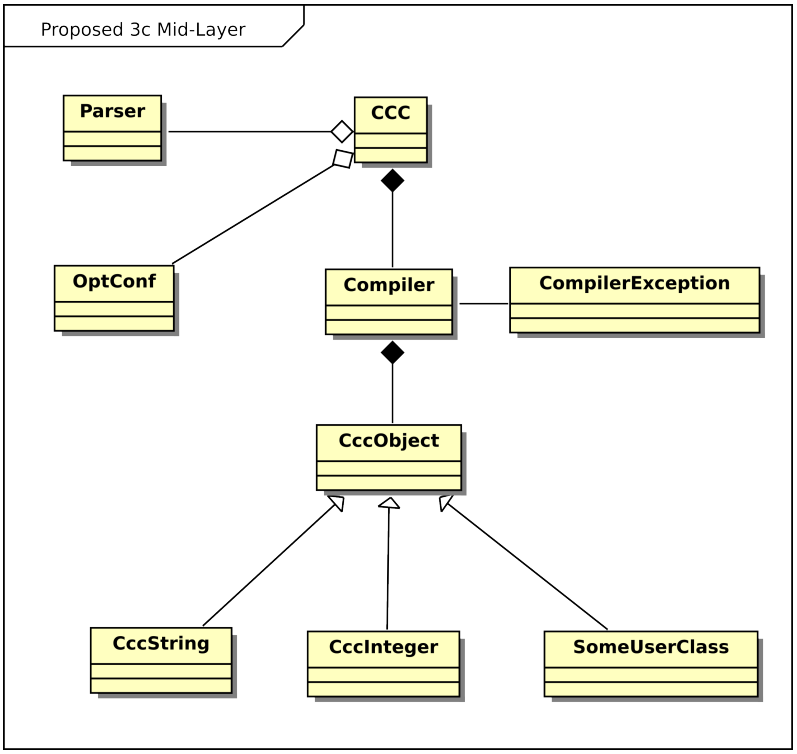


Figure 5.3: Proposed class re-factoring of the 3c mid-layer.

5.3.3 Enhancements

Currently 3c's class hierarchy can not be extended. This makes it limited to string and integer types and operations, which for most programmers will not be enough. The implementation of *user classes*, would allow programmers to make their own classes based upon the built-in classes, via the inheritance mechanism already existing in 3c. This was considered as an iteration in the late stages of development, but discarded due to time constraints.

A common extension to computer languages are *bindings* to external libraries, allowing the programmer to source functionality from elsewhere (usually from a C library) and use that functionality within the source code of the language. Often preparing a language to interface with external libraries requires a small shared object wrapper to be developed, which converts C types into that of the programming language in question and despatches the C function calls to the library itself [The Python Development Team, 2009] [Jung and Brown, 2007]. LLVM potentially has the ability to allow 3c to be extended without needing an intermediate shared object. This is because (as discussed in section 2.6.3) LLVM can call system C functions already, however types would still need conversion. Perhaps extending 3c could be achieved by describing directly in (extended) 3c source language, the interactions with external libraries, without the need for any C shared object wrappers. Without further research and experimentation, it is difficult to say.

5.4 Conclusion

Overall LLVM, even in it's unfinished state, facilitated the construction of 3c comfortably. LLVM has proven to be the most flexible compiler construction kit encountered by the author, provid-

ing many features others do not, such as native code generation facilities, customisable optimiser passes and pure interpretation/JIT options. It is suitable for writing both small plug-in languages, but as proven here, also for fully fledged object oriented languages with polymorphism and inheritance. This research has also confirmed that compiler construction is no longer constrained to low level languages like C. The entirety of 3c was implemented in Python and many other bindings exist, providing low level assembler programming with the convenience of a high level, richly featured scripting language. Following on from this, a wider range of parsing and tokenising tools become available to the developer, such as the one used here (Aperiot), however one should consider carefully the parsing algorithm a parser implementation uses, as the LL(1) algorithm was somewhat limited.

The spiral development model accommodated well, the experimental nature of the project allowing the flexibility of an un-finalised design and set of requirements which are free to evolve. Such a development model is highly recommended for research based and open-source projects, where design is important but evolutionary. Source control packages like subversion are highly recommended, even in a single developer project. Source control was invaluable as a backup and debugging aid throughout the development of 3c. Although the disaster recovery measures implemented were not needed, it gives the author great confidence that they were there, should data loss have occurred.

The only unfortunate elements of 3c (which are not blamed on LLVM), were its memory footprint and performance compared other languages. Given further time perhaps these issues could have been investigated in further detail and resolved. Aside from these set-backs, the author feels that the project was successful, that many lessons were learnt via the fabrication of 3c and that it is a huge personal achievement.

A. References

Virtualization: The big picture. 2007. URL http://btquarterly.com/?mc=virtualization-big-picture&page=virt-viewresearch. Accessed on May 6th 2009.

GNU bison. 2009. URL http://www.gnu.org/software/bison/. Accessed on April 26th 2009.

CWM's parse.y - revision 1.19 (current at time of writing). 2009. URL http://www.openbsd.org/cgi-bin/cvsweb/xenocara/app/cwm/parse.y?rev=1.19. Accessed on April 25th 2009.

Cygwin. 2009. URL http://www.cygwin.com/. Accessed on April 26th 2009.

DejaGnu. 2009. URL http://www.gnu.org/software/dejagnu/. Accessed on May 15th 2009.

Dalvik virtual machine. 2008. URL http://www.dalvikvm.com/. Accessed on May 7th 2009.

Flex. 2009. URL http://flex.sourceforge.net/. Accessed on April 26th 2009.

Cygwin. 2009. URL http://gcc.gnu.org/. Accessed on April 26th 2009.

Overview of arm jazelle technology. 2009. URL http://www.arm.com/products/multimedia/java/jazelle_architecture.html. Accessed on May 7th 2009.

The Low Level Virtual Machine Compiler Infrastructure Web Page. 2009a. URL http://llvm.cs.uiuc.edu/. Accessed on 11th February 2009.

LLVM language reference manual -type system. 2009b. URL http://llvm.org/docs/LangRef.html#typesystem. Accessed on April 27th 2009.

llvmruby blog. 2009c. URL http://llvmruby.org/wordpress-llvmruby/. Accessed on April 27th 2009.

OpenBSD: malloc.conf(3) manual. 2009. URL http://www.openbsd.org/cgi-bin/man.cgi?query=malloc. Accessed on April 29th 2009.

PHP's root configure.in - revision 1.676 (current at time of writing). 2009. URL http://cvs.php.net/viewvc.cgi/php-src/configure.in?revision=1.676&view=markup. Accessed on April 25th 2009.

The Python Scripting Language Web Page. 2009. URL http://www.python.org/. Accessed on 11th February 2009.

Ruby sibversion - view of /trunk/parse.y - revision 23474. 2009. URL http://svn.ruby-lang.org/cgi-bin/viewvc.cgi/trunk/parse.y?revision=23474&view=markup. Accessed on May 18th 2009.

Ruby's root Makefile.in - revision 23150 (current at time of writing). 2009. URL http://svn.ruby-lang.org/cgi-bin/viewvc.cgi/trunk/Makefile.in?revision=23150&view=markup. Accessed on April 25th 2009.

The secure shell. 2009. URL http://www.openssh.org. Accessed on May 12th 2009.

Subversion - open source version control system. 2009. URL http://subversion.tigris.org/. Accessed on May 12th 2009.

Tcl's Makefile.in - revision 1.269 (current at time of writing). 2009. URL http://tcl.cvs.sourceforge.net/viewvc/tcl/tcl/unix/Makefile.in?revision=1.269&view=markup. Accessed on April 26th 2009.

Apache tomcat. 2009. URL http://tomcat.apache.org/. Accessed on May 18th 2009.

Trac - integrated scm and project management. 2009. URL http://trac.edgewall.org/. Accessed on May 12th 2009.

The valgrind web-page. 2009. URL http://valgrind.org/. Accessed on April 29th 2009.

Berkely yacc. 2009. URL http://invisible-island.net/byacc/byacc.html. Accessed on April 26th 2009.

The zonecfg source code (zonecfg_grammar.y and zonecfg_lex.l). 2009. URL http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/cmd/zonecfg/. Accessed on April 25th 2009.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, techniques and tools*. second edition, 2007. ISBN 0-321-48681-1.

Anders Magnusson, based upon works of Stephen C. Johnson of Bell Labs. The portable C compiler source code (cpy.y and scanner.l). 2009. URL http://pcc.zentus.com/cgi-bin/cvsweb.cgi/cc/cpp/. Accessed on April 25th 2009.

Jeff Atwood. Code smells. 2006. URL http://www.codinghorror.com/blog/archives/000589.html. Accessed on May 18th 2009.

Edward Barrett. Software quality and testing assignment. 2009.

Barry W. Boehm. A spiral model of software development and enhancement. 1988. URL http://www.cs.usu.edu/~supratik/CS%205370/r5061.pdf. Accessed on May 12th 2009.

Hans Boehm. Boehm gc - a garbage collector for c and c++. 2009. URL http://www.hpl.hp.com/personal/Hans_Boehm/gc/. Accessed on May 15th 2009.

Pascal Van Cauwenberghe. Going round and round and getting nowhere extremely fast - another look at incremental and iterative development. *Methods and Tools - Global Knowledge Source for Software Development Professionals (Volume 10, Number 4)*, 2002. URL http://www.methodsandtools.com/PDF/dmt0402.pdf.

Alistair Cockburn. *Agile Software Development*. Addison Wesley, 2 edition, 2007. ISBN 0-321-48275-1.

Luiz Henrique de Figueiredo, Waldemar Celes, and Roberto Ierusalimschy. *Lua Programming Gems*. 2008. ISBN 8590379841.

Charles N. Discher and Jr. Richard J. LeBlanc. *Crafting a Compiler with C*. first ed edition, 1991. ISBN 0-8053-2166-7.

Kevin Forsberg and Harold Mooz. The relationship of system engineering to the project cycle. 1994. URL http://www.csm.com/Repository/Model/rep/o/pdf/Relationship%20of%20SE%20to%20Proj%20Cycle.pdf. Accessed on May 20th 2009.

Martin Fowler. *Re-factoring - Improving the Design of Existing Code*. Adisson-Wesley, 1999. ISBN 0-201-48567-2.

Brian Goetz. Java theory and practice: Dynamic compilation and performance measurement. 2004. URL http://www.ibm.com/developerworks/library/j-jtp12214/. Accessed on April 29th 2009.

James Gosling and Henry McGilton. The java™language environment: A white paper. 1995. URL http://www.cab.u-szeged.hu/WWW/java/whitepaper/java-whitepaper-1.html. Accessed on April 29th 2009.

Dick Grune and Ceriel Jacobs. *Parsing Techniques: A Practical Guide*. 2 edition, 2008. ISBN 978-0-387-20248-8. URL http://www.cs.vu.nl/~dick/PTAPG.html. Accessed on May 10th 2009.

Roberto Ierusalimschy. The virtual machine of lua 5.0. 2003. URL http://www.inf.puc-rio.br/~roberto/talks/lua-ll3.pdf. Accessed on May 7th 2009.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The implementation of lua 5.0. URL http://www.tecgraf.puc-rio.br/~lhf/ftp/doc/jucs05.pdf. Accessed on May 8th 2009.

Kurt Jung and Aaron Brown. *Beginning Lua programming*. John Wiley and Sons, 2007. ISBN 0470069171.

Chris Lattner. Kaleidoscope: Extending the language: Mutable variables. 2007. URL http://llvm.org/docs/tutorial/LangImpl7.html. Accessed on May 20th 2009.

Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004. Accessed on April 27th 2009.

Chris Lattner and Gordon Henriksen. Boehm gc - a garbage collector for c and c++. 2009. URL http://llvm.org/docs/GarbageCollection.html. Accessed on May 15th 2009.

Tim Lindholm and Frank Yellin. *The Java™Virtual Machine Specification*. 2nd edition, 1999. URL http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html. Accessed on April 28th 2009.

Glenford J Myers. *The Art of Software Testing*. Wiley Publishing, 1st edition, 1979. ISBN 0471043281.

Bryan O'Sullivan. Haskell bindings for LLVM. 2009. URL http://darcs.serpentine.com/llvm/. Accessed on April 27th 2009.

Bruce Perens. Free software from bruce perens (electric fence). 2009. URL http://perens.com/FreeSoftware/. Accessed on April 29th 2009.

Ernesto Posse. The Aperiot Web Page. 2009. URL http://sites.google.com/site/aperiotparsergenerator/. Accessed on 15th February 2009.

Ernesto Posse. Parsing revisited: a transformation-based approach to parser generation. 2007. URL http://sites.google.com/site/aperiotparsergenerator/documentation-1/posse_aperiot_pycon07.pdf?attredirects=0. Accessed on May 1st 2009.

R Mahadevan. LLVM Python Bindings. 2009. URL http://mdevan.nfshost.com/llvm-py/. Accessed on 11th February 2009.

Marc Roper. *Software Testing*. McGraw Hill International, 1994. ISBN 0-07-707466-1.

Tim Rowledge. A tour of the squeak object engine. 2001. URL http://stephane.ducasse.free.fr/FreeBooks/CollectiveNBlueBook/oe-tour-sept19.pdf. Accessed on May 6th 2009.

Matthew V. Rushton and Haverford College. Static and dynamic type systems. 2004. URL http://www.google.co.uk/url?sa=t&source=web&ct=res&cd=11&url=http%3A%2F%2Ftriceratops.brynmawr.edu%2Fdspace%2Fbitstream%2F10066%2F624%2F2%2F2004RushtonM.pdf&ei=xfITSsG6DtqZjAfckfnjCA&usg=AFQjCNH6yAbNj-7JmGLhgzL3Dg0gPKIRSA&sig2=NABc6FJjwMXWOl6TP27Qnw. Accessed on May 20th 2009.

Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual machine showdown: Stack versus registers. 2005. URL http://www.usenix.org/events/vee05/full_papers/p153-yunhe.pdf. Accessed on May 7th 2009.

Reid Spencer. LLVM's analysis and transform passes. 2009. URL http://llvm.org/docs/Passes.html. Accessed on April 27th 2009.

W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992. ISBN 0-201-56317-7.

The LLVM Development Team. Broken versions of gcc and other tools. 2009. URL http://llvm.org/releases/2.5/docs/GettingStarted.html#brokengcc. Accessed on May 9th 2009.

The Python Development Team. Extending python with c or c++. 2009. URL http://docs.python.org/extending/extending.html. Accessed on May 9th 2009.

Andrew Tridgell and Paul Mackerras. Rsync - fast incremental file transfer. 2009. URL http://www.samba.org/rsync/. Accessed on May 12th 2009.

Larry Wall. Perl 5's Makefile.SH - current revision at time of writing. 2009. URL http://perl5.git.perl.org/perl.git/blob/3e21d4f03715f95b91263d6985791a97f088a54e:/Makefile.SH. Accessed on April 26th 2009.

B. 3c Documentation

B.1 Installation Instructions

In order to install 3c you will need to install LLVM-2.4, Python-2.6 and llvm-py-0.5. 3c *may* work with other versions of these components, but are untested.

1. Copy the `phase-15` directory out of the distribution into the destination installation directory.
2. Add the `phase-15` directory to your system PATH.
3. Run `python ccc.py < arguments >`.

B.2 Manual Page

NAME

The 3c Compiler

SYNPOSIS

3c [-b] [-c] [-d] [-o *dumpfile*] [-v] [-V] *program*

DESCRIPTION

The 3c compiler is a pure object oriented programming language which is executed using the LLVM JIT engine. It is not designed for general purpose use, but instead as an example of how to write JIT compiler using the LLVM compiler construction kit.

COMMAND LINE OPTIONS

-b	Causes 3c to dump a byte-code file prior to JIT execution. The name of the file will be prompted unless *-o* is specified.
-c	Enters the interactive optimiser configuration editor before compilation begins.
-d	Causes 3c to do a "dry run". The process will exit after parsing and IR generation.
-o dumpfile	Used with *-o* in order to specify the byte-code dump file name.
-v	Causes the 3c mid-layer to print miscellaneous debugging information, such as parser hook information.
-V	Turns on *settrace()* based debugging. For use by developers only.
program	Path to a 3c source code file.

HISTORY

3c was developed as a Bournemouth University final year project in the years 2008 and 2009.

CAVEATS

3c does no garbage collection or stack frame freeing.

AUTHORS

Edward Barrett <eddbarrett@googlemail.com>

Operator	Comparison
==	Equality
!=	Inequality
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Figure B.1: Valid comparison operators for 3c.

B.3 3c Syntax Reference

Construct	Description
`pass`	Do nothing.
`print` <*expression*>	Calls the object returned by <*expression*>'s `__print()` method. In the case of the String and Object classes, this prints the value of the object to standard output.
`let` <*var*> = <*expression*>	Assign and if not already defined, define variable <*var*> to the return value of <*expression*>.
`func` <*func*>	Define a function named <*func*>. Statements up until the next `func_done` are taken as the body of the function.
`func_done`	End a function block.
`ret` <*expression*>	Return the evaluated value of <*expression*> from the current function.
`call` <*func*> ([<arg^1>] [...] [<arg^n>])	Call a function named <*func*>with argument list <arg^1> ... <arg^n>. An argument may be a Integer/String constant or a variable name. Argument lists may be zero length.
`call ->` <*method*> ([<arg^1>] [...] [<arg^n>])	Call a method named <*method*> with argument list <arg^1> ... <arg^n>. An argument may be a Integer/String constant or a variable name. Argument lists may be zero length.
`new` <*obj*>([<arg^1>] [...] [<arg^n>])	Instantiate a new object of type <*obj*>, passing argument list <arg^1> ... <arg^n> to the constructor. An argument may be a Integer/String constant or a variable name. Argument lists may be zero length.
`if` <*expression*> <*comp_op*> <*expression*>	Conditionally branch to the next block if <*expression*> <*comp_op*> <*expression*> returns true, otherwise branch to the `else` block of the conditional (if it exists). If the condition evaluates false and an `else` block is absent, execution resumes after the next `if_done`. See figure B.1 for a list of valid comparison operators.
`else`	Define a the block to jump to in the event that the proceeding conditional statement evaluated false.
`if_done`	Terminate a conditional statement.
`while` <*expression*> <*comp_op*> <*expression*>	Iterate the block up to the next `while_done` while <*expression*> <*comp_op*> <*expression*> evaluates true. The block will either execute in full or not at all. See figure B.1 for a list of valid comparison operators.
`while_done`	Terminate a while loop.

An expression may be any of the following: <*numeric literal*>, <*string literal*>, `new`, `call`, <*expression*> <*poly operator*> <*expression*>, (<*expression*>).

C. Testing Materials

C.1 Test Environment

Listing C.1: The test environment configuration, as reported by `dmesg(8)`.

```
OpenBSD 4.5-current (GENERIC) #88: Tue Apr 21 19:44:23 MDT 2009
    deraadt@i386.openbsd.org:/usr/src/sys/arch/i386/compile/GENERIC
cpu0: Intel(R) Pentium(R) M processor 1700MHz ("GenuineIntel" 686-class) 600 MHz
cpu0: FPU,V86,DE,PSE,TSC,MSR,MCE,CX8,SEP,MTRR,PGE,MCA,CMOV,PAT,CFLUSH,DS,ACPI,MMX,FXSR,SSE,SSE2,TM,SBF,EST,TM2
real mem  = 1072656384 (1022MB)
avail mem = 1028902912 (981MB)
mainbus0 at root
bios0 at mainbus0: AT/286+ BIOS, date 09/22/05, BIOS32 rev. 0 @ 0xfd750, SMBIOS rev. 2.33 @ 0xe0010 (57 entries)
bios0: vendor IBM version "1QET97WW (3.02 )" date 09/22/2005
bios0: IBM 2673W7Z
apm0 at bios0: Power Management spec V1.2
apm0: battery life expectancy 99%
apm0: AC off, battery charge high, estimated 2:51 hours
acpi at bios0 function 0x0 not configured
pcibios0 at bios0: rev 2.1 @ 0xfd6e0/0x920
pcibios0: PCI IRQ Routing Table rev 1.0 @ 0xfdea0/272 (15 entries)
pcibios0: PCI Interrupt Router at 000:31:0 ("Intel 82371FB ISA" rev 0x00)
pcibios0: PCI bus #6 is the last bus
bios0: ROM list: 0xc0000/0x10000 0xd0000/0x1000 0xd1000/0x1000 0xdc000/0x4000! 0xe0000/0x10000
cpu0 at mainbus0: (uniprocessor)
cpu0: Enhanced SpeedStep 600 MHz (956 mV): speeds: 1700, 1400, 1200, 1000, 800, 600 MHz
pci0 at mainbus0 bus 0: configuration mode 1 (bios)
io address conflict 0x5800/0x8
io address conflict 0x5808/0x4
io address conflict 0x5810/0x8
io address conflict 0x580c/0x4
```

```
extent ‘pciio’ (0x0 - 0xffff), flags=0
     0x1800 - 0x186f
     0x1880 - 0x189f
     0x18c0 - 0x18ff
     0x1c00 - 0x1cff
     0x2000 - 0x207f
     0x2400 - 0x24ff
     0x3000 - 0x8fff
extent ‘pcimem’ (0x0 - 0xffffffff), flags=0
     0x1000 - 0x9ffff
     0xd2000 - 0xd3fff
     0xdc000 - 0x3ff78fff
     0x3ff80000 - 0x400003ff
     0xc0000000 - 0xc00003ff
     0xc0000800 - 0xc00008ff
     0xc0000c00 - 0xc0000dff
     0xc0100000 - 0xefffffff
     0xff800000 - 0xffffffff
pchb0 at pci0 dev 0 function 0 "Intel 82855PM Host" rev 0x03
intelagp0 at pchb0
agp0 at intelagp0: aperture at 0xd0000000, size 0x10000000
ppb0 at pci0 dev 1 function 0 "Intel 82855PM AGP" rev 0x03
pci1 at ppb0 bus 1
mem address conflict 0xe0000000/0x8000000
extent ‘ppb0 pciio’ (0x0 - 0xffff), flags=0
     0x0 - 0x30ff
     0x4000 - 0xffff
extent ‘ppb0 pcimem’ (0x0 - 0xffffffff), flags=0
     0x0 - 0xc010ffff
     0xc0200000 - 0xffffffff
vga1 at pci1 dev 0 function 0 "ATI Radeon Mobility M6" rev 0x00
wsdisplay0 at vga1 mux 1: console (80x25, vt100 emulation)
wsdisplay0: screen 1-5 added (80x25, vt100 emulation)
radeondrm0 at vga1: irq 11
drm0 at radeondrm0
uhci0 at pci0 dev 29 function 0 "Intel 82801DB USB" rev 0x01: irq 11
uhci1 at pci0 dev 29 function 1 "Intel 82801DB USB" rev 0x01: irq 11
uhci2 at pci0 dev 29 function 2 "Intel 82801DB USB" rev 0x01: irq 11
ehci0 at pci0 dev 29 function 7 "Intel 82801DB USB" rev 0x01: irq 11
usb0 at ehci0: USB revision 2.0
uhub0 at usb0 "Intel EHCI root hub" rev 2.00/1.00 addr 1
ppb1 at pci0 dev 30 function 0 "Intel 82801BAM Hub-to-PCI" rev 0x81
pci2 at ppb1 bus 2
mem address conflict 0xb0000000/0x1000
mem address conflict 0xb1000000/0x1000
```

```
extent ‘ppb1 pciio’ (0x0 - 0xffff), flags=0
     0x0 - 0x3fff
     0x8000 - 0x803f
     0x9000 - 0xffff
extent ‘ppb1 pcimem’ (0x0 - 0xffffffff), flags=0
     0x0 - 0xc02187ff
     0xc0220000 - 0xc023ffff
     0xd0000000 - 0xffffffff
cbb0 at pci2 dev 0 function 0 "Ricoh 5C476 CardBus" rev 0xaa: irq 11
cbb1 at pci2 dev 0 function 1 "Ricoh 5C476 CardBus" rev 0xaa: irq 11
"Ricoh 5C552 Firewire" rev 0x02 at pci2 dev 0 function 2 not configured
em0 at pci2 dev 1 function 0 "Intel PRO/1000MT (82540EP)" rev 0x03: irq
    11, address 00:11:25:b1:32:43
ral0 at pci2 dev 2 function 0 "Ralink RT2561S" rev 0x00: irq 11, address
    00:12:0e:61:5b:74
ral0: MAC/BBP RT2561C, RF RT5225
cardslot0 at cbb0 slot 0 flags 0
cardbus0 at cardslot0: bus 3 device 0 cacheline 0x0, lattimer 0xb0
pcmcia0 at cardslot0
cardslot1 at cbb1 slot 1 flags 0
cardbus1 at cardslot1: bus 6 device 0 cacheline 0x0, lattimer 0xb0
pcmcia1 at cardslot1
ichpcib0 at pci0 dev 31 function 0 "Intel 82801DBM LPC" rev 0x01: 24-bit
    timer at 3579545Hz
pciide0 at pci0 dev 31 function 1 "Intel 82801DBM IDE" rev 0x01: DMA,
    channel 0 configured to compatibility, channel 1 configured to
    compatibility
wd0 at pciide0 channel 0 drive 0: <HTS726060M9AT00>
wd0: 16-sector PIO, LBA, 57231MB, 117210240 sectors
wd0(pciide0:0:0): using PIO mode 4, Ultra-DMA mode 5
pciide0: channel 1 disabled (no drives)
ichiic0 at pci0 dev 31 function 3 "Intel 82801DB SMBus" rev 0x01: irq 11
iic0 at ichiic0
spdmem0 at iic0 addr 0x50: 512MB DDR SDRAM non-parity PC2700CL2.5
spdmem1 at iic0 addr 0x51: 512MB DDR SDRAM non-parity PC2700CL2.5
auich0 at pci0 dev 31 function 5 "Intel 82801DB AC97" rev 0x01: irq 11,
    ICH4 AC97
ac97: codec id 0x41445374 (Analog Devices AD1981B)
ac97: codec features headphone, 20 bit DAC, No 3D Stereo
audio0 at auich0
"Intel 82801DB Modem" rev 0x01 at pci0 dev 31 function 6 not configured
usb1 at uhci0: USB revision 1.0
uhub1 at usb1 "Intel UHCI root hub" rev 1.00/1.00 addr 1
usb2 at uhci1: USB revision 1.0
uhub2 at usb2 "Intel UHCI root hub" rev 1.00/1.00 addr 1
```

```
usb3 at uhci2: USB revision 1.0
uhub3 at usb3 "Intel UHCI root hub" rev 1.00/1.00 addr 1
isa0 at ichpcib0
isadma0 at isa0
pckbc0 at isa0 port 0x60/5
pckbd0 at pckbc0 (kbd slot)
pckbc0: using irq 1 for kbd slot
wskbd0 at pckbd0: console keyboard, using wsdisplay0
pms0 at pckbc0 (aux slot)
pckbc0: using irq 12 for aux slot
wsmouse0 at pms0 mux 0
pcppi0 at isa0 port 0x61
midi0 at pcppi0: <PC speaker>
spkr0 at pcppi0
lpt2 at isa0 port 0x3bc/4: polled
npx0 at isa0 port 0xf0/16: reported by CPUID; using exception 16
fdc0 at isa0 port 0x3f0/6 irq 6 drq 2
biomask effd netmask effd ttymask ffff
mtrr: Pentium Pro MTRR support
softraid0 at root
root on wd0a swap on wd0b dump on wd0b
```

C.2 Boundary Value Analysis Tests

The expected outcomes are encoded within the source of the 3c programs.

C.2.1 Equality Operator Test

Input Program

```
#!/usr/bin/env 3c

# test equality partition
print "subtest 1"
if 99 == 99
        print "pass"
else
        print "fail"
if_done

# test lower partition
print "subtest 2"
if 98 == 99
        print "fail"
else
        print "pass"
if_done

# test upper partition
print "subtest 3"
if 100 == 101
        print "fail"
else
        print "pass"
if_done
```

Output

```
subtest 1
pass
subtest 2
pass
subtest 3
pass
```

TEST PASSED

C.2.2 Less Than Operator Test

Input Program

```
#!/usr/bin/env 3c

# test true outcome
print "subtest 1"
if 1 < 2
        print "pass"
else
        print "fail"
if_done

# test false outcome
print "subtest 2"
if 2 < 2
        print "fail"
else
        print "pass"
if_done
```

Output

```
1-pass
2-pass
```

TEST PASSED

C.2.3 Greater Than Operator Test

Input Program

```
#!/usr/bin/env 3c

# test true outcome
print "subtest 1"
if -1 > -99
        print "pass"
else
        print "fail"
if_done

# test false outcome
print "subtest 2"
if -123 > -99
        print "fail"
else
        print "pass"
if_done
```

Output

```
subtest 1
pass
subtest 2
pass
```

TEST PASSED

C.2.4 Less Than or Equal Operator Test

Input Program

```
#!/usr/bin/env 3c

# test true outcome
print "subtest 1"
if -2221 <= -2221
        print "pass"
else
        print "fail"
if_done

# test false outcome
print "subtest 2"
if -2220 <= -2221
        print "fail"
else
        print "pass"
if_done
```

Output

```
subtest 1
pass
subtest 2
pass
```

TEST PASSED

C.2.5 Greater Than or Equal Operator Test

Input Program

```
#!/usr/bin/env 3c

# test true outcome
print "subtest 1"
if 36 >= 36
        print "pass"
else
        print "fail"
if_done

# test false outcome
print "subtest 2"
if 35 >= 36
        print "fail"
else
        print "pass"
if_done
```

Output

```
subtest 1
1-pass
subtest 2
2-pass
```

TEST PASSED

C.2.6 Inequality Operator Test

Input Program

```
#!/usr/bin/env 3c

# test true outcome on lower side
print "subtest 1"
if  100 != 101
        print "pass"
else
        print "fail"
if_done

# test false outcome
print "subtest 2"
if 101 != 101
        print "fail"
else
        print "pass"
if_done

# test true outcome on upper side
print "subtest 3"
if 102 != 101
        print "pass"
else
        print "fail"
if_done
```

Output

```
zsh: segmentation fault (core
    dumped)  3c test.3c |
zsh: done

    tee result.txt
```

TEST FAILED

C.3 Fibonacci Tests

C.3.1 Program Listings

Listing C.2: Fibonacci Number Generator in 3c

```
#!/usr/bin/env 3c
# Fibonacci number generator
# $Id: fib2.3c 238 2009-05-14 02:25:35Z edd $

func fib(n)
        let result = 0
```

```
7
8         if n <= 1
9                 let result = n
10        else
11                let n1 = n - 1
12                let n2 = n - 2
13                let result = call fib(n1) + call fib(n2)
14        if_done
15
16        ret result
17func_done
18
19let loop = 0
20let out = ""
21
22while loop < 25
23        let out = out + call fib(loop) + " "
24        let loop = loop + 1
25while_done
26
27print "Your Fibonacci numbers:"
28print out
```

Listing C.3: Fibonacci Number Generator in Java

```
1 import java.io.*;
2
3 class Fib {
4
5         private static int fib(int n) {
6                 if (n <= 1) return n;
7                 else {
8                         return fib(n-1) + fib(n-2);
9                 }
10        }
11
12        public static void main(String[] args) {
13
14                String out = "";
15                for(int i = 0; i < 25; i ++) {
16                        int res = fib(i);
17                        out = out + res + " ";
18                }
19
20                System.out.println("Your Fibonacci numbers:");
21                System.out.println(out);
```

```
22
23                  return;
24          }
25  }
```

Listing C.4: Fibonacci Number Generator in Lua

```
1  #!/usr/bin/env lua
2
3  -- http://en.literateprograms.org/Fibonacci_numbers_(Lua)
4  function fib(n) return n<2 and n or fib(n-1)+fib(n-2) end
5
6  out = ""
7  i = 0
8  while i < 25 do
9          fib(i)
10         out = out .. fib(i) .. " "
11         i = i + 1
12 end
13
14 print("Your Fibonnaci numbers:")
15 print(out)
```

```
1
3
6
7
8
9
10
11
12
13
15
16
21
27
28
29
40
47
59
60
```

Figure C.1: Optimisation configuration file for optimised `fib(25)` runs.

C.3.2 Results

Test	Size	Time in Seconds											
		1	2	3	4	5	6	7	8	9	10	Total	Avg
3c/Not Optimised/JIT	1612 (lines IR asm)	16.622	16.340	16.644	17.722	16.638	16.194	16.796	16.525	16.451	16.357	165.299	16.53
3c/Optimised/JIT	2271 (lines IR asm)	16.675	17.106	18.867	17.019	17.155	17.100	16.728	17.189	17.424	16.795	172.058	17.206
3c/Not Optimised/Native	132.0K (binary on disk)	13.258	13.278	13.974	13.206	13.996	13.730	13.283	13.557	13.644	13.681	135.607	13.561
3c/Optimised/Native	136.0K(binary on disk)	13.537	13.321	13.385	13.475	13.352	14.589	13.358	13.787	13.149	13.827	135.780	13.578
Java	2.0K (byte-code on disk)	0.440	0.424	0.428	0.412	0.435	0.445	0.435	0.436	0.415	0.431	4.301	0.430
Lua	N/A	0.628	0.616	0.609	0.614	0.622	0.611	0.614	0.603	0.616	0.614	6.147	0.615

Figure C.2: Execution results of the fibonnaci test programs.

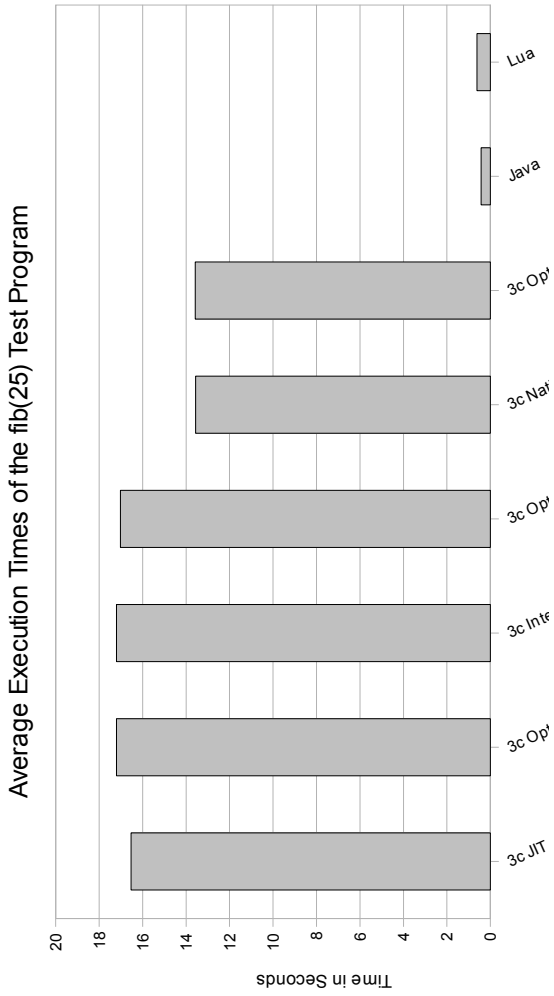


Figure C.3: Graphed average execution times of `fib(25)`

C.4 Nesting Test

Listing C.5: C/L stack nest test source code.

```
#!./3c
# test nesting blocks in 3c
# $Id: nest_test.3c 241 2009-05-14 16:46:16Z edd $

func nest()
        let r = 3
        while r > 0
                print "loop1"

                let j = 4
                if j == 2
                        print "fail"
                else
                        print "cond1 pass"
                        if 1 == 2
                                print "fail"
                        else
                                print "cond2 pass"
                                let j =2
                                while j > 0
                                        print "loop2"
                                        let j = j -1
                                while_done
                                print "loop2 exit"
                        if_done
                if_done
                let r = r - 1
                print "---"
        while_done
        print "loop1 exit"
        ret 0
func_done

call nest()
```

Listing C.6: C/L stack nest test expected and actual outcome

```
loop1
cond1 pass
cond2 pass
loop2
loop2
```

```
loop2 exit
---
loop1
cond1 pass
cond2 pass
loop2
loop2
loop2 exit
---
loop1
cond1 pass
cond2 pass
loop2
loop2
loop2 exit
---
loop1 exit
```

TEST PASSED

C.5 Contrived Optimiser Test

Listing C.7: Un-optimised

```
; ModuleID = 'mod'

declare i32 @printf(i8*, ...)

define i32 @test(i32) {
entry:
        %1 = alloca i32         ; <i32*> [#uses=7]
        store i32 0, i32* %1
        %2 = load i32* %1               ; <i32> [#uses=1]
        %3 = add i32 %2, 1              ; <i32> [#uses=0]
        %4 = load i32* %1               ; <i32> [#uses=1]
        %5 = add i32 %4, 1              ; <i32> [#uses=0]
        %6 = load i32* %1               ; <i32> [#uses=1]
        %7 = add i32 %6, 0              ; <i32> [#uses=0]
        %8 = load i32* %1               ; <i32> [#uses=1]
        %9 = add i32 %8, 0              ; <i32> [#uses=0]
        %10 = load i32* %1              ; <i32> [#uses=1]
        %11 = add i32 %10, 1            ; <i32> [#uses=0]
        %12 = load i32* %1              ; <i32> [#uses=1]
        %13 = add i32 %12, 1            ; <i32> [#uses=1]
        ret i32 %13
```

```
}

define i32 @main() {
entry:
        %0 = alloca [4 x i8]            ; <[4 x i8]*> [#uses=2]
        store [4 x i8] c"%d\0A\00", [4 x i8]* %0
        %1 = getelementptr [4 x i8]* %0, i32 0, i32 0          ; <i8*> [#
            uses=0]
        %2 = alloca [5 x i8]            ; <[5 x i8]*> [#uses=2]
        store [5 x i8] c"here\00", [5 x i8]* %2
        %3 = getelementptr [5 x i8]* %2, i32 0, i32 0          ; <i8*> [#
            uses=0]
        %4 = alloca [4 x i8]            ; <[4 x i8]*> [#uses=2]
        store [4 x i8] c"%s\0A\00", [4 x i8]* %4
        %5 = getelementptr [4 x i8]* %4, i32 0, i32 0          ; <i8*> [#
            uses=0]
        %6 = alloca i32         ; <i32*> [#uses=4]
        store i32 0, i32* %6
        br label %check

check:          ; preds = %body, %entry
        %7 = load i32* %6               ; <i32> [#uses=1]
        %8 = icmp eq i32 %7, 1215752191         ; <i1> [#uses=1]
        br i1 %8, label %exit, label %body

exit:           ; preds = %check
        ret i32 0

body:           ; preds = %check
        %9 = call i32 @test(i32 1)              ; <i32> [#uses=1]
        %10 = call i32 @test(i32 %9)            ; <i32> [#uses=1]
        %11 = call i32 @test(i32 %10)           ; <i32> [#uses=1]
        %12 = call i32 @test(i32 %11)           ; <i32> [#uses=1]
        %13 = call i32 @test(i32 %12)           ; <i32> [#uses=1]
        %14 = call i32 @test(i32 %13)           ; <i32> [#uses=0]
        %15 = load i32* %6              ; <i32> [#uses=1]
        %16 = add i32 %15, 1            ; <i32> [#uses=1]
        store i32 %16, i32* %6
        br label %check
}
```

Listing C.8: Optimised

```
; ModuleID = 'mod'

define i32 @main() {
```

```
4  ; <label>:0
5          br label %1
6
7  ; <label>:1             ; preds = %5, %0
8          %2 = phi i32 [ %6, %5 ], [ 0, %0 ]              ; <i32> [#uses=2]
9          %3 = icmp eq i32 %2, 1215752191         ; <i1> [#uses=1]
10         br i1 %3, label %4, label %5
11
12 ; <label>:4             ; preds = %1
13         ret i32 0
14
15 ; <label>:5             ; preds = %1
16         %6 = add i32 %2, 1              ; <i32> [#uses=1]
17         br label %1
18 }
```