A BCPL Front End for GCC

Thomas Crick
BSc (Hons) Computer Science
University of Bath

May 2004

Abstract

The BCPL programming language was developed by Martin Richards at the University of Cambridge in 1967. It was a simplified version of the earlier language CPL and over the following fifteen years became a popular language for system programming. It was typeless, block structured and gave the programmer enormous power and flexibility. The development of its descendant C as the language of choice for system programming led to its decline, but large amounts of legacy code exists today. This dissertation attempts to apply modern compilation techniques to craft a BCPL compiler and also investigates a number of recent developments in the GNU Compiler Collection (GCC). A solution is proposed by developing a BCPL front end for GCC using the optimisation framework for trees based on the Static Single Assignment (SSA) form. It is concluded that the implementation would provide legacy BCPL users with access to a popular, robust and multi-platform compiler toolchain.

A BCPL Front End for GCC

submitted by Thomas Crick

COPYRIGHT

DECLARATION

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of this work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university of institution of learning. Except where specifically acknowledged, it is the work of the author.

This thesis may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purpose of consultation.

Signed: ....

Acknowledgements

I would like to say a massive thanks to Professor John Fitch, whose experience and guidance during the project was invaluable. I would also like to thank my girlfriend Kate and my parents for their support over the past four years.

Contents

List of Figures

Chapter 1

Introduction

1.1 Aim

The aim of this dissertation is to develop a compiler for the system programming language BCPL.

1.2 Objectives

- To provide a compiler for the BCPL language as defined by the creator Martin Richards (Richards 1967).
- To provide for some of the commonly-used extensions of the language.
- To investigate common compiler toolchains and see how they could be utilised to provide a framework for a BCPL compiler.
- To develop a modern compilation strategy for BCPL legacy users by developing a BCPL front end for GCC.

1.3 Structure

Chapter 2 discusses in depth the history and heritage of BCPL and details some of the more important language features. It also investigates previous work in developing a BCPL compiler and discusses a wide range of approaches to the problem. In Chapter 3 the process of requirements elicitation and capture is examined with respect to the problem at hand, followed by an overview of the software and design decisions in Chapter 4. Chapter 5 discusses in detail the chosen implementation of the project and reflects on how specific difficulties were overcome. The process of testing the system, followed by concluding remarks and future work is in Chapters 6 and 7 respectively.

Chapter 2

Literature Survey

2.1 History

2.1.1 CPL

The publication in 1963 of the revised ALGOL report (Naur, Backus & McCarthy 1963) marked an enormous step forward in the design of programming languages. Probably the most important idea, which arose almost by accident, was that of block structure and the stack mechanism for implementing it. A block is a set of data definitions followed by a sequence of commands and may be nested one within another. Each data definition implies a scope, over which the defined term is visible, i.e. where it may be legally referred to in either commands or other definitions. Outside this scope, the item has no existence. Thus memory space need be allocated for data only while there are commands to process it; and data names can be reused in different contexts (Fischer & LeBlanc 1991).

During the decade that followed the ALGOL report, several new languages were devised, each claiming some superiority over ALGOL 60, but all borrowing ideas from it, including that of block structure. CPL (Combined Programming Language) was a collaborative venture between the University of London Institute of Computer Science and the Cambridge University Mathematical Laboratory. CPL was never fully implemented, partly due to the immature compiler technologies of the time, but it did appear in restricted forms on the Atlas and Titan computers at Cambridge (Barron, Buxton, Hartley, Nixon & Strachey 1963).

CPL was heavily influenced by ALGOL 60, but unlike ALGOL 60 which was extremely small, elegant and simple, CPL was big, only moderately elegant and complex. It was strongly typed, but also had a ’general’ type enabling a weak form on polymorphism (Emery 1986). It also had a purely functional subset, providing a `where` form of local definitions. List structures were polymorphic, with list selection being done through structure matching. It was intended to be good for both scientific programming (in the way of Fortran and ALGOL) and also commercial programming (in the way of Cobol), but essentially tried to do too much. It could be seen as a similar effort to PL/1 in this way, or to later efforts such as Ada. However, the principle motivation for CPL was the contention that while ALGOL 60 was an ideal medium for defining algorithms, it was too far removed from the realities of computing hardware. It is probably this ideal more than anything else that led to the success of CPL as the progenitor of system programming languages, which are nothing if not efficient in the generation of object code. However, there was still a need for an efficient system programming language that was based along the original ideas of CPL but removed the layers of complexity and could handle the technology of the time. This was the prime motivator for the development of BCPL.

2.1.2 BCPL

BCPL (Basic Combined Programming Language) was designed as a seriously simplified, cut-down version of CPL. It was low-level, typeless and block-structured and was devised by Martin Richards at Cambridge University in 1966. The first compiler implementation was written while he was visiting MIT in the spring of 1967, while the language was first described in a paper presented to the 1969 AFIPS[1] Spring Joint Computer Conference (Richards 1969).

The language itself was lean, powerful, and portable. It proved possible to write small and simple compilers for it and was therefore a popular choice for bootstrapping a system. Reputedly some compilers could be run in 16Kb. Several operating systems were written partially or wholly in BCPL (for example, Tripos and AmigaDOS). A major cause of its portability lay in the form of the compiler, which was split into two parts. The front end parsed the source and generated OCODE, an assembly language for a stack-based virtual machine, as the intermediate language. The back end took the OCODE and translated it into the code for the target machine (Richards 1971). Soon afterwards this became fairly common practice, *cf.* Pascal PCODE or the Java Virtual Machine, but the Richards BCPL compiler was the first to define a virtual machine for this purpose. The most recent machine independent implementation of BCPL uses INTCODE, a low-level language interpreted by a fast and simple abstract machine (Richards 1975). This implementation can be downloaded free of charge (for private and academic purposes) from Martin Richard's personal website at the University of Cambridge (Richards 2000).

As said previously, BCPL was typeless in the modern sense, but its one data type was the word, a fixed number of bits usually chosen to align with the architecture's machine word. It could be said that BCPL is an 'operator-typed' language, rather than a 'declaration-typed' language. That is, each object in the language can be viewed as having any type, and can change type at any time. It is the operators used to manipulate the object that determine the type it has at that moment.

---

[1] American Federation of Information Processing Societies

For example, '`+`' adds two values together treating them as integers; '`!`' indirected through a value effectively treats it as a pointer. In order for this to work, the implementation provided no type checking. This had obvious speed benefits from a compiler point of view, but created potential semantic problems. Other interesting properties of the language were all parameters were call-by-value and only one-dimensional arrays were provided. The philosophy of BCPL can summarised by quoting from (Richards 1967):

> *"The philosophy of BCPL is not one of the tyrant who thinks he knows best and lays down the law on what is and what is not allowed; rather, BCPL acts more as a servant offering his services to the best of his ability without complaint, even when confronted with apparent nonsense. The programmer is always assumed to know what he is doing and is not hemmed in by petty restrictions."*

This was a common philosophy with certain language paradigms and perhaps contributed to the open source ideal - the fact that the programmer has nearly total control over what the program can do, even if this means they can perform 'dangerous' operations. This certainly helped BCPL gain large support, especially at a time when restrictive languages were becoming more commonplace.

BCPL's heyday was perhaps from the mid 1970s to the mid 1980s, where implementations existed for around 25 architectures. Today it sees little use outside of legacy systems and academia. However, it is a useful language for experimenting with algorithms and for research in optimising compilers. Its descendant, C, is now the language of choice for system programming. The design of BCPL strongly influenced B, which developed into C. It is also said to be the language in which the original "`hello world`" program was written.

2.1.3 Development of B and C

As stated previously, the development of BCPL has a major impact on the design of system programming languages during the late 1960s and early 1970s. B was designed by Dennis Ritchie and Ken Thompson for primarily non-numeric applications and systems programming (Johnson & Kernighan 1973). It was essentially BCPL stripped of anything Thompson felt he could do without, in order to fit it on very small computers, and with some changes to suit Thompson's tastes (mostly along the lines of reducing the number of non-whitespace characters in a typical program). All B programs consisted of one or more functions, which were similar to the functions and subroutines of a Fortran program, or the procedures of PL/1. B, like BCPL, was a typeless language, all manipulations being on the implementation-dependent word (Emery 1986).

As in BCPL, the only compound object permitted was a one-dimensional vector or words. B does however, incorporate floating point operations, which is implemented in an untyped manner by providing a set of distinct operators that perform specialised operations on which are in effect short word vectors. However, apart from the typeless characteristics and the implications that follow from that, B has more in common with C than with BCPL. The conventions of its syntax are similar to those of C; even extending to the use of identical symbols for identical purposes; and its runtime library foreshadows many of the features of the C library (Emery 1986). In common with many other languages, B was designed for separate compilation. It achieved this aim not with the aid of a global vector like BCPL, but in the conventional way of linking at compile-time like C. A program in B consists of a set of modules, which was not necessarily an individual file, but a distinct object presented to the compiler. Thus several modules may reside in a single file.

Early implementations of B were for the DEC PDP-7 and PDP-11 minicomputers running early UNIX, but as it was ported to newer machines, changes were made to the language while the compiler was converted to produce machine code. Most notable was the addition of data typing for variables. During 1971 and 1972 this became ’New B’ and eventually evolved into C. B was, according to Ken Thompson, greatly influenced by BCPL, but the name B had nothing to do with BCPL. B was in fact a revision of an earlier language, bon, named after Ken Thompson’s wife, Bonnie (Ritchie 1993).

The C programming language came into being in the years 1969–1973 in parallel with the early development of the UNIX operating system (Ritchie & Thompson 1974). In contrast to BCPL and B, C is a typed language. This involved a totally different compilation method, with the introduction of a preprocessor. This is a simple macro processor that runs before the main compiler and was chiefly for handling named constants. Since it is a *pre*processor, it cannot interfere with the compiler, but it converts the program text into a form suitable for the compiler (for example, removing the readability introduced to aid programming). Another spate of changes occurred between 1977–1979, when the portability of the UNIX system was being demonstrated (Johnson & Ritchie 1978). In the middle of this second period, the first widely available description of the language appeared: The C Programming Language, often called the ’white book’ or ’K&R’ (Kernighan & Ritchie 1978). However, in 1989, the language was officially standardised by the ANSI[2] X3J11 committee. In 1990, the ANSI C standard (with a few minor modifications) was adopted by the International Standards Organisation (ISO) as ISO/IEC 9899, with further modifications made in 1999 (ISO 1999).

---

[2] The American National Standards Institute (ANSI) is a private, non-profit organisation that produces industrial standards in the United States.

Until the early 1980s, although compilers existed for a variety of machine architectures and operating systems, the language was almost exclusively associated with UNIX. More recently, its use has spread much more widely and today it is among the languages most commonly used throughout the computer industry (Ritchie 1993). C has a richer set of operators than any other widely used programming language with the possible exception of APL. You will find that in most cases C will do with an expression exactly what you intended to do. However, it is possible to write extremely unreadable C code, as shown by winning examples from the International Obfuscated C Code Contest (IOCCC 2003).

2.1.4 Code Examples

For small code examples of BCPL, B and C, see Appendix B.

2.2 Compiler Phases

Since this is a discussion of developing a compiler for BCPL, it is important to address the issues with developing a compiler and what this entails. To do this, we need to understand the different stages in the compilation process and what actions occurs at each stage.

A compiler is a program that converts another program from a source language to machine language (object code). Some compilers output assembly language which is then converted to machine language by a separate assembler. A compiler is distinguishable from an assembler by the fact that each input statement does not, in general, correspond to a single machine instruction or fixed sequence of instructions. There are generally two parts to compilation: analysis and synthesis. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. The synthesis part constructs the desired target program from the intermediate representation. Of the two parts, synthesis requires the most specialised techniques.

As stated in (Aho, Sethi & Ullmann 1986), a compiler operates in phases, each of which transforms the source program from one representation to another. A typical decomposition of this process can be found in Appendix A.

The phases are often collected into a front and back end, though often a third section is described (the ’middle’ end). The front end consists of those phases, or parts of phases, that depends primarily on the source language and are largely independent of the target machine. These normally include the lexical and syntactic analysis, where the source code is broken into tokens and syntactic structures are identified. These are known as scanning (or ’lexing’) and parsing. The creation of the symbol table occurs during these stages (Wilhelm & Maurer 1995). Semantic analysis is done to recognise the meaning of the program code and start to prepare for output. Scoping and type-checking occurs here (if relevant to source language), with most compilation errors occurring at this stage. One of the final tasks is the generation of the intermediate language to pass to the middle and back ends. The front end also includes the error handling for each of these phases.

The middle end is usually language independent, except for a few callback hooks into the front end. The primary purpose of the middle end is to convert trees (abstract syntax) to RTL (intermediate representation), using various code generation strategies. The RTL maps into machine instructions, so essentially code generation is done first, then optimisation. There is also some, but not much, optimisation of the trees before you generate RTL. In certain compiler models, the middle end phases are shared between the front and back ends.

The back end includes those portions of the compiler that depend on the target machine, and generally, these portions do not depend on the source language, just the intermediate language. In the back end, we find aspects of the code optimisation phase and code generation, along with the necessary error handling and symbol table operations. Resourcing and storage decisions become important, such as deciding which variables to fit into registers (if appropriate) and memory and the selection and scheduling of appropriate machine instructions (Wilhelm & Maurer 1995).

It has become fairly routine to take the front end of a compiler and redo its associated back end to produce a compiler for the same source language on a different architecture. Alternatively, it is also common to change the front end for a compiler and reuse the back end to compile a new language for the same machine (known as ’re-sourcing’). See For a more detailed look at the different phases of a compiler, see (Aho et al. 1986).

2.3 Static Single Assignment Form

Static Single Assignment (SSA) form is an intermediate representation (IR) developed by researchers at IBM in the 1980s in which every variable is assigned exactly once (Cytron, Ferrante, Rosenn, Wegman & Zadeck 1991). Existing variables in the original intermediate representation are split into versions, new variables typically indicated by the original name with a subscript, so that every definition gets its own version.

The primary benefit of SSA comes from how it simultaneously simplifies and improves the results of a variety of compiler optimisations by simplifying the properties of variables. For example, consider the first three lines of code in Figure 2.1.
As humans, we can see that the first assignment is unnecessary and that the value of `y` being used in the third line comes from the second assignment of `y`. A program would have to perform reaching definition analysis (a special type of data flow anal-

$$
\begin{aligned}
y &:= 1 \\
y &:= 2 \\
x &:= y \\
&\ldots \\
y_1 &:= 1 \\
y_2 &:= 2 \\
x_1 &:= y_2
\end{aligned}
$$

Figure 2.1: SSA form example

ysis) to determine this (Cytron et al. 1991). But if the program were in SSA form, as in the last three lines of Figure 2.1, both of these are immediate.

The increased potential for optimisation is the prime motivation for SSA, as many optimisation algorithms need to find all use-sites for each definition and all definition sites for each use (for example, constant propagation must refer to the definition-site of the unique reaching definition). Usually information connecting all use-sites to corresponding definition-sites can be stored as *def-use* chains (for each definition `d` or `r`, list all pointers to all uses of `r` that `d` reaches) and/or *use-def* chains (for each use `u` or `r`, list of pointers to all definitions of `r` that reach `u`). The improvement of def-use chains is one of the key feature of SSA, as each temporary has only one definition in the program, meaning that for each use `u` of `r`, only one definition of `r` reaches `u`. Less space is required to represent def-use chains, as they only require space proportional to `uses * defs` for each variable. SSA also eliminates unnecessary relationships and creates potential register allocation optimisations. Building def-use chains costs quadratic space whereas SSA encodes def-use information in linear space (Cytron et al. 1991).

By using a SSA-based compiler tool, it would be possible to gain large optimisation performance benefits for the compiler. Compiler optimisation algorithms which are either permitted or strongly enhanced by the use of SSA include constant propagation, dead code elimination, global value numbering, partial redundancy elimination, register allocation and sparse conditional constant propagation (for technical details concerning these machine independent optimisations, see (Aho et al. 1986) and (Grune, Bal, Jacobs & Langendoen 2000)). Numerous studies have shown the benefit of SSA forms; their application and optimisations are still current research areas. A method for converting into and using the SSA form is discussed in (Alpern, Wegman & Zadeck 1988).

2.4 GNU Compiler Collection Overview

GCC is the GNU[3] Compiler Collection. Originally, it stood for 'GNU C Compiler', but it now handles many different programming languages besides C. GCC is a GPL[4]-licensed compiler distributed by the Free Software Foundation[5] and a key enabling technology for the Open Source Software (OSS) and free software movements. Originally written by Richard Stallman in 1987, the main goal of GCC was to make a good, fast compiler for machines in the class that the GNU system aims to run on: 32-bit machines that address 8-bit bytes and have several general registers. Elegance, theoretical power and simplicity were only secondary considerations (Stallman 2004*b*).

An important advantage of GCC is its high level of reuse of the larger part of the source code. An important goal was for the compiler to be as machine independent as possible and GCC managed this by getting most of the information about target machines from machine descriptions which give an algebraic formula for each of the machine's instructions. Therefore, GCC does not contain machine dependent code, but it does contain code that depends on machine parameters such as endianness and the availability of auto-increment addressing. The purpose of portability is to reduce the total work needed on the compiler (Stallman 2004*a*). GCC is now maintained by a varied group of open source programmers from around the world, and has been ported to more kinds of architectures and operating systems than any other compiler. GCC has been adopted as the main compiler used to build and develop for a number of systems, including GNU/Linux and Mac OS. Current release series (as of 2004-04-20) is GCC 3.4.0 (available from the GCC Home Page (GCC 2004*c*)). For further details about the current development of GCC, see (GCC 1999). A schematic of the existing GCC infrastructure is shown by Figure E.1 in Appendix E.

2.4.1 GCC Front Ends

As described in Section 2.2, the front end of a compiler handles the source language dependent tasks and is largely independent of the target machine. The interface to front ends for languages in GCC, and in particular the tree structure, was initially designed for C, and many aspects of it are still somewhat biased towards C and C-like languages. Currently the official GCC distribution contains front ends for C (*gcc*), C++ (*g++*), Objective-C, Fortran (*gfortran*), Java (*GCJ*) and Ada (*GNAT*). Several front ends exist for GCC that have been written for languages yet to be integrated into the official distribution. Some of these front ends are works in progress,

---

[3] The GNU Project was launched by Richard Stallman on 27th September 1983 with the goal of creating a free operating system. GNU is a recursive acronym for "GNU's Not UNIX". See `http://www.gnu.org`

[4] The GNU General Public License is a copyleft free software license.

[5] A tax-exempt charity founded by Stallman in 1985 to provide logistical, legal and financial support for the GNU Project. See `http://www.fsf.org`

examples include the GNU Pascal Compiler (*GPC*), Cobol for GCC, GNU Modula-2 and GHDL (VHDL front end).

If we are successful in creating a new front end for GCC, we will in effect be creating a compiler that can create machine code for approximately thirty processor families, including Intel x86, Sun SPARC, DEC Alpha, ARM and Motorola 68000.

2.4.2 GCC Intermediate Representations

Register Transfer Language (RTL) is an intermediate representation (IR) used by the GCC compiler and is used to represent the code being generated, in a form closer to assembly language than to the high level languages which GCC compiles. RTL is generated from the language-specific GCC abstract syntax tree representation, transformed by various passes in the GCC 'middle-end', and then converted to assembly language. GCC currently uses the RTL form to do most of its optimisation work. RTL is usually written in a form which looks like a Lisp S-expression[6]. For example, the 'side-effect expression' shown in Figure 2.2 says 'add register 138 to register 139, and store the result in register 140'.

$$(set : SI(reg : SI140)(plus : SI(reg : SI138)(reg : SI139)))$$

Figure 2.2: RTL example S-expression

The RTL generated for a program is different when GCC generates code for different processors, though the meaning of the RTL is more or less independent of the target: it would be usually be possible to read and understand a piece of RTL without knowing what processor it was generated for. Similarly, the meaning of the RTL does not usually depend on the original high-level language of the program. However, recent developments with GCC have introduced a new intermediate representation framework to improve optimisations and language independence.

The goal of the GCC 'SSA for Trees' project is to build an optimisation framework for trees based on the SSA form (see Section 2.3). While GCC trees contain sufficient information for implementing SSA, there exists two major problems. Firstly, there is no single tree representation in GCC, as each front end defines and uses its own trees. This means code duplication for each front end. Secondly, the trees are arbitrarily complex, giving problems for optimisations that wish to examine them, along with code duplication in the numerous versions.

---

[6] An S-expression is a convention for representing data or an expression in a computer program in a text form. The most common feature is the extensive use of prefix notation with explicit use of brackets.

To address the first problem, GCC have created a common tree representation called GENERIC that is able to represent all the constructs needed by the different front ends whilst removing all the language dependencies (GCC 2003*c*). The design of GENERIC was first discussed on the GCC mailing lists due to fundamental flaws with the existing tree intermediate representations (Merill 2002*a*) as it was felt that they were designed as a convenient shorthand for representing C trees, but is fairly unwieldy for use by optimisers. Essentially, it seems as if the current tree intermediate representation is restricting the amount of possible optimisations, as even some simple optimisations were not being fully implemented. In the `tree-ssa` branch, sparse conditional constant propagation, dead code elimination and partial redundancy elimination with strength reduction (among others) have been implemented in the optimisation passes. A description of the current design can also be found on the GCC mailing lists (Merill 2002*b*).

To address the complexity problem, GCC have implemented a new simplified intermediate representation based on GENERIC. The intermediate representation, called GIMPLE, is a very simple C-like three-address language that is straightforward to analyse and maintains all of the high-level attributes of data types. GIMPLE is derived from the SIMPLE representation proposed by the McCAT project out of McGill University (Hendren, Donawa, Emami, Gao & Sridharan 1992).

As stated previously, the middle end is language independent, except for a few hooks into the front end. The primary purpose of the middle end is to convert trees (abstract syntax) to RTL (intermediate language), using various code generation strategies. The RTL maps into machine instructions, so we essentially do code generation first, and then optimisation. There is also some, but not much, optimisation of the trees before we generate RTL. This changes a bit on the `tree-ssa` branch, as we now have a higher level intermediate language (GIMPLE), and a number of optimisation passes on it, but the main set of optimisations are still done on the RTL (Stallman 2004*a*).

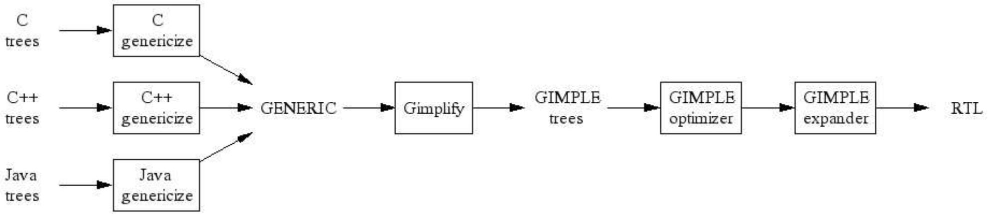

Figure 2.3: GCC implementation of SSA (Novillo 2003*a*)

However, the `tree-ssa` side branch has only recently been fully merged with mainline (2004-05-12) and is now the active development release branch for the next GCC release, 3.5.0. The stabilisation process has being driven by the merge criteria stated in (GCC 2003*c*). Detailed API documentation (created by Doxygen[7]) exists for the `tree-ssa` branch (for example, listing data structures, source file lists, data fields, globals), along with an updated version of the GCC Internals Manual (Stallman 2004*a*).

2.5 Possible Implementations

The following sections discuss a range of different approaches for the project implementation.

2.5.1 Flex and Bison

Lex (Lexical Analyser Generator) and Yacc (Yet Another Compiler Compiler) are tools designed for writers of compilers and interpreters and were both developed at Bell Laboratories in the 1970s. Both Lex and Yacc have been standard UNIX utilities since 7th edition UNIX (circa 1978), but most UNIX/Linux systems use the GNU Project's distributions of Flex (Fast Lexical Analyser Generator) and Bison (Yacc-compatible parser generator). Flex is a tool for generating programs that perform pattern-matching on text. There are many applications for Flex, though it is mainly used for developing compilers in conjunction with GNU Bison. Bison is a general-purpose parser generator that converts a grammar description for an LALR context-free grammar (Chomsky Type II grammar, see (Chomsky 1956)) into a C program to parse that grammar. For this project, we will be using the GNU tools, but the terms Lex/Flex and Yacc/Bison are used interchangeably.

The starting point to this problem is to use Flex and Bison to create a scanner and a parser to process the BCPL code, which could then be compiled using an appropriate C compiler. The hardest part is ensuring that the lexical, syntactic and semantic phases are robust enough to copy with the intricacies of the BCPL syntax (see Section 2.6.1). This phase of the development could be time-consuming, due to the complexity of certain language features in BCPL (see Section 2.6). However, this would be common ground for all approaches, as each require some form of lexical and syntactic phase. Certain problems can occur with using these tools, more so if you wish to customise actions with the generated parser, as it is generally accepted that Bison generates poor C code for humans to maintain.

2.5.2 BCPL to C Conversion

The process of converting a source language into another could be thought of as a somewhat loose definition of compilation. In the BCPL to C case, it could be thought of as a form of 'up-compilation', due to the relationship between BCPL

---

[7] Doxygen is a documentation system for C++, C, Java and IDL, that enables you to automatically create online documentation and reference manuals from the source code. For more information, see `http://www.doxygen.org/`.

and its descendant C. Nevertheless, difficulties occur when trying to convert whole programs into a different language when certain language features are not replicated or allowed, or when certain semantics are hard to preserve. For example, converting from a functional language such as Haskell to an imperative language like C is certainly not trivial, with handling variable assignment and preserving referential transparency being two obvious problems. However, as discussed in Section 2.1.3, C evolved from B, which itself evolved from BCPL. Therefore, certain language features in C have persisted or have derived from features in BCPL, making it a reasonable choice as an intermediate language. It would not be, however, a straight conversion, due to major differences in some critical areas. An obvious example would be handling the conversion into data types, which would have to be done very carefully as it would require analysis of the surrounding environment and operators, as it would be too crude to attempt to convert everything to a single type, such as an integer. This is a perennial problem with the acquisition of type information from a typeless language; a common example would be the conversion of JavaScript code.

Amongst other things, we would also need to ensure that consecutive words in memory have consecutive numerical values (especially with respect to pointer arithmetic) and also the problems of communicating with the global vector (discussed further in Section 5.6.8). This approach still remains feasible, especially since previous work exists in this area (see Section 2.5.8) - it may be possible to build upon this work and extend the functionality. One of the downsides is with the inelegance of the solution, as it would seem crude to force BCPL to C and then compile with a C compiler. Also, C's runtime semantics do not make it easy to honour BCPL's guarantees about dynamic variable values, so there would be numerous problems to resolve. If this were the option chosen, it would probably be easier to use a scripting language such as Perl to write a converter between BCPL and C, but this would circumvent some of the requirements of this project.

2.5.3 Norcroft C Compiler

This second approach builds on the first approach by using Flex and Bison, but instead uses the Norcroft[8] C compiler (NCC) middle and back end phases. The Norcroft C compiler is a optimising and re-targetable ANSI C compiler, which includes many lint[9]-like features and warnings for common syntactic and semantic errors. This method would entail replacing the existing front end and build Norcroft trees for BCPL. We would then be able to reuse some of the existing tree optimisations and generate code with the back end. The Norcroft compiler is a robust and proven C compiler, but the main problem is that it lacks full documentation

---

[8] Available from Codemist Ltd. See `http://homepage.ntlworld.com/codemist/ncc/` for more information.

[9] *lint* is a programming tool for C that performs the lexical and syntactic portions of the compilation with substantial additional checks to trap common coding errors.

with respect to functionality, interfaces and structures. It was not developed with resourcing in mind; this is highlighted by some C language-specific functionality in the middle and back ends. This in itself is not a major problem, but falls short in comparison to other strategies discussed in the following sections. Previous work does exist this this area, as this strategy was attempted last year with some success (see Section 2.5.8).

2.5.4 lcc, A Re-targetable Compiler for ANSI C

lcc is a re-targetable compiler for ANSI C developed by the Department of Computer Science at Princeton University. It generates code for the ALPHA, SPARC, MIPS R3000 and Intel x86 architectures. It has a small and compact interface that includes only essential features which make certain simplifying assumptions that limits the interface's applicability to other languages (Fraser & Hanson 1991*a*). However, many of these assumptions concern the relative size and properties of different data types; this may not be a major problem for the untyped BCPL language. Nevertheless, it is still hard to anticipate the range of restrictions that could exist within the infrastructure. Using lcc would be similar to using the Norcroft compiler, as we would need to replace the existing front end and reuse the existing middle and back ends. The existing lcc front end performs lexical and syntactic analysis as normal, with some minor optimisations. This is connected to the back end by an intermediate language consisting of 36 operators and shared data structures. However, the front and back ends are tightly coupled, with the functions in the interface split between each, to enable callback (Fraser & Hanson 1991*b*). The major disadvantage of lcc is that it does no further optimisation than what is done in the front end. This eliminates a large amount of optimisations that are more effective when performed at the intermediate code level. Also, because of the poor modularity between its front and back ends, replacing the early phases would not be trivial and would also probably require removing the optimisations. Nevertheless, the popularity and pedigree of this production compiler cannot be ignored, but it would require a larger amount of work even when compared with using NCC. Further information about lcc can be found on the project website (Fraser & Hanson 2003).

2.5.5 The Stanford University Intermediate Format Project

The SUIF (Stanford University Intermediate Format) compiler, developed by the Stanford Compiler Group, is a free infrastructure designed to support collaborative research in optimising and parallelising compilers. It is a part of the national compiler infrastructure project funded by DARPA[10] and the NSF[11] (Aigner, Diwan, Heine, Lam, Moore, Murphy & Saputakis 2003). One of the project's main aims

---

[10] The Defence Advanced Research Projects Agency (DARPA) is the central research and development organisation for the US Department of Defence.

[11] The National Science Foundation (NSF) is an independent US government agency responsible for promoting science and engineering through programs that invest in research and education.

has been to create an extensible and easy to use compiler infrastructure but some efficiency and robustness may have been sacrificed to achieve this (SUI 2001). The compiler is based upon a program representation, also called SUIF, with the emphasis on maximising code reuse by providing useful abstractions and frameworks for developing new compiler passes. It also provides an environment that allows compiler passes to easily inter-operate (Aigner et al. 2003). The SUIF2 system is a new design and implementation and is completely different from the previous SUIF1 system, which was originally designed to support high-level program analysis of C and Fortran programs. In comparison to the lcc infrastructure, SUIF has a well designed, modular subsystem that allows different components to be combined easily. It also has an extensible program representation that allows users to create new instructions to capture new programs semantics or new program analysis concepts (Aigner et al. 2003). The existing predefined object hierarchy can thus be further extended or refined to create news abstractions. This would seem to be suitable for our needs, though it is not clear how easy it would be to re-source and redefine the front end and also assembling a suitable back end from existing passes. However, the modular nature of this infrastructure and the very fact that it was designed to be extensible means SUIF is a viable option; more so with the fact that the user is insulated from the details of the implementation (by using a high-level specification language called 'hoof' that is macro translated into C++ interfaces and implementations). As remarked for the lcc project, the popularity and pedigree of this compiler framework cannot be ignored, but it is hard to gauge the amount of development required to re-source a SUIF implementation. Further information about SUIF can be found on the project website (SUI 2001)

2.5.6 The Low Level Virtual Machine Project

The LLVM (Low Level Virtual Machine) compiler infrastructure project is a product of the Lifelong Code Optimisation Project in the Department of Computer Science at the University of Illinois, Urbana-Champaign. Like SUIF, it is sponsored by both the NSF and DARPA. Fundamentally, LLVM is a compilation strategy designed to enable effective program optimisation at compile-time, link-time, runtime and offline, while remaining transparent to developers. LLVM's virtual instruction set is a low-level code representation that uses simple RISC-like instructions. However, it provides rich, language-independent, type information and data flow (SSA) information about operands (Lattner & Adve 2004*b*). LLVM is also a collection of source code that implements the language and compilation strategy. The primary components of the LLVM infrastructure are:

- A GCC-based C and C++ front end.
- A link-time optimisation framework.
- Static back ends for the SPARC v9 and Intel x86 architectures.

- A back end which emits portable C code.
- A Just-In-Time (JIT) compiler for SPARC v9 and Intel x86 (Lattner & Adve 2004*a*).

LLVM is a robust system and is particularly well suited for front end development - it currently includes front ends for C, C++ and Stacker (a Forth-like language), with front ends for Java, Microsoft CLI and Objective-Caml in early development. LLVM would definitely be suitable for this project, especially with the extensive online developers documentation (for example (Lattner, Dhurjati & Stanley 2004) and (Lattner & Adve 2004*b*)). One downside is the fact that it is implemented in C++, with heavy use of the STL (Standard Template Library). A lack of technical expertise with C++ would certainly make re-sourcing LLVM harder, more so with the heavy use of templates. Nevertheless, the aggressive life-long optimisation model of LLVM, along with the range of targets and native code generators would certainly make it a viable infrastructure for this project. It has become a popular and commonly-used compiler infrastructure and is still under active development. Further information about LLVM can be found on the project website (Lattner 2004).

2.5.7 GNU Compiler Collection

The popularity and pedigree of GCC is arguably the best out of the options discussed in this section. As mentioned earlier in Section 2.4, the existing supported front ends are complemented by a range of front ends in development. GCC has been available since 1987, when the v0.9 beta was first released. The years of development that have followed have established GCC as the de facto standard open source C compiler. Detailed documentation exists for working with GCC (Stallman 2004*b*) and developing for GCC (Stallman 2004*a*). There also exists a good range of information about developing front ends for GCC (GCC 2003*b*), including a 'Toy' language example and also guides from existing front ends (see Section 2.5.8).

Developing a front end for GCC, rather than compiling directly to assembler or generating C code which is then compiled by GCC, has several advantages. For example, GCC front ends benefit from the support structure for many different target machines already present in GCC, along with the range of optimisations. Some of these, such as alias analysis, may work better when GCC is compiling directly from source code then when it is compiling from generated C code. This is an important point, as better debugging information is generated when compiling directly from source code than when going via intermediate generated C code. As discussed previously, GCC has developed significantly from being just a C compiler. The framework exists to enable compilation of languages fundamentally different from those for which GCC was designed, such as the declarative logic/functional language Mercury (GCC 2003*b*). It is hard to create a truly independent and all-encompassing compilation strategy with respect to intermediate representations and data structures, but the aims of GCC have been to develop a fairly language-independent tree representation and a large number of optimisation algorithms that work on it.

As discussed more thoroughly in Section 2.4.2, the incorporation of the two new high-level intermediate languages (GENERIC and GIMPLE) and the optimisation framework for GIMPLE based on the SSA form for the GCC 3.5.0 release has created an opportunity to develop a cutting-edge front end using the future GCC internal structure. The various improvements to the internal structure of the compiler, along with the several SSA-based optimisers have created new optimisation opportunities that were previously difficult to implement. A problem exists with the amount of documentation available for GENERIC and GIMPLE, as updates to the GCC Internals manual (Stallman 2004*a*) are still incomplete. It has been remarked that if you can say it with the codes in the source definition `gcc/tree.def`, it is in GENERIC form (Merrill 2003). It is not an ideal situation to develop in a language that is not fully documented, but the existing front ends should provide good starting points to work from and reference. Therefore, due to the relationship to C, it would feasible to start with the C front end and attempt to reuse parts for use with BCPL. The sensible approach would be to have the parser directly build GENERIC trees, as this approach has the advantage that we would not need to define our own language-specific syntax trees and then face the extra work required to walk the trees and perform conversions.

Another issue to be considered is the enormity of the task in hand. This would be the same for many of the other options discussed here, but developing a GCC front end is known to be difficult but effective. The scale of the problem and the time allocated for the project would mean that it would probably be likely that for any option chosen, a complete working compiler is unrealistic. For most options, and especially GCC, it is likely that the latter stages of the project would be more research based and a become a discussion about the likely implementation. Nevertheless, it seems that GCC would be ideal for this project, especially from a learning and research viewpoint. It would also be a robust and professional solution to a commonly looked at problem. Further details about GCC and its releases can be found on the project website (GCC 2004*c*).

2.5.8 Existing Work

For what is remarked to be an unused legacy language, numerous attempts at developing a modern compilation strategy for BCPL exist. This section is a discussion of existing implementations and developments involving BCPL.

Martin Richard's original BCPL implementation has been updated and ported to most modern architectures, as mentioned in Section 2.1.2. It uses INTCODE, an low-level language interpreted by a fast and simple abstract machine (Richards 1975). This would be an obvious resource for understanding how the original implementation of BCPL was done and also how the designer of the language envisaged a modern implementation. It also seems that the BCPL to C conversion route has been done previously, with numerous levels of success. However, Martin Richards was unaware of implementations that re-sourced an existing compiler tool, especially not with GCC. He has also developed some BCPL to C conversion routines and has also entertained the idea of a GCC front end, but little has been done in recent years. Further information about Martin Richard and his BCPL implementation can be found on his website at the University of Cambridge (Richards 2000).

Mark Manning of Selwyn College, University of Cambridge, has also done previous work with BCPL. His implementation uses the BCPL to C conversion route, which is then compiled by GCC. This is, therefore, not a true front end for GCC as claimed on his website (Manning 1999), but a converter for BCPL to ANSI C, using some GCC extensions. There is a small technical note on this implementation, as it was not designed as a true C converter, due to the intermediate stages producing C code that would not be written by a human. The code produced is merely a stage before compilation via GCC. The program implements the BCPL language as defined in (Middleton, Richards, Firth & Willers 1982), along with some commonly-used extensions such as the infix byte operator. There is some promising work in this implementation that could be reused, especially with the handling of the library functions. As discussed previously, the translation of BCPL to C is by no means trivial and this is highlighted by the fact that `LONGJUMP` and `LEVEL` have yet to be implemented, along with a range of other library functions. Nevertheless, it may be possible to reuse some of the work from the front end and the implementation of the global vector and base library functions in C. Further information about Mark Manning and his work can be found on his website (Manning 1999).

Developing a compiler for BCPL has previously been set as a final-year undergraduate dissertation project at the University of Bath. In 2003, it was undertaken by Darren Page, who attempted to re-source the Norcroft back end. This was met with some success, with the Norcroft compiler proving itself to be robust enough to handle most of the BCPL language. Some of BCPL's more troublesome features such as `VALOF` and `RESULTIS` had similar representations in the Norcroft intermediate data structures. However, he noted problems with using the Norcroft compiler due to its lack of documentation, especially with using the Norcroft AST as the intermediate representation (Page 2003). It may be possible to reuse some of the existing front end, as he also used Flex and Bison for the early phases. It might also be possible to replicate the implementation of some of the more difficult language features with respect to building an intermediate representation.

Fergus Henderson, formerly of the Department of Computer Science and Software Engineering at the University of Melbourne, has developed an example front end for GCC based around a simple 'Toy' language[12]. The toy language is a small language

---

[12] The Toy language is available for download from `http://www.cs.mu.oz.au/˜fjh/`

intended to demonstrate some of the basic concepts in programming a language front end for GCC. The Bison grammar contains a small amount of productions, with the scanner handling a small subset of keywords and tokens. The current implementation was designed to work with GCC 2.95 and 3.0, so this means that most of the intermediate representation generation code will not be relevant to building GENERIC trees. However, the Toy language is a good starting point of how to develop a front end for GCC and could be useful for creating the appropriate configuration and build files. It will also be simpler than looking at one of the official release front ends, due to the primitive nature of the language. A similar example front end, including a GCC front end 'how-to' has also been written by Sreejith Menon, of the Department of Computer Science and Automation at the Indian Institute of Science[13].

As would be expected, the official GCC Front Ends resource (GCC 2003*b*) is a very good repository for information about developing a front end for GCC. It describes the main front ends distributed with the official releases of GCC, but also details existing front ends that have yet to be integrated into the main distribution. A further resource is an overview of front ends that are currently works in progress, enabling comparisons with how they are implemented and how certain language features are handled. Some of the more advanced front end resources include: the GNU Pascal Compiler (GNU 2003*c*); GNU Modula-2 (GNU 2004*b*); COBOL for GCC (Josling 2001); PL/1 for GCC (Sorensen 2004); and the Design and Implementation of the GNU INSEL Compiler (Pizka 1997).

Difficulties may occur in attempting to analyse and reuse some of the existing work, but some of it presents good starting areas on how to handle BCPL language features or how to start developing a front end for GCC. Both Martin Richards and Mark Manning have been good sources of information for their work and have expressed an interest in the development of this project.

2.6 Design Issues

2.6.1 BCPL Syntax Features

The relationship between BCPL and C has been discussed previously, but certain language features of BCPL may potentially cause problems in this project. Writing a compiler for BCPL will involve consideration of many similar issues to writing a compiler for any other procedural language that uses static scoping. For example, if expressions were confined to those conventionally found in other languages, then functions in BCPL would be very restricted indeed, rather like statement functions in Fortran (Emery 1986). However, BCPL provides an operator

---

`gcc/example-front-end.shar`.

[13] Available from `http://people.csa.iisc.ernet.in/sreejith/`.

`VALOF` which enables a function to be defined in terms of a sequence on commands. A `VALOF` operator must always be matched with a `RESULTIS` command within the block (Emery 1986). Implementing the `VALOF` operator will not be trivial, but a solution may exist using one of GCC's extensions. There is, however, uncertainty how this will be affected by using the GENERIC intermediate representation. There may also be a problem with ensuring ANSI C-compliance, if required.

Another example, is that in contrast to most other block-structured languages, BCPL permits a direct jump from an inner block to an outer block containing it, i.e. from a higher to a lower level on the stack (Richards & Whitby-Strevens 1980). Such a jump is typically performed in the case of exceptions. By providing a direct jump out of a block, rather than obliging control to retrace the full sequence of calls that got it there, BCPL in some sense 'saves' code. A jump out of a block involves two actions: going to a label in another block, and restoring the stack to a level appropriate for the code following that label. The jump is performed by using the library routine `LONGJUMP` (Emery 1986), with the destination label having been declared globally. `LONGJUMP` and `LEVEL` will be difficult features to implement, because it is unknown how the the scoping and stack functionality required will work using GENERIC.

Further issues that need to be considered are the passing of call-by-value parameters, block/lexical scoping, stack frames and the global vector. If these issues cause major problems than potentially the project scope may have to reduced to a subset of the BCPL language. Also, since no official standard for the language was ever completed, the range of commonly-used extensions in the different implementations of BCPL may conflict or be difficult to implement.

2.6.2 Back End Phases

The back end processes are arguably the most complicated to implement in a compiler, due to the problems and intricacies of code generation and optimisations for different platforms. It can be fairly trivial to get a working front end for a language once you have formalised a grammar, but the creation of machine code for a certain platform can be very time consuming. This is another of the benefits from reusing an existing toolchain, as it eliminates the need to worry about code generation and optimisations and also enables you to target the chosen language to the architectures supported by that toolchain. For example, by using GCC, it would be possible to compile BCPL to run on over 30 architectures, including Intel x68, Sun SPARC and ARM. It is obvious from numerous posts on the GCC mailing lists that porting to a new platform is by no means a trivial task, so to remove this aspect from the project to concentrate on the front end is an easy choice. Also, the back end phases are not really the main targets of this project, as we are more interested in investigating re-sourcing an existing compiler framework. The timescales for this project would also prohibit developing a complete back end for even a single architecture. This would exclude having to investigate and implement the range of machine independent and dependent optimisations (for example, common subexpression elimination and peepholing (Aho et al. 1986)), as this will be managed by the GCC middle and back ends.

2.6.3 Coding Standards

The use of coding standards will be important in this project if the GCC approach is used. For work to be accepted into GCC, both GNU and GCC set a range of coding standards (GNU 2003*a*, GCC 2003*a*), which have to be adhered to so as to ensure consistency with existing development code. Therefore, it will be necessary to research the required coding styles and language features that are used or disallowed within the GCC toolchain. Some simple examples covered would be naming conventions, code formatting and Makefile conventions.

2.6.4 Legal

There also exists certain legal ramifications when developing for GCC and GNU. When this project reaches a point where it may be mature enough for a public release, it would be necessary to release the front end as open source, for example as the BCPL front end for GCC, under the GNU Public License (GNU 2003*b*). In this case, certain prerequisites have to be met, including assignments or disclaimers of copyright. This is not current an issue, but needs to be considered, especially as there could be a conflict with the University of Bath's intellectual property regulations (see dissertation copyright declaration). For further details, see the Contributing to GCC website (GCC 2004*a*).

Chapter 3

Requirements

A well constructed set of requirements are essential to the success of a software development effort (Pressman 2000). With this in mind, the requirements specification below is based on the structure provided by the IEEE Recommended Practice for Software Requirements Specifications (IEEE 1998).

3.1 Requirements Analysis

The numerous methods for analysis can be categorised broadly into *top-down* and *bottom-up* approaches. Top-down approaches, such as a 'goal-directed' method (Sommerville 2001) advocate the use of high-level goals as a starting point, whereas bottom-up methods, such as the production of 'use cases' (Sommerville 2001) emphasise details of system usage scenarios. The approach taken for this project was to use a top-down, goal-directed approach.

3.2 Requirements Specification

This section identifies and discusses key requirements for the project and will focus upon areas of particular difficulty.

3.2.1 System Perspective

The aim of this project is to investigate re-sourcing GCC for the BCPL language. We will attempt to adhere to the BCPL language as defined in the Backus-Naur Form in Appendix C, which has been adapted from (Richards & Whitby-Strevens 1980). We will attempt to implement some of the commonly-used extensions in the language.

The intention is to develop a BCPL front end for GCC using the current active mainline development branch (previously the `tree-ssa` branch), which will be released as GCC 3.5.0. The implementation of the SSA-based optimisation framework will provide a host of benefits, especially with the development of the two new high-level, language independent intermediate representations, GENERIC and GIMPLE. The BCPL compiler will be implemented as extension utilising aspects of the GCC system such as the intermediate representations, optimisation passes and code generation phases.

3.2.2 System Features

The system requirements for this project would be similar to most other compilers, though especially similar to those for GCC. For example, the application should accept an input file of BCPL code. It should then perform lexical and syntactic analysis on this file; if any errors are detected at this stage, the user should be notified with appropriate error information and, depending on severity, build should cease. After these stages, the system should build the trees using the GENERIC intermediate representation and then pass this to the GCC middle and back end. At this point, the compilation would become entirely handled by the GCC process. The GENERIC representation would then be converted into GIMPLE form ('gimplification') and the GCC optimisation and SSA passes would be run. If no errors are detected and the build completes successfully, then an executable should be produced, depending on the host and target platform. This would be the basic usage of the compiler, though it should be possible at a later stage to perform the same options for compilation as is normal with GCC. Examples of this would be halting after preprocessing, dumping the parse tree and producing assembler output. It should also be possible to produce linkable object files.

As previously discussed in Section 2.6.4, there may be certain legal issues affecting the system when developing for GCC. The ramifications of reassigning the copyright to the project as and when it is ready for public release is not entirely known, so would require further investigation. Nevertheless, this does not currently affect the development but would need to be considered at a later date.

Emergent system properties for this project may be hard to predict, but it is feasible to aim for a high level of reliability. Other factors like response time, storage capacity and constraints on I/O will be dependent on the host system hardware specification. It may be necessary to set a minimum hardware requirement for memory, hard drive space or even possibly processor speed. Currently, this is not done for GCC, apart from the minimum required for installation and as long as the hardware is still supported. Security will not be an issue with this project, though on certain systems access requirements may apply. It will, however, be necessary to ensure that the compiler is tested thoroughly with problematic, high-resource and boundary cases so as to ensure a large coverage of potential problems. In this way, common application problems such as buffer overflows and memory leaks may be found. There may also exist non-functional requirements in the system such as constraints on the development process and the ramifications of relevant standards. However, a common problem with many non-functional requirements is that they are hard to verify (Sommerville 2001).

Lexical Analyser

The lexical phase of the compiler should take the source language as input and break it into specific lexical tokens. These tokens should represent the units of information in the language, such as identifiers, keywords and operators. The lexical phase should control the input of the source code file and should ensure that the named file exists and is accessible. If the file is not found or is unable to be accessed, an error should be produced. The lexical analyser should be produced by using the compiler tool Flex and should use the necessary language features (such as regular expressions, pattern matching and exclusive states) to produce an efficient lexical phase. It should also be developed so that easy integration with a Bison-generated parser is possible, ensuring any shared data structures or information are accessible. There should also exist some form of error detection, analysis and recovery in the lexical phase. This should notify the user with details about the type and location of the error and, depending on severity, halt compilation.

Symbol Table

The symbol table will be the most persistent data structure during the compilation process. It is important for it to be quick to access, resource efficient and available to all phases that require access. For this project, it is particularly important for the symbol table to be visible to the GCC middle and back ends during the lowering from GENERIC to GIMPLE form and optimisation passes. For a prototype compiler, it would not be necessary to implement a truly efficient symbol table, but for this project we have chosen to implement a separately-chained hash table for our symbol table (see Section 4.4 for technical details). At this time, the efficiency given by the hash table implementation would be suitable for the scale of the project. This should give a scalable implementation that could be improved for a later version. The symbol table should provide the basic functions to insert, lookup and delete entries in the symbol table. It should ensure that any resource allocations it makes are released upon ending of the compilation process or on error. For diagnostic purposes, it should provide a function to dump the contents of the symbol table to a file. Any other functionality should be provided when and if required.

Parser

The parser should be implemented as bottom-up parser generated by Bison, and should be able to easily integrate with the lexical analyser produced by Flex. It should have control of the lexical phase and should parse the code in the selected input file. The implementation of the grammar productions in the Bison parser should follow the BCPL language as detailed in the BNF definition given in Appendix C. However, these may require minor changes to resolve ambiguity and overcome conflicts or errors. The parser should be able to access the symbol table and be able add, remove or lookup entries. One of the main tasks of the parser is to start building the intermediate representation from the parsed code. When a grammar rule or production is detected and accepted, the parser should start to build tree nodes in GENERIC format. No other internal, bespoke or custom tree representation should be used, apart from the GENERIC intermediate language. A key feature of the parser is the integration into the GCC toolchain and how the intermediate representations are passed onto the GCC middle and back ends. It should ensure that the GENERIC trees are passed successfully and that resources are released whenever possible. Like the lexical analyser, some error detection, analysis and recovery should be present in the parser and this again should notify the user with details about the type and location of the error and, depending on severity, halt compilation. As discussed in the requirements for the symbol table, efficiency is often a factor in the parser and so attempts should be made to produce a fast and efficient parser with Bison.

Intermediate Representations

For most compilers, the intermediate representation is invariably designed specifically for the language and reflects certain features of that language. However, since we have chosen to use the new GCC representations in this project, it is hard to state requirements. This is partly due to the fact that we have no control over the development and direction of the language and also because it has yet to be conclusively defined and fully documented. This problem would certainly represent a non-functional requirement or an emergent system property, as it is not under the control of this project. This is mitigated by the recent developments in GCC that make it highly unlikely that major changes to either of these representations or their usage will occur.

GCC Integration

The problems discussed above are similar for the integration with GCC as for the intermediate representations. Because we are utilising an existing compiler toolchain to handle all of the optimisation passes and code generation phases, it is difficult to define requirements for these phases. Nevertheless, since we are using GCC, it is possible to set some metrics about certain expected levels of performance. Without any optimisation options set, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. The compiler should, however, be able to provide a range of optimisation flags that would attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program. As stated, we will have available the whole range of optimisation options for GCC[1]. Requirements would also exist for the target architectures, as we will be constrained by which currently exist for GCC. However, much of this area is beyond the scope of the project, but it is important to recognise the range of the portability of GCC and the difficulty in porting GCC to a new architecture.

3.2.3 User Characteristics

The user of this application would typically be a BCPL (or possibly C) software developer, well versed in using compilers and in particular, GCC. No administrative or superuser privileges would normally be required, but this may differ from system to system. However, by the very nature of this application, the user would hopefully be an experienced computer user with an experience in programming.

3.2.4 Operating Environment

This project was implemented and tested on the Linux operating system (see Appendix G for version information), so initially the operating system requirements would be similar. However, as the project develops and integration in GCC progresses, the range of operating environments would be the same as is for GCC (see (GCC 2004*c*) for further details). At first, the compiler would be delivered as a pre-compiled binary, but if and when GCC integration progresses, it would be possible to distribute with the GCC source or as a pre-built GCC binary. In this case of distributing as source, an existing compiler that is able to bootstrap GCC would then be required. For further details of the programs used in the development of this project, see Appendix H.

3.2.5 Documentation

As previously mentioned, the users of this system would need to be proficient with the BCPL language or have experience with other block-structured or imperative languages, with knowledge or experience with using compilers assumed. With this in mind, providing system documentation or a user manual was deemed not to be primary goal of the project. Nevertheless, the standard command line help (for example with GCC, this is invoked on the command line by `gcc -h`), detailing the compilation options will be provided, along with a version information option (again with GCC, `gcc --version`). Also, all of the source code is commented, with important features and functionality explained (in accordance with the GNU and GCC coding standards, see Section 2.6.3). Two important sources of information with respect to the GCC side of the project would be the 'Using the GNU Compiler Collection' (Stallman 2004*b*) and 'GNU Compiler Collection

---

[1] For more information about the options that control optimisation in GCC, please see `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

Internals' (Stallman 2004*a*) manuals. These are comprehensive references to both running and developing GCC and would be an appropriate resource for problems associated with GCC.

Chapter 4

Design

4.1 System Architecture

As discussed in Section 2.2, the important phases of a compiler can be broken down into basic components, which can then be discussed as high-level design topics. This will give an overview and understanding of the overall composition of the design. The basic components covered in the design will be the lexical analyser, symbol table, parser, generation of the intermediate representations and the integration into GCC. The parser is central to the main design of this system, as it drives the front end and control the lexical analysis. It also builds the intermediate representations and handles the integration with the rest of the GCC toolchain.

4.2 Methodology

Expanding on the requirements analysis, the methodology for the design of the system was based upon the generic architectural model as described in (Sommerville 2001). Generic models are a type of domain-specific architectural models which are abstractions from a number of real systems. They encapsulates the principle characteristics of these systems. A compiler model is a good fit as a generic architectural model and some of the important modules are described below. The phases of lexical, syntactic and semantic analysis would be organised sequentially as shown in Figure 4.1. The components which make up a compiler can also be organised according to different architectural models. A compiler could also be implemented using a composite model, where a data-flow architecture could be used and the symbol table acting as a repository for shared data. However, large systems rarely conform to a single architectural model, as they are heterogeneous and tend to incorporate different models at different levels of abstraction (Sommerville 2001).

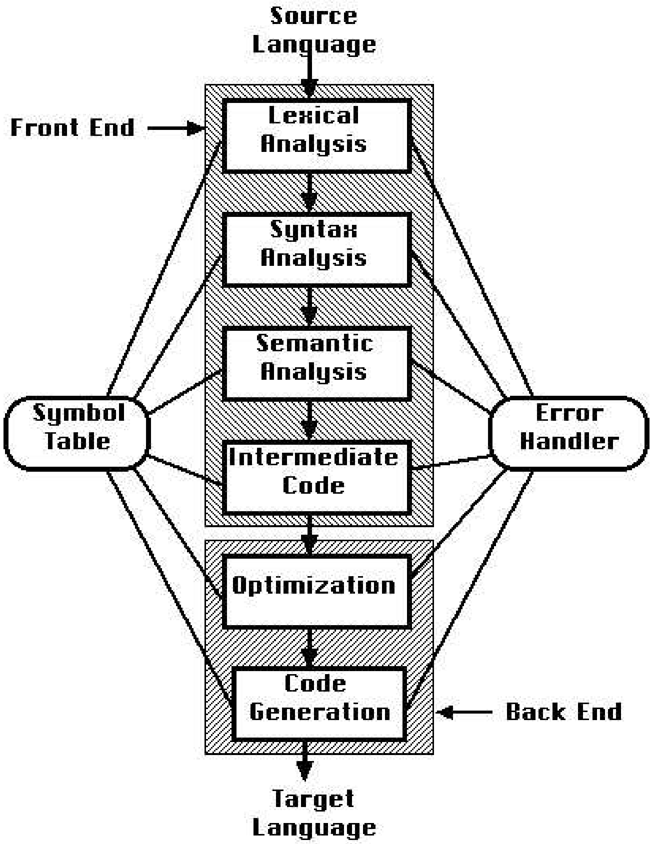

Figure 4.1: Compiler system overview

4.3 Lexical Analyser

The design of the lexical phase needs to ensure that the input language is ’tokenised’ correctly and efficiently. By using the generic model as described above, it would be feasible to create a lexical analyser module that takes the input language and converts it into recognisable symbols ((Richards & Whitby-Strevens 1980) points out that there are about 75 such symbols). This would ensure that the principle characteristic of the lexical phase is encapsulated.

The lexical analyser would need to be driven by a starter class that processes the input files and options and starts the compilation process. Minimal explicit setup would be required, as many of the internal data structures and rules are automatically created by Flex, but it would be sensible to enable some of the diagnostic and debugging options available (see (Paxson 1995) for further details). As soon as each token is encountered, a rule would be triggered and then appropriate actions would follow. The types of token for the BCPL language would include identifiers, numeric constants, keywords and operators; each time one of these tokens were encountered, it would need to be detected and handled appropriately. For example, in most cases, it would be possible just to acknowledge and return that you have detected that particular token, but certain cases would require special handling and may require insertion into the symbol table. Certain cases may also require functionality that copies or adjusts the current token so that it can be passed onto the parser or be analysed further. Like the original BCPL compilers, the lexical analyser will not need to perform any backtracking; that is, it performs the analysis while reading the basic tokens in one at a time without having to consider a symbol previously dealt with (Tremblay & Sorenson 1985).

A problem that is common with the lexical phase is ensuring that you have correctly identified the current token. This is important, otherwise it can be possible to get spurious semantic errors in the parser. A good way to alleviate this problem is to ensure that all keywords, commands and operators are explicitly defined as tokens in the rules section of the scanner and are detected and return as such. It is also important to make sure that the alphabetical case of the input is taken into consideration, as some implementations of BCPL allowed both upper and lower cases to be used (Richards & Whitby-Strevens 1980). When using regular expressions in the other rules to detect identifiers and constants for example, care must be taken to design and test these expressions to check that the character classes are matching only the expected input. It is all too easy to make mistakes whilst using regular expressions for controlling input.

Another important design feature for the lexical analyser is error detection and recovery. It should be possible, whenever a error is detected in the scanner, to maintain control of the compiler without it crashing and also to receive information about the type, location and severity of the error. Certain errors or warnings may be allowed, but some errors may require compilation to halt. The error information that should be presented back to the user should contain the error token, the line and column number of its location and the action problem. This would require tracking of whitespace and newlines, but should be trivial to implement. As much error information as possible is important from a diagnostic and debugging point of view and would also contribute to the testing of the program itself.

We have already covered the use of Flex in both Section 2.5.1 and Section 3.2.2, but there will be a more detailed technical discussion in the implementation of the lexical analyser later on in this report.

4.4 Symbol Table

The symbol table is the major persistent attribute of a compiler and, after the intermediate representations, forms the major data structure as well (Louden 1997). The efficiency of the symbol table in a large production compiler is normally an important issue, but the scope of this project precludes a large amount of time spent optimising symbol table transactions and storage size. However, the chosen design of a separately-chained hash table for the symbol table is a good choice as it provides a good compromise between efficiency and ease of implementation.

The efficiency of the symbol table depends on the implementation of its data storage. Typical implementations include linear lists, various tree structures (such as binary search trees, AVL trees and B trees) and hash tables. Linear lists are a good basic data structure that are simple and quick to implement and can provide easy and direct implementation of the three basic operations. There exists a constant-time insert operation, by always inserting at the front or rear of the list, with lookup and delete operations being linear time in the size of the list (Loudon 1999). This option would be acceptable for a prototype or experimental compiler implementation, where compilation speed is not a major concern. Search tree structures are somewhat less useful for the symbol table, partly because they do not provide best case efficiency, but also because of the complexity of the delete operation (Louden 1997). The hash table will often provide the best choice for implementing the symbol table, since all three operations can be performed in almost constant time and is used most frequently in practice (Aho et al. 1986).

A hash table consists of an array of entries, called buckets, indexed by an integer range. A hash function is required to turn the search key (usually the identifier name, consisting of a string of characters) into an integer hash value in the index range of the table, giving the location of the bucket where that item is then stored. Care must be taken to ensure that the hash function distributes the key indices as uniformly as possible over the index range, since hash collisions (where two keys are mapped to the same index by the hash function) can cause significant performance degradation in the lookup and delete operations. It is also important for the hash function to operate in constant time, or at least time that is linear to the size of the key (this would amount to constant time if there is an upper bound to the size of the key) (Loudon 1999).

The way we have chosen to address the problem of collision resolution in the hash table is using a technique, as mentioned previously, called separate chaining, which is shown in Figure 4.2. In this method, each bucket is actually a linear list and collisions are resolved by inserting the new item into the front of the bucket list (Louden 1997). Another common method used is open addressing, where only enough space is allocated for a single item in each bucket, with collisions being resolved by inserting new items in successive buckets. This method was not chosen because of the performance degradation when collisions become frequent and also that deletions do not improve the subsequent table performance. The size of the initial bucket array is an important design choice that needs to be defined at compiler construction time. Typical sizes range from a few hundred to over a thousand, but the use of dynamic allocation can ensure that small arrays will still allow the compilation of very large programs, at the cost of extra compile time. An interesting feature is that if the actual size of the bucket array is chosen to be a prime number, a typical hashing function behaves more efficiently (Loudon 1999).

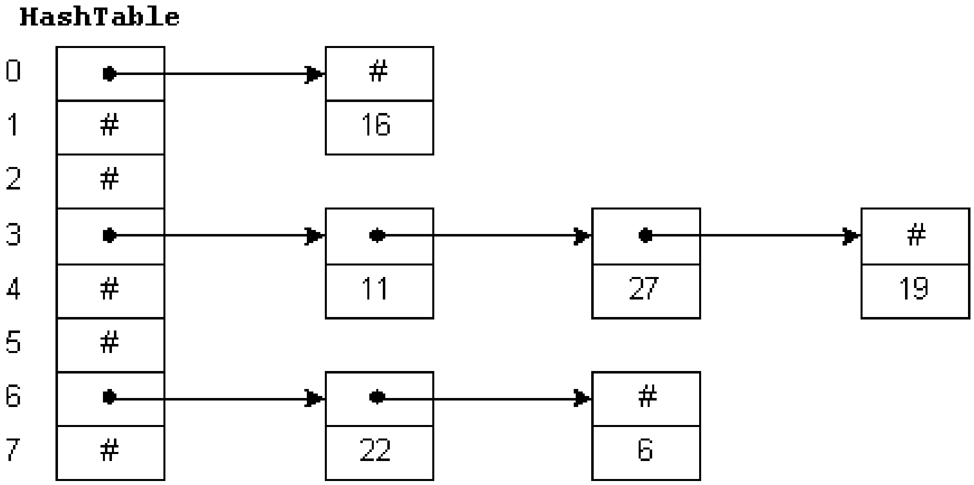

Figure 4.2: Separately chained hash table schematic

As mentioned in the requirements section, the symbol table will provide the principle functions of insertion, lookup and deletion. As their names suggest, insert is used to store symbols, lookup to retrieve them, and delete to remove them from external visibility. The delete function would not physically remove the entry from the table, but set a flag to make it inactive similar to a database record. This would provide a form of transaction logging and ensure that items are not fully removed until the whole table is destroyed. These principle interfaces will be available to all phases that require access and be visible to the GCC middle and back ends during the lowering from GENERIC to GIMPLE form and the optimisation passes. The symbol table is intimately involved with both the scanner and the parser, either of which may need to enter information directly into the symbol table or query it to resolve ambiguities. A print table function will also be available that dumps the information in each bucket of the symbol table to a file for diagnostic purposes. Other functions that would be available would include a check to see if the table is empty, a symbol count function and a function to free the whole symbol table when the compilation has ended.

Since BCPL is typeless in the modern sense, the usual information about data types is not necessary for each entry in the symbol table. An important feature for BCPL is the block-structured nature of the language and how this affects scoping of variables. A block in BCPL is any construct that contains one or more declarations and commands. BCPL permits the nesting of blocks inside other blocks and the scope of an identifier is the declaration itself (to allow recursive definitions), the subsequent declarations and the commands of the block. The scope does not include earlier declarations or extend outside the block (Richards & Whitby-Strevens 1980). Since block-structured languages permit multiple scopes for variables, this complicating the design of the symbol table as multiple scopes imply that a symbol may have different meanings in different parts of the code as highlighted in Figure 4.3. This example in C demonstrates how it is possible to declare new variables at the start of every block and how this can complicate the program. This also highlights lexical scoping present in BCPL, C and most other languages descended from ALGOL. A variable is said to be lexically scoped if its scope is defined by the text of the program (Fischer & LeBlanc 1991). In this way, a programmer can guarantee that their private variables will not be accidentally accessed or altered by functions that they call.

```
static int w;              /* scope level 0 */
int x;
void example(a,b)
{
    int a,b;               /* scope level 1 */
    {
        int c;
        {
            int b,z;       /* scope level 2a */
            ...
        }
        {
            int a,x;       /* scope level 2b */
            ...
            {
                int c,x;   /* scope level 3 */
                b = a + b + c + x;
            }
        }
    }
}
```

Figure 4.3: Nested code blocks example

4.5 Parser

The design of the parser is very important in ensuring that the semantics and the grammar of the language are preserved. The parser is perhaps the most important module in the front end and drives the lexical phase by controlling the scanner and then receiving the token that has been identified. It is still a separate module, but needs to integrate cleanly with the scanner. Again using the generic architectural model described earlier, it should be possible to develop a parser that acts of the data given to it by the lexical phase and then attempt to match it with the grammar productions. The principle characteristic of the system is to match the input against a set of grammar rules and start to build the intermediate representations with the information generated.

By using the Bison parser generator, we are able to take advantage of many automatically generated features. In order for Bison to parse a language, it must be described by a context-free grammar. This means that you must specify one or more syntactic groupings and then give rules for constructing them from their constituent parts (Donnelly & Stallman 2002). For example, in BCPL, one such grouping is called an 'expression', with one of the rules for making an expression being defined as, 'an expression can be made of a minus sign and another expression".

The most common formal system for presenting the grammar rules for humans to read is Backus-Naur Form or 'BNF', which was originally developed in order to specify the language ALGOL 60. Any grammar expressed in BNF is a context-free grammar (see (Chomsky 1956) and (Chomsky 1959) for further details), though it may contain ambiguities. Therefore, the input to Bison is essentially a type of machine-readable BNF. This means that we are able to use the BNF description for BCPL as given in (Richards & Whitby-Strevens 1980) and build the necessary grammar productions. This is a good starting point to getting a working parser for BCPL, even though it may be necessary to make changes whilst implementing due to ambiguities and conflicts that may exist in the given grammar. This is a good point to note that whilst we are working from the BNF as defined in (Richards & Whitby-Strevens 1980), no formal standard exists for the BCPL language. Attempts were made in the early 1980s to create a common standard, but the large number of implementations combined with the waning usage of the language and the popularity of C contributed to it being unsuccessful. This means that the BNF we have formalised is possibly the most comprehensive, but is does not include some of the commonly-used extensions or language features which are not expressible in the BNF notation. A further discussion about the changes made to the BNF is described in Chapter 5, along with details of which extensions have been implemented.

The initial choice of the Bison parser implementation was tempered with initial research into developing a recursive-descent parser like the original BCPL compilers. Recursive-descent parsing is a top-down method of syntax analysis in which you execute a set of recursive procedures to process the input (Aho et al. 1986). This contrasts with using a parser generator like Bison, which produces bottom-up parsers. A recursive descent parser consists of a set of mutually-recursive procedures or non-recursive equivalent where each such procedure implements one of the production rules of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognises. Bison and most other parser generators produce a particular family of bottom-up parsers called LALR(1)[1] parsers. Look-Ahead LR or LALR parsers are a specialised form of LR parsers that can deal with more context-free grammars than SLR parsers but less than LR(1) parsers can (Grune et al. 2000). It is a very popular type of parser because it gives a good trade-off between the number of grammars it can deal with and the size of the parsing tables it requires. This is the one of the key features of using Bison, as the production of parsing tables and the internal handling of the stack to shift and reduce the productions automates a considerable amount of work. It is also a good starting point for understanding the language and its intricacies, as the tool provides a simple interface for starting to build the parser. One of the problems with using a recursive-descent compiler was the greater amount of work involved for implementation, as it would have required a greater understanding of the language to begin with. It would probably in the long term have been a more elegant and efficient solution, but Bison was the choice for ease of implementation and the timescales involved. Also, with previous experience of using Bison, it was possible to reuse existing code for a C-like language rather than attempting to create a parser from the beginning.

The design of the error recovery strategy is another important factor for the parser. This should be used in conjunction with simple error recovery in the lexical phase. It is not usually acceptable to have the program terminate on the occurrence of the first parse error. For example, the compiler should recover sufficiently to parse the rest of the input file and continue checking it for errors. A common technique when a parse error is discovered is calling an error handling routine that advances the input stream to some point where parsing should once again commence. It is then possible to provide error information for the whole input file, rather than stopping at the first error. However, error recovery strategies are by their very nature informed guesses; when you guess wrong, one syntax error often leads to another (Donnelly & Stallman 2002).

4.6 Intermediate Representations

As previously mentioned, one of the key tasks of the parser is to build the GENERIC trees. It is hard to set designs on this part of the program, as we are constrained by the fact it is not our own custom intermediate representation. However, it is possible to state that when we build GENERIC trees, we will adhere to the recommendations as set out in the GCC 3.5.0 version of the GCC Internals Manual (Stallman 2004*a*). This is also the case for the lowering pass to GIMPLE form, which is done by a

---

[1] The notation used for describing parser types, for example LR(k) parsers, means that they read their input from left to right, produce a rightmost derivation and k refers to the number of unconsumed 'look ahead' input symbols that are used in making parsing decisions. Usually k has a value of 1 and is often omitted.

function call provided by the infrastructure.

4.7 GCC Integration

The GCC integration falls under the same situation as discussed above for the intermediate representations. As we are reusing the GCC back end for the optimisation passes and the code generation, we are unable to set or control any features of the design. Also, most of this stage of the compilation will be automatic once we have processed and passed on the intermediate representations. However, we will again adhere to the recommendations as set out in the GCC 3.5.0 version of the GCC Internals Manual (Stallman 2004*b*).

Chapter 5

Implementation

5.1 Overview

This chapter presents an overview of the software architecture implemented and a high-level discussion of the implementation process and the key components of the system. In this project, we have attempted to use an evolutionary and rapid prototyping strategy (Sommerville 2001) for implementation, as this seemed to suit the type of project and the speed of development required.

5.2 Tools

The aim of this project has been to develop a compiler for BCPL by integrating the GCC toolchain to reuse the middle and back end. This design strategy heavily influences our choice of implementation tools, especially if we wish for interoperability, standardisation and support. The choice of tools has also been constrained by the need to support the design choices made in Chapter 4.

The choice of using C as the main implementation language was easy to make, as it has become the de facto standard for compiler implementations. This is compounded by our use of GCC and the intermediate representations, as most of GCC is written in C. C++ is starting to be used for certain compiler implementations, as language features such as templates are useful when creating front ends (a good example would be the LLVM Project, as discussed in Section 2.5.6). A further deciding factor would be if we wished to eventually integrate the BCPL front end fully into GCC, the coding standards for both GNU (GNU 2003*a*) and GCC (GCC 2003*a*) would dictate our choice of language and tools. It is unlikely, however, that choosing C as the implementation language would simplify certain features because of it being a descendant of BCPL. The expressive power of C, coupled with its common usage as a compiler tool are the primary implementation reasons.

5.3 Lexical Analyser

As discussed in Chapter 4, the purpose of the scanner is to recognise tokens from the input file and match them to the corresponding symbols of the language. The choice of implementing the scanner in Flex was again easy to make. Flex and its variants remain the standard tools for creating scanners for modern compilers. A Flex input file consists of three sections, *definitions, rules* and *user code* (Paxson 1995). An example set of rules from the scanner are shown in Figure 5.1. These demonstrate how basic tokens like keywords and operators are detected and their token type is passed to the parser. The scanner uses the enumerated token types as defined in the Bison parser.

```
...
"GLOBAL"        { return GLOBAL; }
"LET"           { return LET; }
"WHILE"         { return WHILE; }
"TABLE"         { return TABLE; }
":="            { return ASSIGN; }
"!"             { return PLING; }
"->"            { return IMPLIES; }
...
```

Figure 5.1: Example Flex rules for keywords and operators in BCPL

However, for handling more complicated token such as identifiers and numeric constants, the use of regular expressions[1] were required to control the range of tokens captured. A good example from the definitions section of the scanner would be the regular expression for identifiers as shown in Figure 5.2. This shows how you can define character classes to create groups of characters to define larger entities. Therefore, an identifier consists of a string starting with a letter, followed by either an alphanumeric character, a period or an underscore.

The scanner performs tracking of section brackets (see Appendix C) by recording the entry and exit of a new block and hence new scope for the program. This means that when an opening bracket, ’`$(`’, or closing bracket, ’`$)`’, is found, a new scope is either opened or closed. A section bracket may be optionally tagged with an identifier to aid tracking of blocks; when this occurs the value associated with the opening bracket is pushed onto the stack. When a closing bracket is recognised by the scanner, the top of the stack is popped and compared with the tag and if there

---

[1] A regular expression (often abbreviated as regexp or regex) is a string that describes a whole set of strings, according to certain syntax rules. These expressions are used by many text editors and utilities to search bodies of text for certain patterns and, for example, replacing strings. The background of regular expressions lies in automata theory and formal language theory.

```
...
letter          [A-Za-z]
integer         [0-9]
identifier      {letter}({letter}|{integer}|\.|_)*
...
```

Figure 5.2: Example Flex regular expressions for BCPL

are the same the scope is closed. If they are different, the closing tag will be put back onto the input stream so that the next call to the scanner will attempt to match it against the new value at the top of the stack. The tracking of section brackets, and hence scope, is also important when adding and looking up entries in the symbol table. Therefore, an identifier or other allowed entity is only inserted into the symbol table if it is the first declaration or instance in that scope. If an entry already exists for that scope, a reference is made to the original entry.

It is possible to include a file in the source text of a program by using the get directive, which is comparable to the `#include` feature in C. The get directive is handled in the scanner by using a feature in Flex that provides a mechanism for conditionally activating rules called exclusive start states. A condition is declared in the definitions section of the scanner and then when the state is entered (when the `BEGIN` action is executed) only rules qualified with the exclusive start condition will be active. This enables the creation of 'mini-scanners' within the main scanner and are used extensively for our project. The other instances are for character constants, string constants and 'eating' single-line and multi-line comments. By having certain rules for certain states, it is possible to scan portions of the input that are syntactically different from the rest (Paxson 1995). The benefit of using the exclusive state for the get directive is that you have more control over the switching of inputs and also you are able to control the depth of the nesting level.

Some of the commonly-used extensions to BCPL have been implemented in the scanner, including:

- The option of using lower case letters, dots and underscores in identifiers and section bracket tags. This was done by adjusting the regular expressions governing identifiers and adding these to the character classes.
- The option of writing numeric constants in binary, octal or hexadecimal format. Again, this was done with changes to the character classes by creating new expressions for the new formats. A small change in the parser grammar was required to ensure the new numeric formats were recognised.
- The option of using '`?`' to represent an undefined constant, which may be different each time it is evaluated. Any constant expression containing an

evaluated '`?`' is itself a '`?`'. This was implemented by allowing the scanner and parser grammar to recognise '`?`' as a valid constant, but without explicitly accepting a value.

- The full set of comment forms extend either from `//` or `||` or `\\` to end of line, or from `/*` or `|*` or `\*` to the corresponding close symbol (the two characters reversed). This is done by adding these formats to the exclusive states used for comments. Comments are not allowed to nest, and all comment symbols other than the required close symbol are ignored in a comment.

Implementation of other language extensions have required small changes in the scanner or parser. Examples include the addition of an infix byte operator, floating-point operators (the same as the integer equivalents, prepended with '#'), field selectors (which allow quantities smaller than a whole word to be accessed with reasonable convenience and efficiency (Richards 1973)) and combined operator-assignments (the most recent implementations allowed the assignment symbol to be prefixed with any dyadic operator, for example `+:=`).

Simple rules have been created to ignore whitespace and remove comments from the input file, but both the line count and the column count are tracked so as to provide debugging location information for errors. As discussed in the design, the scanner does not perform any backtracking for error recovery purposes. This technique can be time-consuming and not particularly effective. The technique of accepting illegal input and then reporting it with an error or warning is a powerful one that can be used to improve the error reporting of the compiler (Levine, Mason & Brown 1995). Therefore, on finding an error, the line in which it occurs is skipped and the scanner resynchronises and continues scanning.

5.4 Symbol Table

As discussed in Chapter 4, the purpose of the symbol table is to store information about certain entities in the program throughout the entire compilation process. A discussion has already been made about the importance of efficiency in symbol table design, so the choice of using a separately chained hash table is a good compromise between performance and ease of implementation. We have decided to reuse an existing symbol table, which has been taken from (Louden 1997) and adapted for our purposes. The main functions implemented are:

- An insert function to add a symbol, its scope and source code line number to the table.

- A lookup function to check if a symbol exists in the table; if so, the token is returned.

- A similar boolean lookup function to check if a token exists, returning true or false.
- A delete function to mark an entry as inactive, but not remove it from the symbol table.
- A size function to return the number of entries in the symbol table.
- A boolean function is check if the table is empty.
- A destroy function to free all allocations made in the symbol table
- A print table function to dump all symbol table entries to a file, listing their value, scope and line number.

The issue of scoping within the block structure was handled by inserting the scoping 'depth' for each variable as tracked by the scanner. This means that variables are only added when they are the first instance in that particular scope. (Fischer & LeBlanc 1991) discusses the fact that in block-structured languages you do not usually have the scanner enter or lookup identifiers in the symbol table because it can be used in many contexts. It may not be possible, in general, for the scanner to know when an identifier should be entered into the symbol table for the current scope or when it should return a pointer to an instance from an earlier scope. However, it has been possible to resolve the scope of the variable by the tracking of the section brackets and also by following the lexical scoping rules. The idea suggested of allowing individual semantic routines to resolve the identifier's intended usage is not necessary.

The two important data structures which store the entries in the hash table are `LineListRec` and `BucketListRec`, as shown in Figure 5.3. `LineListRec` is a list of line numbers of the source code in which the identifier is referenced and is stored as one of the fields in a `BucketListRec`. This is the record in the bucket lists for each identifier, and details the symbol name, scope and line list.

A good hash function for the symbol table is of vital importance if efficiency and collision avoidance is required. The hash function converts a character string into an integer is the range zero to one less than the size of the table. Hashing can provide almost direct access to records, which means that, on average, a lookup can require just one or two probes into the symbol table to obtain the record. The hash function that we have used in our implementation is shown in Figure 5.4 and works through the character string to give an integer key by shifting modulo the table size. This is not the most complicated or efficient hashing function, but it is suitable for our purposes. As mentioned in the symbol table design, the table size has been set to a prime number (with the closest prime to 1024 being 1021) to possibly improve performance. Pre-loading of the symbol table with all of the

```
        typedef struct LineListRec
        {
                int lineno;
                struct LineListRec* next;
        } *LineList;


        typedef struct BucketListRec
        {
                TOKEN* token;
                LineList lines;
                struct BucketListRec* next;
        } *BucketList;
```

Figure 5.3: Symbol table data structures

keywords and commands was discussed during the design phase, but it did not seem to be particularly worthwhile, especially as the performance benefits are debatable.

```
static int hash (char* key)
{
        int temp = 0;
        int i = 0;
        while (key[i] != ’\0’)
        {
                temp << SHIFT) + key[i]) % TABLE_SIZE;
                ++i;
        }
        return temp;
}
```

Figure 5.4: Symbol table hash function

5.5 Parser

The purpose of the parser is to analyse the tokens identified by the scanner in order to determine the semantic structure of the language and build the intermediate representation. In most compilers, the major portion of semantic analysis is to do with types inference and checking. For BCPL, this is obviously not an issue, but the role of the parser is important in ensuring the syntactic and semantic preservation of the language via the grammar. As we have chosen to build GENERIC trees immediately rather than via our own custom tree, the process of parsing and the generation of the intermediate representation are combined into the single phase. Therefore, when grammar production rules are fired, GENERIC nodes are built and when a top-level production is accepted, a GENERIC tree is returned.

The choice of using Bison to implement the parser was again trivial. Flex and Bison are designed to integrate and work together and still remain the standard tools for creating scanners and parsers. A Bison input file consists of four main sections, *C declarations*, *Bison declarations*, *grammar rules* and *additional C code* (Donnelly & Stallman 2002). The important logic within the parser is contained within the grammar rules section. These production rules describe the grammar of the language in terms of terminals and nonterminals and how these are related and how they decompose. An example set of production rules from the BCPL parser are shown in Figure 5.5, which descend from two purposely created top level constructions called *program* and *program element* which are the entry points to the parser. This example demonstrates how declarations in BCPL decompose to four different types and would eventually decompose to a nonterminal symbol.

```
...
...
declaration            : simult_declaration
                       | manifest_declaration
                       | static_declaration
                       | global_declaration
                       ;

simult_declaration     : LET definition AND definition
                       | LET definition
                       ;
...
...
```

Figure 5.5: Bison production rules for declarations in BCPL

Recursion is another important property when developing the production rule. A rule is recursive when a result nonterminal appears also on its right hand side (Donnelly & Stallman 2002). Nearly all Bison grammars need to use recursion, because it is the only way to define a sequence of any number of particular entity. This is demonstrated in Figure 5.6, where manifest lists in BCPL can be composed of one or more manifest items. Unfortunately, many of the rules given in the original BNF had no termination from their recursion (with examples being many of the expressions and lists). This required the addition of single ’drop-out’ nonterminals on the right hand side of the rule, as highlighted again by the manifest lists in Figure 5.6.

```
...
...
manifest_list   : manifest_list SEMICOLON manifest_item
                | manifest_list manifest_item
                | manifest_item
                ;

manifest_item   : IDENTIFIER EQUALS const_expr
                ;
...
...
```

Figure 5.6: Bison production rules demonstrating left-recursion for manifest lists in BCPL

As previously declared, we are using the BNF definition for BCPL as given in Appendix C, which has been adapted from (Richards & Whitby-Strevens 1980). On implementation, parts of the original version of the grammar were ambiguous (and hence could not be parsed by a LALR(1) parser) and did not represent all of the original language features. At first this was demonstrated by being unable to parse simple code examples, but was later indicated by the large number of reduce/reduce and shift/reduce conflicts. By creating precedence and associativity rules within the Bison declarations section, it was possible to overcome these some of conflicts and use Bison's internal conflict resolution rules. However, as of 2004-04-05, the parser reported 48 shift/reduce and 118 reduce/reduce conflicts. This is not an ideal situation, but at present it does not cause errors when tested with a wide range of code examples. Bison warns when there are conflicts in the grammar, but most real grammars have harmless shift/reduce conflicts which are resolved in a predictable way and can be difficult to eliminate (Donnelly & Stallman 2002). Coupled with the fact that Bison is designed to resolve shift/reduce conflicts by choosing to shift (unless otherwise directed by an operator precedence declaration), means that while the parser may require some attention it by no means indicates that the parser is broken. A good example of this would be before the GCC C++ parser was rewritten to use recursive descent for the 3.4.0 release, its Bison-generated parser reported over 30 shift/reduce conflicts and over 50 reduce/reduce conflicts.

A more significant problem with the original BCPL grammar is that not all of the original language features are represented in the BNF definition, along with all of the commonly-used extensions. Some of the extensions implemented have already been described in Section 5.3, but certain features required changes to the parser. Some of the more important examples include:

- The most commonly-used keyword synonyms are for `THEN` and `DO` which are interchangeable in all circumstances, as are `OR` and `ELSE`. Also, `THEN` or `DO` may be omitted if immediately followed by another keyword or block (Willers & Mellor 1976). This was done by replicating the rules involving `THEN` and `DO` and checking that the proceeding token was 'safe' and not one which would change the semantics of the statement from if the keyword was present. Appendix D gives a listing of some of the common operator synonyms.

- A particularly difficult language feature in BCPL allows the programmer in most cases to dispense with separating a command or line with a semicolon. This is an important point to note, because unlike C, the semicolon in BCPL is a separator rather than a terminator. The original BCPL compilers worked with the premise that if a semicolon is syntactically sensible at the end of a line, and one is not already there, it was inserted (Feather 2002). However, in order that the rule allowing the omission of most semicolons should work properly, a dyadic operator may not be the first symbol on a line (Richards 1973). This was implemented by an addition to the grammar allowing rules without semicolons for declaration parts and command lists. Unfortunately, this choice of implementation could possibly have been the cause for a number of reduce/reduce conflicts present in the parser. The reason for this is due to the fact that for two rules (depending on whether or not there is a semicolon between the declarations and the commands), the actions in each of these will be treated as different states even though they are the same. This leads to a reduce/reduce conflict, as the correct resolution cannot be known at the time the action must be performed because it depends on whether or not there is a semicolon between the declarations and commands. A similar situation occurs with the procedure definitions, as the action must be performed before it is known whether the definition is for a function or a routine (Page 2003). This would have have potentially been a trivial problem if it were a recursive descent parser, because it could have been possible to have simply set a flag when it expected a semicolon, to indicate to the scanner that it is possible to accept a newline instead of a semicolon.

- Certain constructions in BCPL can be used only in special contexts and not all of these restrictions are defined in the BNF. The important examples of this problem are that `CASE` and `DEFAULT` can only be used in switches and `RESULTIS` only in expressions (Richards & Whitby-Strevens 1980). Another interesting example of this would be the necessity of explicitly declaring all identifiers in a program.

A syntactic ambiguity arises relating to the repeated command constructs shown in Figure 5.7. Command `C` is executed repeatedly until condition `E` becomes true or false as implied by the command. If the condition precedes the command (`WHILE` or `UNTIL`), the test will be made before each execution of `C`. If it follows the command (`REPEATWHILE` or `REPEATUNTIL`), the test will be made after each execution of `C`. In the case of line 3 in Figure 5.7, there is no condition and termination must be by a transfer or `RESULTIS` command in `C`, which will usually be in a compound command or block. The resolution comes around by declaring that within `REPEAT`, `REPEATWHILE` and `REPEATUNTIL`, `C` is taken as short as possible. Thus, for example `IF E THEN C REPEAT` is the same as `IF E THEN $( C REPEAT $)` and `E := VALOF C REPEAT` is the same as `E := VALOF $( C REPEAT $)`.

```
WHILE E DO C
UNTIL E DO C
C REPEAT
C REPEATWHILE E
C REPEATUNTIL E
```

Figure 5.7: Syntactic ambiguity in BCPL repetitive commands

When an error is detected, the Bison parser is left in an ambiguous position. Therefore, it is unlikely that meaningful processing can continue without some adjustment to the existing parser stack. There is no reason why error recovery is necessary, but it may be possible to improve productivity for the programmer by recovering from the initial error and continue examining the file for additional errors. This technique shortens the edit-compile-test cycle, since several errors can be repaired in each iteration of the cycle (Levine et al. 1995). By using the `error` token provided by Bison for managing recovery, we can attempt to find a synchronisation point in the grammar from which it is likely that processing can continue. Unfortunately, the emphasis is on *likely*, as our attempts at recovery may not remove enough of the erroneous state to continue and the error states may cascade. At this stage, the parser will reach a point where processing can continue or it will abort. The use of the `error` token means that after reporting a syntax error, the Bison parser discards any partially parsed rules until it finds one if which it can shift the `error` token. It then performs resynchronisation by reading and discarding tokens until it finds one which can follow the `error` token in the grammar (Levine et al. 1995). A key point, therefore, is the placement of error tokens in the grammar. The two conflicting goals are between placing it at the highest level possible, so that there will always be a rule to which the parser can recover, against the fact you want to discard as little input as possible before recovering by minimising the number of partially matched rules the parser has to discard during recovery (Donnelly & Stallman 2002). It was decided to place the `error` token at the highest level possible and resynchronise after the next newline, so as to aid maximum recovery during the development stages of the compiler. This method of ad-hoc recovery provides the opportunity to anticipate the possible occurrences of errors and in particular their position. A more detailed discussion of ad hoc and syntax directed error recovery techniques can be found in (Tremblay & Sorenson 1985).

The most difficult task of the parser is to start to build the intermediate representations to pass onto the latter stages of the compiler. This part of the development is where the most technical knowledge is required, as Flex and Bison have been supported by the long usage of these tools and the large amount of resources available. However, the elegance of re-sourcing GCC is matched by its immediate difficulty with both the building of the intermediate representations and the integration into the main toolchain. The possible implementation of the intermediate representations is discussed in detail in the following section.

5.6 Intermediate Representations

5.6.1 Overview

The development of the project was forked into two at this stage, as it was deemed important to ensure a contingency plan was in place if problems occurred in the implementation and integration of the GCC part of the project. Therefore, the code was frozen at a stage where the scanner and parser could parse a wide range of BCPL test code. The side branch consisted of the GENERIC tree building and the rudimentary BCPL system library with other general improvements to the compiler infrastructure.

Our main resource of existing code is from the C and Fortran front ends[2] taken from the GCC 3.5.0 daily snapshot (previously `tree-ssa` branch, available from (GCC 2004*b*)). The C front end uses a strange fusion of GENERIC and GENERIC trees in the same routines (see Section 2.4.2), plus a handful of language specific tree codes defined in '`c-common.def`'. We have also investigated the `gimplify_expr` function, which either returns GIMPLE directly or a GENERIC version of the given node to be handled by the `gimplifier`. This seemingly poor modularity has arisen as an artifact of the way the infrastructure evolved, as the C front end was the driver for the development of the `tree-ssa` branch. The Fortran front is probably a purer implementation of the infrastructure, but because of the relationship between BCPL and C, it makes sense to study and reuse the code for similar language constructs. Therefore, when presented with a C or C++ source program, GCC parses the program, performs semantic analysis (including the generation of error messages), and then produces the intermediate representation. This representation contains a complete representation for the entire translation unit provided as input to the front end. It is then typically processed by the code generator in order to produce machine code (Stallman 2004*a*). A more detailed discussion about the in-

---

[2] Details about the format of the GCC source directory can be found at `http://gcc.gnu.org/onlinedocs/gcc-3.4.0/gccint/Top-Level.html`, but most of the main source of GCC itself is contained with the *gcc* subdirectory. The format of the GCC subdirectory can be found at `http://gcc.gnu.org/onlinedocs/gcc-3.4.0/gccint/Subdirectories.html`.

tegration of the front end to the rest of the GCC toolchain in described in Section 5.7.

The main purpose of GENERIC has been to provide a language-independent way of representing entire function in trees (Stallman 2004*a*). As mentioned earlier, a problem with using GENERIC is the apparent lack of documentation and the fact that many parts of it are still under active development. An unfortunate quote that seems to sum up the situation is if you are able to express it with the codes in the source file `gcc/tree.def`, it is GENERIC. It is hard to think of a more difficult situation then when writing code in a language that is only defined in its own code implementation. The existing front end ’how-to’ documents have yet to be updated to reflect the major changes in the intermediate representations and as such are irrelevant for this stage of the implementation. GIMPLE is a simplified subset of GENERIC for use in optimisations. The particular subset chosen was heavily influenced by the SIMPLE intermediate language used by the McCAT compiler project at McGill University (Hendren et al. 1992). In GIMPLE, expressions are broken down into three-address form, using temporary variables to hold intermediate values for use in the later optimisation phases. Also, control structures are lowered to `goto`s (Stallman 2004*a*).

5.6.2 Trees

The central data structure used by the internal representation is the tree. These nodes, while all of the C type `tree`, are of many varieties, as tree can be a pointer to a variety of different types. The `TREE_CODE` macro is used to tell what kind of node a particular tree represents. The front end needs to pass all function definitions and top level declarations off to the middle end so that they can be compiled and emitted to the object file. This means that this is done as each top level declaration or definition is seen, as GENERIC represents *entire* functions as trees. The BCPL front end at present only uses rudimentary GENERIC trees for certain constructs in the grammar and attempts to attack some of the more difficult language features by working from the C and Fortran front ends. It may be necessary to create some language-dependent tree codes for our implementation, as language features such as `RESULTIS` and `SWITCHON` have caused problems on implementation. The question of certain library functions, such as `LONGJUMP` and `LEVEL` may also require some special handling. This is an acceptable approach, so long as these custom codes provide hooks for converting themselves to GIMPLE and are not expected to work with any optimisers that run before conversion to GIMPLE. Currently, all optimisation passes are performed on the GIMPLE form, but it would be possible for some local optimisations to work on the GENERIC form of a function; indeed, the adapted tree inliner works fine on GENERIC, but the existing GCC compiler performs inlining after lowering to GIMPLE (Stallman 2004*a*).

As stated previously, we have only been able to implement a small subset of the functionality of BCPL in GENERIC tree form. This has been partly due to a very steep learning curve on working with GENERIC, but also problems of actually validating the trees produced. Therefore, the following set of implementation choices have been made concerning particular features of the BCPL language. This is not an exhaustive list of either the intended implementation, or the functionality of GENERIC. A detailed description of the important nodes and constructs in GENERIC can be found in (Stallman 2004*a*).

5.6.3 Identifiers

Identifiers have been implemented in GENERIC by using the `IDENTIFIER_NODE`. This, however, represents a slightly more general concept than the standard C or C++ concept of an identifier, which is turn is slightly different for BCPL, as an `IDENTIFIER_NODE` may contain a '`$`' or other extraordinary characters (Stallman 2004*a*). An important point is that there are never two distinct `IDENTIFIER_NODEs` representing the same identifier, so it is possible to use pointer equality rather than using a routine like `strcomp` to compare `IDENTIFIER_NODEs`. Two important macros that are available whilst working with `IDENTIFIER_NODEs` are the `IDENTIFIER_POINTER` which is the `NUL`-terminated string represented by the identifier as a `char*`; and the `IDENTIFIER_LENGTH`, which is the length of the string, not including the trailing `NUL`, returned by `IDENTIFIER_POINTER`. Therefore, the value of `IDENTIFIER_LENGTH(x)` is always the same as `strlen(IDENTIFIER_PO`
2003*d*).

5.6.4 Declarations

There are two basic kinds of declarations in BCPL, section declarations and simultaneous declarations. Identifiers declared in section declarations have a scope that starts at the end of that identifier's declaration; identifiers declared in simultaneous declarations have a scope that starts at the first of the set of simultaneous declarations. In each case, the scope ends at the end of the block the declaration occurs in, or the end of the file if not in a block (Feather 2002). All declarations except function declarations are implemented similarly in GENERIC. The macro `DECL_NAME` returns an `IDENTIFIER_NODE` giving the name of the entity stored. Most declarations in BCPL can be easily handled by the various kinds of declaration nodes available, such as `LABEL_DECL` to define labels such as label prefixes, `CASE` labels and `DEFAULT` labels. It may be possible to use the `RESULT_DECL` node to implement `RESULTIS`, which is used with `VALOF` blocks, by converting it to a function declaration and then this could be used as the return value for the `VALOF` block. However, this is currently unimplemented, but remains a potential approach for implementation.

There exists scoping issues with declarations in BCPL, especially with rules that exist for certain language features. For example labels have the form of an identifier followed by a colon and any number of labels may precede a command. A label is only in scope within the smallest of:

- the commands (but not declarations) of the textually surrounding block.
- the body of the textually surrounding routine (if any).
- the body of the textually surrounding `VALOF` (if any).
- the body of the textually surrounding `FOR` (if any)

These rules may be difficult to implement with the GENERIC framework and at present are not fully implemented. It may be possible to use the predicates that control the level of scoping, such as `DECL_CLASS_SCOPE_P`, which holds if the entity was declared at a class scope; or the predicate `DECL_FUNCTION_SCOPE_P`, which holds if the entity was declared inside a function body. Nevertheless, this requires more development time and consideration.

A valuable feature of BCPL is the existence of manifest constants which give the ability to use names to represent constants. The value associated with a manifest constant stays fixed, so you cannot assign to it (Richards & Whitby-Strevens 1980). This is similar to the C preprocessor directive `#define`, except they are restricted to values which can be stored in a single cell and the scope of a manifest constant is the same as that of a identifier declared in the same place. The problem with the implementation of manifest constants is the fact that they can be used in constant expressions, which are required at compile-time rather than run-time. Further research is required to see if functionality exists to provide a binder of sorts that encapsulates the manifest constant and then this can be replaced by its value wherever it is used.

5.6.5 Functions

Both functions and routines exist in BCPL, with arguments passed strictly by value. The corresponding difference between the two is that a functional application is an expression and yields a result, whereas as a routine call is a command and does not. In common practice, the word procedure is used to mean either in contexts where the distinction is unimportant (Richards 1967). However, in GENERIC, they are represented as functions by using the `FUNCTION_DECL` node. Arguments are accessed using the `DECL_ARGUMENTS` node, which returns the `PARM_DECL` for the first argument to the function (GCC 2003*d*). As in C, nested functions in BCPL can be handled by using the `DECL_CONTEXT` macro, which indicates that the context of a function is another higher-level function and that the GNU nested function extension is in use. One problem with this is that nested functions can refer to local variables in its containing function; in certain contexts, this may potentially violate some of the BCPL lexical scoping rules. In BCPL it is legal to nest procedures, but all variable references must be fully global or fully local: inside an inner procedure, variables local to the outer procedure are out of scope. Thus in comparison to C, you get some additional name hiding, but the compiler does not have to maintain static links (Feather 2002). Some further investigation is required to ensure that the scoping semantics are robust for the choice of implementation, but the range of constructs and predicates available in GENERIC should enable an appropriate solution. For example, `DECL_LOCAL_FUNCTION_P` can be used if the function was declared at block scope, even if it has global scope (Stallman 2004*a*).

5.6.6 Statements

Corresponding tree nodes exist for all of the source-level statement constructs used within the C and C++ front end. This means that most, if not all, of the BCPL statements should fit into one of these nodes. In GENERIC, a statement is defined as any expression whose value, if any, is ignored (Stallman 2004*a*). A statement will always have a `TREE_SIDE_EFFECTS` set (or it will be discarded), but a non-statement expression may also have side effects. Many of the statements will have sub-statements when they are represented in a GENERIC tree. For example, a BCPL `WHILE` loop will have a body, which is itself a statement. If the sub-statement is `NULL_TREE`, it is considered equivalent to a statement containing a single '`;`' i.e. an expression statement in which the expression has been omitted. Other implementation examples are `DECL_STMT` for local declarations, with a GCC extension allowing declaration of labels with scope. Also `DO_STMT`, `FOR_STMT`, `GOTO_STMT`, `IF_STMT`, `RETURN_STMT`, `SWITCH_STMT` and `WHILE_STMT` to represent some of the common unlabelled commands in BCPL. Scoping is handled by using the `SCOPE_STMT` node; therefore, when a new scope opens (for example when entering a new block), the predicate `SCOPE_BEGIN_P`, will be set and at the end of a scope, the predicate `SCOPE_END_P` will hold (GCC 2003*d*).

5.6.7 Expressions

The GENERIC representations for expressions are for the most part quite straightforward and nodes exist for most of the common expressions in BCPL. However, it is important to consider the fact that the expression ”tree” is actually a directed acyclic graph (DAG). This means that there may be numerous references to a node throughout a source program, many of which will be represented by the same expression node. Therefore, you cannot rely on certain kinds of node being shared or being unshared (Stallman 2004*a*). The macro `TREE_TYPE` is used to return the type of the expression stored in the node. The range of GENERIC nodes are far too numerous to list here, but again most, if not all BCPL expressions are covered. For example, conditional expressions are handled by `COND_EXPR` and function calls by `CALL_EXPR`. String constants are handled by the `STRING_CST` node, but it is important to consider the implementation of strings in BCPL. They are word packed and not `NUL`-terminated, with the first byte storing the length of the string.

A function for converting between BCPL and C strings has been implemented in this project. In some implementations that did not have an infix byte operator, the only way to reference individual characters was to shift and mask(Feather 2002).

BCPL's logical and relational operators have direct equivalents in C and can be easily represented in GENERIC nodes. However, the clear distinction in C between the bitwise and boolean forms is not present in BCPL, as the distinction depends on context (Richards 1973). In C, `true` can be represented by an non-zero value, whereas it is useful in BCPL for the value representing `true` to be the bitwise negation of the value representing `false`. An example of this would be when a BCPL program assigns a value from a logic expression to a variable that is then used as the test in an `IF` statement, the test will have the desired result. It is worth noting that this behaviour is not actually defined by the language definition and the result can be implementation dependent (Richards & Whitby-Strevens 1980). Therefore, in BCPL, logical operations are evaluated according to one of two rules. If the result of the operator is in *truth context*, then the truth rules are used. Otherwise the operator takes bit-patterns as arguments and operates on each bit separately. Truth context is defined as:

- left hand side of '$\rightarrow$' (implication)
- outermost operator in the controlling expression of a conditional command
- a direct operand of an operator evaluated using truth rules.

This situation also requires further investigation, as it is imperative that the implementation of the logical operators is consistent and semantically correct. A trivial implemented example is of bitwise equivalence (`EQV`) in BCPL, which can be easily represented as the negation of an XOR node, `BIT_XOR_EXPR`.

5.6.8 Global Vector

The global vector, where system and user variables are stored in fixed numerical locations, is the sole means of communication between separately compiled segments of program (Richards & Whitby-Strevens 1980). The first 150 locations are usually allocated to the operating system for many of the functions in the system library, but the declaration `GLOBAL $( N1:K1 $)` associates the identifier `N1` with the location `K1` in the global vector.

There are two major challenges in dealing with the implementation of the global vector. Firstly, it is allowed to be arbitrarily large, so its size is not known until all of the modules in a program have been compiled. Secondly, that a global declaration results in an identifier referring to the cell within the global vector and not a variable containing a pointer to a cell, though this can be dealt with in a similar way to manifest constants. The proposed implementation is to create a C library wrapper that declares an `extern` variable to point to the global vector in the BCPL 'world'. This then gives us an opportunity to preallocate the essential operating system library functions into the global vector. Also, by having a variable that points to the base of the global vector throughout compilation, we know that once it has finished and the size of the global vector is known (as all modules would have been compiled and linked), it would be possible to generate the code to create the global vector and start the program.

5.6.9 System Library

Most BCPL implementations comprise a set of basic procedures, together with a standard library written in BCPL. These basic procedures provide a means of accessing the operating system functions and machine-level facilities such as input and output (Richards & Whitby-Strevens 1980). The system library has been built up by consensus over the years, so it is not always possible to guarantee a particular procedure always being in a particular place in the global vector. All system routines and functions are accessed via the global vector which is populated by the global declarations in the system header `LIBHDR`. The allocation of routines and functions to particular cells is implementation defined, but only cells less than `FIRSTFREEGLOBAL` (a manifest constant) are used (Feather 2002). The chosen method of implementation is obviously related to how the global vector is implemented, as that is the container for all of the procedures. Since it is possible to access the externally declared global vector, we can simply pre-populate it with our own implementations of the critical functions. This means that the C library wrapper essentially re-implements the base functions such as `RDCH` (input) and `WRCH` (output) as calls to the C I/O library. All we need to do is pass a pointer to these calls into the global vector and then it will be possible to access this functionality from within the BCPL program. Examples of this would be setting `CIS` (Current Input Stream, usually global position 24) and `COS` (Current Output Stream, usually global position 25) to `stdin` and `stdout` respectively. The two most important functions that have been implemented as C system calls are for input and output. `RDCH`, which returns the next character from the selected input stream, and `WRCH`, which writes the character to the selected output stream. These are handled by declaring functions that call the C library functions `getc` and `putc`. Pointers to these functions are then passed to the relevant position in the global vector and then can be called as normal. This means that input and output functions such as READN and WRITEN should work, as these just make calls to `RDCH` and `WRCH` respectively. Other important implementations have been for the entry and exits points to a BCPL program, `START` and `FINISH`.

While it remains to be seen how well this will work when implemented in GENERIC, the choice of implementation is sufficiently robust enough to cope with getting a subset of BCPL library function working to try and test programs. Other possible implementations have been to physically rewrite the whole BCPL library in C, but this was deemed to be too time-consuming even if it would probably be fairly efficient. The idea of attempting to look at the BCPL application binary interface and see if it would be possible to link in the existing library at compile-time are not unrealistic, but would require a large amount of investigation into how the BCPL binaries are constructed and how they can be linked.

5.6.10 Miscellaneous

This section describes miscellaneous language features and how they have been implemented in the intermediate representation.

It is hard to discuss writing a compiler for BCPL without tackling the problem of converting from word addressing to byte addressing when required. The use of word-length data objects (called *cells* in BCPL) can cause problems on byte-addressed machines, as BCPL defines consecutive words in memory to have numerically consecutive addresses. Hence, given a pointer to one such word, `p`, you can access the pointer to the next words by writing `p+1`. This is the same as for a `VECTOR` object in BCPL, as this is represented by a variable holding a pointer to its first word. So if `v` is a `VECTOR` then the value `v` points to the first word, `v+1` to the next, and `v+i` to the `i`'th. The dereferencing operator ('pling') in BCPL is written `!`, so `!(v+i)` gets you the `i`'th component of the `VECTOR v`. This can also be written as `v!i`, using the dyadic form of `!`. Recall, however, that BCPL is typeless. The compiler would not know that `v` is a vector and `i` an index; they are both just bit patterns. Hence, the expression `v+i` looks just like normal integer addition. This, therefore, has a strange consequence: if `v+i` is the same code, regardless of whether `v` is an integer or a vector, then the BCPL pointers to adjacent words must indeed differ by 1. This means that on a machine with an underlying byte-addressing scheme, BCPL pointers cannot be true addresses, but must be scaled by 2 (on Intel x86,PDP-11 for example) or by 4 (on the MC68000 for example) before being dereferenced. This means that indirection and addressing referencing is a problem, but as long at all references are shifted before calculations are made, then it should be possible to use the `ADDR_EXPR` and the `INDIRECT_REF` nodes. Again, these features require more work before full functionality is restored.

An interesting language feature in BCPL is the `TABLE` operator, which gives an initialised, permanently allocated vector. It is also one of the few operators in BCPL that associates right to left (see Appendix D). The expression `TABLE k1,k2,...kn` returns the address of `n` contiguous cells initialised to `ki` in order. Since all the values are constant expressions, they must all be evaluated at compile-time. A possible way of implementing this in C could involve using the GNU extension `({ ... })`, which allows several statements between the two brackets and yields an expressions which is the value of the last one (ensuring you take into account the shift required to convert to word address on 32-bit machines). There does not seem to be a simple way of implementing it in ANSI C (Manning 1999). It may be possible to implement it in GENERIC as a function declaration which returns the appropriate value, but this raises problems as we are in an expression, so declarations are not allowed, so the function must be declared before the current function. Also, the values, though constant, could still depend on manifest constants which are in scope only in the current function. The BCPL `VALOF` expression also encounters similar problems. The expression `VALOF c`, where `c` is a command, causes `c` to be executed. If, during the execution of `c`, a `RESULTIS` command is reached, that controls the value of the expression, otherwise the value is unspecified. Therefore, it may be possible to implement the `VALOF` construction as a type of function declaration, but more research is required for both this and the implementation of the `TABLE` operator.

The infrastructure for handling errors in the intermediate representation is done by the special tree `error_mark_node`, which has the tree code `ERROR_MARK`. If an error has occurred during front end processing, the flag `errorcount` is set. If the front end encounters code that it can not handle, it will issue a message to the user and set the flag `sorrycount`. When these flags are set, any macro or function which normally returns a tree of a particular kind may instead return the `error_mark_node`. At present, no processing of erroneous code is implemented, but this would be possible by dealing with the `error_mark_node` (Stallman 2004*a*).

5.7 GCC Integration

The implementation of this part of the project has not advanced beyond a theoretical stage, due to problems in the implementation of the previous sections. However, it is possible to describe some of the important features that need to be addressed and discuss some potential problems.

The existing GCC infrastructure (up to GCC 3.4.0) is shown by Figure E.1 in Appendix E. This shows the how each language front end goes from the language-specific tree to RTL, before being passed to the back end for optimisations and code generation. However, as previously described in Section 2.3 and Section 2.4.2, the new framework that will be released as GCC 3.5.0 has some significant infrastructure changes. This new structure is shown by Figure E.2 in Appendix E. The three main intermediate languages that GCC uses to represent the program during compilation are GENERIC, GIMPLE and RTL. GENERIC serves as the interface between the parser and the optimiser, while GIMPLE and RTL are used during program optimisation. Most of the work of the compiler is done on RTL, with the instructions to be output described in an algebraic form that describes what the instruction does. GIMPLE is used for target and language independent optimisations, so we do not need to worry about the lowering pass from GENERIC to GIMPLE, as this is a built-in function (`gimplify_function_tree`). Our only difficulty may occur if we decide to implement special language-specific tree functionality, as we would need to provide our own method of lowering to GIMPLE. Therefore, a large discussion of GIMPLE is not necessary, but further details about its nodes and format can be found in (Stallman 2004*a*), while a rough GIMPLE grammar can be found in (Merrill 2003). This therefore means that once the GENERIC representation is built and lowered to GIMPLE form, it can be passed onto the GCC infrastructure and automatically handled. The only actions that would require attention would be the handling of errors, but if the GENERIC form had been built correctly and lowered without any problems, they would more likely be errors outside of our scope and control.

Most of the tree optimisers rely on the data flow information provided by the Single Static Assignment (SSA) form. GCC implements the SSA form as described in (Cytron et al. 1991), which requires a conversion from the GIMPLE format. The compiler modifies the program representation so that every time a variable is assigned in the code, a new version of the variable is created.The process is controlled by the source code in `tree-ssa.c`. Although this process would be automatically controlled within the GCC back end, it is important to understand how the optimisation passes are implemented and the range of optimisations that occur. The SSA form depends on the control flow graph (CFG) data structure that abstracts the control flow behaviour of a function that is being compiled and is built on top of the intermediate code (the RTL or tree instruction stream). The CFG is a directed graph where the vertices represent basic blocks and the edges represent possible transfer of control flow from one basic block to another (Stallman 2004*a*). It is important to ensure that each compiler phase keeps the control flow graph and all profile information up-to-date, as the CFG also contributes to maintaining liveness information for registers. Further details about the SSA form and the optimisations implemented can be found in (Stallman 2004*a*).

Another key feature that has yet to be implemented is the problem of providing an interface for the intermediate representations to the symbol table data. Strangely, this is not actually documented in the GCC Internals Manual (Stallman 2004*a*), but a possible implementation may be to build and attach `BLOCK` nodes, containing declaration chains, to `BIND_EXPR` nodes. This is the technique that is used in the GNU Fortran (`gfortran`) front end (GNU 2004*a*) and may be possible to adapt for our usage.

The problem with the lack of documentation and resources about GCC integration and building GENERIC has been a major problem with this stage of the implementation. The only current and authoritative resources have been the GCC Internals Manual (Stallman 2004*a*) and the Tree SSA online documentation (GCC 2003*d*), which have been used extensively. However, the GCC Internals Manual is currently undergoing major rewrites to actually document many of the features discussed and implemented here. For many of the critical features that required more information, the documentation was sparse or actually missing totally (frustratingly, there are numerous instances of ’documentation incomplete’). This contributed to a very slow development progress for building the intermediate representations and very little progress with the integration into GCC. However, a helpful section in the GCC Internals Manual has been the ’Anatomy of a Front End’, which describes in some detail the configuration files expected by GCC for building and using a new language front end, which has been used extensively. As described earlier, the proper way to interface GCC to a new language front end is with the `tree` data structure, described in the source files `tree.h` and `tree.def`. Sadly, there remains many sections that are incomplete and it freely admits it is only preliminary documentation.

Chapter 6

System Testing

While the latter parts of this dissertation has focused on developing a new front end for GCC, a key requirement has always been ensuring that the semantics of the BCPL language are preserved. This has been a cause of many of the implementation problems encountered, from scoping issues to word addressing. Because of the problems encountered whilst building the intermediate representations and with the integration of GCC, the lack of existing framework to support exhaustive testing means that the main focus has been on testing the scanner, parser and symbol table. Therefore, it was felt that the testing performed on the working system should focus on determining if the system meets the requirements as specified in Chapter 3. The results for certain tests can be found in Appendix F, though a short overview of the testing process is discussed below.

Verification and validation are the names given to the checking and analysis processes that ensure that software conforms to its specifications (Sommerville 2001). Verification and validation are not the same, although they are often confused. A succinct definition that highlights the differences between the two is that validation can be thought of as ’are we building the right product?’ and verification as ’are we building the product right?’. The two common techniques that we have used for our basic testing strategy are software inspections and software testing. Software inspections are static techniques that involve analysing and checking system representations such as the requirements document, design diagrams and the source code. Software testing involves executing an implementation of the software with test data and examining the outputs of the software and its operational behaviour to check that it is performing as required (Sommerville 2001). Both of these are simple, commonly-used techniques and were deemed appropriate for the type and stage of development. It did not seem particularly worthwhile to spend a large amount of time creating and performing a complex testing strategy when the testing performed throughout development was sufficient. At the time of writing, the compiler is still in a prototype stage and has been developed to explore functionality and for evaluation. Since it has no security or safety critical application, exhaustive testing is not necessarily appropriate and is not one of the key aims of this project. Nevertheless, in the future, it would be sensible to implement some form of formal testing strategy, possibly involving black box testing (see (Sommerville 2001) for more details).

Therefore, the testing and debugging cycles that have been performed during the development have been sufficient to catch a significant amount of errors. This has been helped by the range of option flags set for Flex, Bison and GCC. These have contributed to better debugging in both the scanner and the parser, plus stricter checking of the C code. For example, GCC is invoked with flags to enforce and issue all the warnings demanded by strict ISO C, along with others that report constructions which are not inherently erroneous but which are risky or suggest there may have been an error (Stallman 2004*b*). It is also invoked with the option to produce debugging information for use by GDB (GNU Debugger) and DDD (GNU Data Display Debugger), which has been used for testing, along with Splint[1] which is a tool similar to lint for statically checking C programs for security vulnerabilities and coding mistakes. Flex is invoked in debug mode, so that whenever a pattern is recognised the scanner will output information about the matched rule and the line it occurred on. Messages are also generated when the scanner backs up, accepts the default rule or reaches an end-of-file (Paxson 1995). Bison is also invoked with debugging options set, plus the verbose output option, which writes an extra output file containing verbose descriptions of the grammar and parser (Donnelly & Stallman 2002). This was extremely useful when attempting to find the cause of conflicts and grammar errors, in particular when attempting to resolve the optional semicolon problem (see Section 5.5).

Even though testing whilst building the intermediate representation building and for the GCC integration is still in the debugging stages, certain features have been discovered that would be helpful in the future. The GCC 3.5.0 snapshot was itself built and configured with the `--enable-checking` option set, which means that all tree types are checked at run-time, resulting in a performance penalty (Stallman 2004*a*). This would obviously not be suitable for a release version, but would be extremely helpful in development when testing the tree building functions. The file `tree-pretty-print.c` implements several debugging functions that when given a GENERIC tree node, they print a C representation of the tree. The output is not meant to be compilable, but it would be of great help when debugging transformations done by the tree building passes. Other compiler flags that could possibly be helpful for testing are `-ffump-tree-gimple`, which dumps C-like representations of the GIMPLE form and the `-d` flag which is used for getting RTL dumps at different stages of optimisation during compilation.

In summary, the testing strategy used was based around a significant amount of testing and debugging during the development, plus the use of specially constructed BCPL test code examples. These test examples have attempted to utilise many of

---

[1] Available from `http://www.splint.org/`

the BCPL language features, with the emphasis placed on creating structures likely to cause problems. Some examples of the output from these test cases can be found in Appendix F.

Chapter 7

Conclusion

7.1 Overview

The software developed as a result of this project has provided a starting framework towards creating a BCPL front end for GCC. The scope of this project from the start was massive and this was increased by some of the design decisions made. However, these choices were made because of suitability and elegance of the solution, rather than on time or implementation constraints. The depth of the Literature Survey from Chapter 2 gave us a firm appreciation of the domain and the current status of the GCC infrastructure. It also highlighted the important issues when developing a compiler and reusing an existing toolchain. The range of possible implementation options enabled us to compare and contrast existing compiler architectures and whether they were suitable or appropriate for this project. A summary of some of the key problems encountered during this project are listed below:

- The main problem discovered as we advanced in the project was the incomplete nature of the documentation and resources for GCC 3.5.0. The initial research into the domain highlighted the fact that developing a front end for GCC would not be trivial, but surprisingly for such a well managed and documented software development, the available resources were, at times, disappointing. This was compounded by the fact that the incomplete parts of the documentation for the 3.5.0 release were seemingly the most critical parts, insofar for building the intermediate representations and the integration of GCC. Fairly comprehensive documentation existed for much of the GIMPLE and RTL syntax and functionality, but large sections of the important GENERIC tree building functions and how you then pass these trees to the middle and back end were missing. This meant that for most of the intermediate representation development, only two current sources of information were used for reference: the GCC Internals Manual (Stallman 2004*a*) and the Tree SSA online documentation (GCC 2003*d*). It is understandable that the fast developments over the past few months with GCC and the recent mainline merge would mean that documentation would take a back seat, but it seems unusual that such an important change to the GCC infrastructure is not fully documented. The GCC Mission Statement (GCC 1999) declares its first design and development goal to be ’new languages’, but surprisingly little up-to-date documentation exists on developing new language front ends.

- It could be said that the choice of using GCC as our implementation framework was perhaps flawed, but it was felt that compared to the other potential infrastructures discussed, GCC provided the best solution. The domain research performed in Chapter 2 demonstrated that while the other options such as LLVM, lcc and SUIF were extremely well designed compiler projects, none were comparable to GCC for popularity and support. It would have been interesting to see how the other projects would have developed, but it seems that irrespective of the toolchain used, certain BCPL language features would have been troublesome. Nevertheless, the learning curve of the GCC internals was steep and attempting to understand partially implemented code, or poorly documented code has definitely contributed to less development than anticipated. However, this is tempered by the fact that the research performed lays the groundwork for a future implementation.

- A range of problems were associated with certain BCPL language features, such as the implementation of `VALOF`, `LONGJUMP`, `LEVEL`, the global vector and the system library. It seemed that these had no trivial solution and perhaps were made harder by the modern intermediate representation chosen. Nevertheless, solutions were discussed and it seems that there may exists some GENERIC nodes that could implement some of the features mentioned.

- Definite problems were encountered when attempting to formulate software engineering principles and methods for this compiler project. It seems exceptionally hard to define a reasonable set of requirements and perform elicitation and analysis. It also occasionally seemed a futile exercise attempting to provide a model to adhere to, when in most iterations, the development and design was based around past experience and knowledge of the domain and GCC. Nevertheless, a range of software engineering principles have been applied in the requirements, design and implementation, but it seemed as if the formulation of a rigorous test plan was both unnecessary and impractical.

- Due to the fact that it was necessary to stay current with the versions and snapshots released by GCC, the time taken to configure, build and install GCC became a problem. Each build took upwards of two hours, because it was necessary to ensure a full build was done and all of the necessary tools were built. This impacted on development work, as GCC was built approximately fifteen times during the project.

- Even though it was discussed during the requirements and design phase, it was hard to ensure that a secondary plan was in place, if any problems occurred. As soon as we were committed to GCC, the development had to make changes and alterations to accommodate the infrastructure, so this was the reason the code was forked into two branches. continue.

7.2 Milestones

A summary of some of the key milestones from this project are listed below:

- Development of an efficient Flex scanner for BCPL with utility functions and lexical error handling.
- Development of a Bison-generated parser with syntactic error handling.
- Creation of a separately-chained hash table as the symbol table.
- Formalisation of a BNF for BCPL, developed from multiple sources and incorporating commonly-used extensions to the language.
- Rudimentary implementation of the BCPL system library with the global vector, enabling input/output functionality.
- Preliminary GENERIC nodes built for a subset of the BCPL language, adapted from the GCC C and Fortran front ends.
- Research into the GENERIC/GIMPLE infrastructure and the new optimisation framework for GCC, giving an understanding of the future development of GCC and the integration of new languages.
- Interaction with the GCC developer community via the GCC mailing lists, and communications with the mainline branch maintainers. There seems to have been some positive interest from the GCC community in relation to this project.
- The creation of a SourceForge community project to continue with this project in the future. It should also be possible to get other developers involved with the project and hopefully at some point in the future make an official release.
- Adherence to the appropriate GNU, GCC and ANSI C coding standards.
- Successfully meeting most of the requirements set for this project. Even though this project has not been a total success, we have been able to develop some interesting information about the GCC infrastructure and have shown that this project is viable and achievable within a larger time frame.

7.3 Further Work

The system produced from the work of this dissertation fulfils a number of the objectives as specified in Chapter 1. Due to the time limitations and open-ended nature of this project, there exists a lot of scope for improvements and future work. One major reimplementation would be to change the Bison parser to a hand-written recursive descent parser. After the lessons learnt from implementation in Bison, the benefits from using recursive descent would be significant and would remove a whole host of conflicts within the present parser. As stated previously, it would also benefit the implementation of language features such as the optional semicolon problem (see Section 5.5). Improvements would also be made to the symbol table and also how it integrates and interacts with the rest of the GCC toolchain. It would be important to fully implement the features discussed in Section 5.6, especially improvements to the implementation of the global vector and the BCPL system library. It would be feasible to investigate BCPL's application binary interface and attempt to create a framework where it would be possible to link in the original BCPL library and compile time and call library functions as normal. Also, once the implementation has advanced to a more serious level, it would also be important to develop an improved testing strategy, possibly based upon black box testing or unit testing.

Depending on the changes to GCC in the upcoming year, the `tree-ssa` branch will be released as GCC 3.5.0 (GCC 2004*b*). Full integration for the BCPL front end would still require a large amount of development effort and also some new approaches for implementing certain language features. However, as the active development for GCC mainline advances, the improvement in the documentation may enable this to occur. To enable the continuation of this project and to take advantage of more online collaboration, a SourceForge.net[1] project has been proposed and accepted by the SourceForge maintainers and been named 'BCPL for GCC'. The project website can be accessed at `http://sourceforge.net/projects/gccbcpl/`. It is likely that there will be interest in the SourceForge development, as a similar project to develop a PL/1 front end for GCC (Sorensen 2004) has just released version 0.0.6 and is currently under active development. The benefits of using the SourceForge infrastructure would include the availability of source management tools such as CVS, support management tools, mailing lists, discussion forums, shell services and compile farms to name a few. The impact of these tools on the project would have undoubtedly been positive, as in retrospect, the use of tools such as CVS would have been enormous beneficial to this project, more so when it was forked to handle the GCC development.

---

[1] SourceForge.net is the world's largest open source software development web site, providing free hosting to tens of thousands of projects. See `http://sourceforge.net/` for further details.

7.4 Concluding Remarks

The rapid development and popularity of the system programming language C in the 1980s led to the demise of BCPL. However, the popularity of BCPL as an efficient and flexible programming language means that interest and affection for the language survives today. It is quite possible that if the popularity of byte-addressed minicomputers had not happened in the 1970s, UNIX would have been initially written in BCPL. It is by no means a dead language, as large amounts of legacy code exists today and it remains a popular language in academia. Numerous developments still involve BCPL; attempts have been made to port BCPL Cintcode programs over to the Microsoft .NET Common Language Runtime (Singer 2002).

This dissertation has discussed the history and heritage of BCPL and has looked at the range of modern compiler infrastructures. The use of a modern compiler implementation will provide a proven existing framework to create a robust compilation solution for legacy BCPL users. A discussion of the methods used for the design and implementation of such a system has followed. The final system, even if unfinished, has achieved many of the objectives set out in Chapter 1 and this was briefly demonstrated in the test code examples. The use of the GCC toolchain will provide an elegant solution to the BCPL legacy problem, and would be excellently supported by active development and modern optimisation strategies.

Bibliography

Aho, A. V., Sethi, R. & Ullmann, J. D. (1986), *Compilers: Principles, Techniques and Tools*, World Student Series, 1st edn, Addison-Wesley.

Aigner, G., Diwan, A., Heine, D. L., Lam, M. S., Moore, D. L., Murphy, B. R. & Sapuntakis, C. (2003), An Overview of the SUIF2 Compiler Infrastructure, Technical report, Stanford University.

Alpern, B., Wegman, M. N. & Zadeck, F. K. (1988), Detecting Equality of Variables in Programs, *in* 'Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages', ACM Press, pp. 1–11.

Barron, D. W., Buxton, J. N., Hartley, D. F., Nixon, E. & Strachey, C. (1963), 'The Main Features of CPL', *The Computer Journal* **6**(2), 134–143.

Bennett, J. P. (1996), *Introduction to Compiling Techniques: A First Course Using ANSI C, LEX and YACC*, The McGraw-Hill International Series in Software Engineering, 2nd edn, McGraw-Hill.

Chomsky, N. (1956), 'Three Models for the Description of Language', *IRE Transactions on Information Theory* **2**, 113–124.

Chomsky, N. (1959), 'On Certain Formal Properties of Grammars', *Information and Control* **2**, 137–167.

Curry, J. F. (1969), *The BCPL Reference Manual*, Xerox PARC.

Cytron, R., Ferrante, J., Rosenn, B. K., Wegman, M. N. & Zadeck, F. K. (1991), 'Efficiently Computing Static Single Assignment Form and the Control Dependence Graph', *ACM Transactions on Programming Languages and Systems (TOPLAS)* **13**(4), 451–490.

Donnelly, C. & Stallman, R. (2002), *Bison: The YACC-compatible Parser Generator*, GNU/Free Software Foundation Inc.

Emery, G. (1986), *BCPL and C*, 1st edn, Blackwell Scientific Publications.

Feather, C. (2002), 'A Brief Description of BCPL', `http://www.lysator.liu.se/c/clive-on-bcpl.html` (10 November 2003).

Fischer, C. N. & LeBlanc, R. J. (1991), *Crafting a Compiler with C*, Programming/Compiler Design, 1st edn, Benjamin/Cummings Publishing.

FOLDOC (2004), 'Free On-Line Dictionary of Computing', `http://foldoc.doc.ic.ac.uk/foldoc/index.html` (10 November 2003).

Fraser, C. & Hanson, D. (2003), 'lcc, A Re-targetable Compiler for ANSI C', `http://www.cs.princeton.edu/software/lcc/` (20 December 2003).

Fraser, C. W. & Hanson, D. R. (1991*a*), 'A Code Generation Interface for ANSI C', *Software - Practice and Experience* **21**(9), 963–998.

Fraser, C. W. & Hanson, D. R. (1991*b*), 'A Re-targetable Compiler for ANSI C', *ACM SIGPLAN Notices* **26**(10), 29–43.

GCC (1999), 'GCC Development Mission Statement', `http://gcc.gnu.org/gccmission.html` (10 December 2003).

GCC (2003*a*), 'GCC Coding Standards', `http://gcc.gnu.org/codingconventions.html` (02 December 2003).

GCC (2003*b*), 'GCC Front Ends', `http://gcc.gnu.org/frontends.html` (20 October 2003).

GCC (2003*c*), 'SSA for Trees', `http://gcc.gnu.org/projects/tree-ssa` (20 October 2003).

GCC (2003*d*), 'Tree SSA Documentation', `http://people.redhat.com/dnovillo/tree-ssa-doc/html/index.html` (01 November 2003).

GCC (2004*a*), 'Contributing to GCC', `http://gcc.gnu.org/contribute.html` (14 January 2004).

GCC (2004*b*), 'GCC 3.5 Release Series - Changes, New Features and Fixes', `http://gcc.gnu.org/gcc-3.5/changes.html` (22 April 2004).

GCC (2004*c*), 'GCC Home Page - GNU Project - Free Software Foundation (FSF)', `http://gcc.gnu.org` (20 October 2003).

GNU (2003*a*), 'GNU Coding Standards', `http://www.gnu.org/prep/standards_toc.html` (02 December 2003).

GNU (2003*b*), 'GNU General Public License', `http://www.gnu.org/copyleft/gpl.html` (10 December 2003).

GNU (2003*c*), 'GNU Pascal', `http://www.gnu-pascal.de/gpc/index-orig.html` (02 April 2004).

GNU (2003*d*), 'Information For Maintainers of GNU Software', `http://www.gnu.org/prep/maintain.html` (10 December 2003).

GNU (2004*a*), 'GNU Fortran 95', `http://gcc.gnu.org/fortran/` (02 April 2004).

GNU (2004*b*), 'GNU Modula-2', `http://floppsie.comp.glam.ac.uk/Glamorgan/gaius/web/GNUModula2.html` (02 April 2004).

Grune, D., Bal, H. E., Jacobs, C. J. & Langendoen, K. G. (2000), *Modern Compiler Design*, Worldwide Series in Computer Science, 1st edn, John Wiley & Sons Ltd.

Hendren, L., Donawa, C., Emami, M., Gao, G. & Sridharan, B. (1992), Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations, *in* 'Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing', number 457 *in* 'Lecture Notes in Computer Science', Springer-Verlag, pp. 406–420.

IEEE (1998), *IEEE/ANSI 830-1998 Recommended Practice for Software Requirements Specifications*, The Institute of Electrical and Electronics Engineers.

IOCCC (2003), 'International Obfuscated C Code Contest', `http://www.ioccc.org/main.html` (10 December 2003).

ISO (1999), *ISO/IEC 9899:1999: Programming Languages – C*, International Organisation for Standardisation.

ISO (2003), *ISO/IEC 9945:2003: Portable Operating System Interface (POSIX)*, International Organisation for Standardisation.

Johnson, S. C. & Kernighan, B. W. (1973), *The Programming Language B*, Bell Laboratories.

Johnson, S. C. & Ritchie, D. M. (1978), 'UNIX Time-Sharing System: Portability of C Programs and the UNIX System', *AT&T Bell Laboratories Technical Journal* **57**(6), 2021–2048.

Josling, T. (2001), 'Using, Maintaining and Enhancing COBOL for the GNU Compiler Collection (GCC)', `http://cobolforgcc.sourceforge.net/cobol_toc.html` (02 April 2004).

Kernighan, B. W. & Ritchie, D. M. (1978), *The C Programming Language*, 1st edn, Prentice Hall.

Lattner, C. (2004), 'The LLVM Compiler Infrastructure', `http://llvm.cs.uiuc.edu/` (01 February 2004).

Lattner, C. & Adve, V. (2004*a*), 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.

Lattner, C. & Adve, V. (2004*b*), 'LLVM Language Reference Manual', `http://llvm.cs.uiuc.edu/docs/LangRef.html` (01 February 2004).

Lattner, C., Dhurjati, D. & Stanley, J. (2004), 'LLVM Programmer's Manual', `http://llvm.cs.uiuc.edu/docs/ProgrammersManual.html` (01 February 2004).

Levine, J. R., Mason, T. & Brown, D. (1995), *lex & yacc*, Unix Programming Tools, 2nd edn, O'Reilly.

Louden, K. C. (1997), *Compiler Construction: Principles and Practice*, 1st edn, PWS Publishing.

Loudon, K. (1999), *Mastering Algorithms with C*, Programming Series, 1st edn, O'Reilly.

Manning, M. (1999), 'BCPL', `http://www.cus.cam.ac.uk/~mrm1/` (10 November 2003).

Merill, J. (2002*a*), 'Language-independent functions-as-trees representation', `http://gcc.gnu.org/ml/2002-07/msg00890.html` (20 November 2003).

Merill, J. (2002*b*), 'Re: Language-independent functions-as-trees representation', `http://gcc.gnu.org/ml/gcc/2002-08/msg01397.html` (20 November 2003).

Merrill, J. (2003), GENERIC and GIMPLE: A New Tree Representation for Entire Functions, *in* 'Proceedings of the 2003 GCC Summit', pp. 171–180.

Middleton, M. D., Richards, M., Firth, R. & Willers, I. (1982), *A Proposed Definition of the Language BCPL*.

Mycroft, A. & Norman, A. C. (1986), *The Internal Structure of the Norcroft C Compiler*, Codemist Ltd.

Naur, P., Backus, J. W. & McCarthy, J. (1963), 'Revised Report on the Algorithmic Language ALGOL 60', *The Computer Journal* **5**(4), 349–357.

Novillo, D. (2003*a*), Tree SSA - A New Optimization Infrastructure for GCC, *in* 'Proceedings of the 2003 GCC Summit', pp. 181–194.

Novillo, D. (2003*b*), Tree SSA: A New High-Level Optimization Framework for the GNU Compiler Collection, Technical report, Red Hat Canada Ltd.

Page, D. (2003), BCPL Compiler. BSc (Hons) Computer Science dissertation, University of Bath.

Paxson, V. (1995), *Flex: A Fast Scanner Generator*, GNU/Free Software Foundation Inc.

Pizka, M. (1997), Design and Implementation of the GNU INSEL Compiler, Technical Report TUM-I9713, Technische Universität München.

Pressman, R. (2000), *Software Engineering: A Practitioner's Approach*, European Adaptation, 5th edn, McGraw-Hill Education.

Richards, M. (1967), *The BCPL Reference Manual*, Massachusetts Institute of Technology.

Richards, M. (1969), BCPL: A Tool for Compiler Writing and System Programming, *in* 'Proceedings of the AFIPS Spring Joint Computer Conference', Vol. 34, American Federation of Information Processing Societies, pp. 557–566.

Richards, M. (1971), 'The Portability of the BCPL Compiler', *Software - Practice and Experience* **1**(2), 135–146.

Richards, M. (1973), *The BCPL Programming Manual*, University of Cambridge.

Richards, M. (1975), INTCODE - An Interpretive Machine Code for BCPL, Technical report, University of Cambridge.

Richards, M. (2000), 'BCPL', `http://www.cl.cam.ac.uk/users/mr/BCPL.html` (20 October 2003).

Richards, M. (2003), *The BCPL Cintcode System Users Guide*, University of Cambridge.

Richards, M. & Whitby-Strevens, C. (1980), *BCPL - the language and its compiler*, 1st edn, Cambridge University Press.

Ritchie, D. M. (1993), 'The Development of the C Language', *ACM SIGPLAN Notices* **28**(3), 201–208. Preprints of the ACM History of Programming Languages Conference HOPLII.

Ritchie, D. M. & Thompson, K. (1974), 'The UNIX Time-Sharing System', *Communications of the ACM* **17**(7), 365–375.

Rosen, B. K., Wegman, M. N. & Zadeck, F. K. (1988), Global Value Numbers and Redundant Computations, *in* 'Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages', ACM Press, pp. 12–27.

Singer, J. (2002), Porting Legacy Interpretive Bytecode to the CLR, *in* 'Proceedings of the Inaugural Conference on the Principles and Practice of Programming', ACM International Conference Proceeding Series, National University of Ireland Publishing, pp. 163–168.

Sommerville, I. (2001), *Software Engineering*, International Computer Science Series, 6th edn, Addison-Wesley.

Sorensen, H. (2004), 'PL/1 for GCC', `http://pl1gcc.sourceforge.net/` (02 March 2004).

Stallman, R. M. (2004*a*), *GNU Compiler Collection Internals*, GNU/Free Software Foundation Inc.

Stallman, R. M. (2004*b*), *Using the GNU Compiler Collection*, GNU/Free Software Foundation Inc.

SUI (2001), 'The SUIF 2 Compiler System', `http://suif.stanford.edu/suif/suif2/` (21 December 2003).

Tremblay, J.-P. & Sorenson, P. G. (1985), *The Theory and Practice of Compiler Writing*, Computer Science Series, 1st edn, McGraw-Hill.

Wikipedia (2004), 'Wikipedia, the free encyclopedia', `http://en.wikipedia.org/` (10 November 2003).

Wilhelm, R. & Maurer, D. (1995), *Compiler Design*, International Computer Science Series, 1st edn, Addison-Wesley.

Willers, I. & Mellor, S. (1976), *The BCPL Mini-Manual*, Data Handling Division, CERN.

Appendix A

Compiler Phases

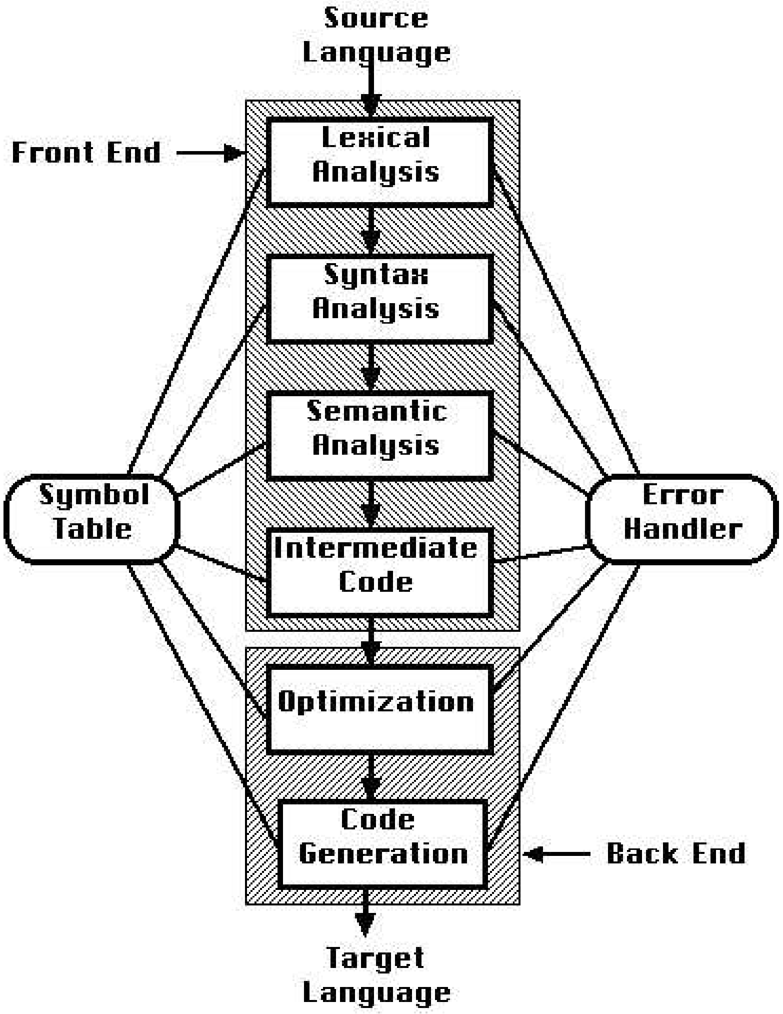

Figure A.1: Basic compiler phases

Appendix B

BCPL, B and C Code Examples

```
GET ‘‘LIBHDR’’

LET START () BE
$(
    WRITES (‘‘Hello, world!*N’’)
$)
```

Figure B.1: BCPL code example

```
main()
{
    putstr(‘‘Hello, world!’’); putchar(’*n’);
}
```

Figure B.2: B code example

```
#include <stdio.h>

int main(void)
{
    printf(‘‘Hello, world!\n’’);
    return 0;
}
```

Figure B.3: C code example

Appendix C

Backus-Naur Form for BCPL

This section presents the Backus-Naur Form (BNF) of the syntax of BCPL. As mentioned previously, this version of the BCPL BNF has been taken from (Richards & Whitby-Strevens 1980), but does not reflect the changes with respect to the commonly-used extensions, such as the optional semicolons or the infix byte operator. At the outermost level, a BCPL program is a sequence of declarations.

Identifiers, strings, numbers

```
<letter>         ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|
                     S|T|U|V|W|X|Y|Z
<octal digit>    ::= 0|1|2|3|4|5|6|7
<hex digit>      ::= 0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F
<digit>          ::= 0|1|2|3|4|5|6|7|8|9
<string const>   ::= "<255 or fewer characters>"
<char const>     ::= ’<one character>’
<octal number>   ::= #<octal digit>[<octal digit>]
<hex number>     ::= #X<hex digit>[<hex digit>]
<number>         ::= <octal number> | <hex number> |
                     <digit>[<digit>]
<identifier>     ::= <letter>[<letter> | <digit>| .]
```

Operators

```
<address op>     ::= @ | !
<mult op>        ::= * | / | REM
<add op>         ::= + | -
<rel op>         ::= = | = | <= | >= | < | >
<shift op>       ::= << | >>
<and op>         ::= &
<or op>          ::= |
<eqv op>         ::= EQV | NEQV
<not op>         ::=
```

Expressions

```
<element>       ::= <char const> | <string const> |
                    <number> | <identifier> | TRUE |
                    FALSE
<primary E>     ::= <primary E>(<expr list>) |
                    <primary E>( ) | (<expression>) |
                    <element>
<vector E>      ::= <vector E> ! <primary E> |
                    <primary E>
<address E>     ::= <address op><address E> | <vector E>
<mult E>        ::= <mult E><mult op><address E> |
                    <address E>
<add E>         ::= <add E><add op><mult E> |
                    <add op><mult E> | <mult E>
<rel E>         ::= <add E>[<rel op><add E>]
<shift E>       ::= <shift E><shift op><add E> | <rel E>
<not E>         ::= <not op><shift E> | <shift E>
<and E>         ::= <not E>[<and op><not E>]
<or E>          ::= <and E>[<or op><and E>]
<eqv E>         ::= <or E>[<eqv op><or E>]
<conditional E> ::= <eqv E> ->
                        <conditional E>,<conditional E> |
                    <eqv E>
<expression>    ::= <conditional E> |
                    TABLE <const expr>[,<const expr>] |
                    VALOF <command>
```

Constant expressions

```
<c element>     ::= <char const> | <number> | <identifier> |
                    TRUE | FALSE | (<constant expressions>)
<c mult E>      ::= <c mult E><mult op><c element> |
                    <c element>
<c add E>       ::= <c add E><add op><c mult E> |
                    <add op><c mult E> | <c mult E>
<c shift E>     ::= <c shift E><shift op><c add E> |
                    <c add E>
<c and E>       ::= <c and E><and op><c shift E> |
                    <c shift E>
<const expr>    ::= <const expr><or op><c and E> |
                    <c and E>
```

Lists of expressions and identifiers

```
<expr list>     ::= <expression>[,<expression>]
<name list>     ::= <name>[,<name>]
```

Declarations

```
<manifest item> ::= <identifier>=<const expr>
<manifest list> ::= <manifest item>[;<manifest item>]
<manifest decl> ::= MANIFEST$(<manifest list>$)
<static decl>   ::= STATIC$(<manifest list>$)
<global item>   ::= <identifier>:<const expr>
<global list>   ::= <global item>[;<global item>]
<global decl>   ::= GLOBAL$(<global list>$)
<simple defn>   ::= <name list>=<expr list>
<vector defn>   ::= <identifier>=VEC <const expr>
<function defn> ::= <identifier>(<name list>)=<expression> |
                    <identifier>( )=<expression>
<routine defn>  ::= <identifier>(<name list>) BE <command> |
                    <identifier> ( ) BE <command>
<definition>    ::= <simple defn> | <vector defn> |
                    <function defn> | <routine defn>
<simult decl>   ::= LET <definition>[ AND <definition>]
<declaration>   ::= <simult decl> | <manifest decl> |
                    <static decl> | <global decl>
```

Left hand side expressions

```
<lhse>          ::= <identifier> | <vector E> ! <primary E> |
                    !<primary E>
<lhs list>      ::= <lhse>[,<lhse>]
```

Unlabelled commands

```
<assignment>    ::= <left hand side list>:=<expr list>
<simple cmd>    ::= BREAK | LOOP | ENDCASE | RETURN | FINISH
<goto cmd>      ::= GOTO <expression>
<routine cmd>   ::= <primary E>(<expr list>) |
                    <primary E> ( )
<resultis cmd>  ::= RESULTIS <expression>
<switchon cmd>  ::= SWITCHON <expression> INTO
                        <command command>
<repeatable cmd>::= <repeatable cmd> REPEAT |
                    <repeatable cmd> REPEATUNTIL
                        <expression> |
                    <repeatable cmd> REPEATWHILE
                        <expression>
<until cmd>     ::= UNTIL <expression> DO <command>
<while cmd>     ::= WHILE <expression> DO <expression>
<for cmd>       ::= FOR <identifier> = <expression> TO
                        <expression> BY <const expr> DO
```

```
                                <command> |
                    FOR <identifier> = <expression> TO
                        <expression> DO <command>
<repetitive cmd>::= <repeated command> | <until cmd> |
                    <while cmd> | <for cmd>
<test cmd>      ::= TEST <expression> THEN <command>
                        ELSE <command>
<if cmd>        ::= IF <expression> THEN <command>
<unless cmd>    ::= UNLESS <expression> THEN <command>
<unlabelled cmd>::= <repeatable cmd> | <repetitive cmd> |
                    <test cmd> | <if cmd>
```

Labelled commands

```
<label prefix>  ::= <identifier> :
<case prefix>   ::= CASE <const expr> :
<default prefix>::= DEFAULT :
<prefix>        ::= <label prefix> | <case prefix> |
                    <default prefix>
<command>       ::= <unlabelled cmd> | <prefix> <command> |
                    <prefix>
```

Blocks and compound statements

```
<command list>  ::= <command>[;<command>]
<decl part>     ::= <declaration>[;<declaration>]
<block>         ::= $( <decl part> ; <command list> $)
<compound cmd>  ::= $( <command list> $)
<program>       ::= <decl part>
```

Appendix D

BCPL Operator Precedence

BCPL operators in order of precedence (highest to lowest - operators near the top bind most tightly):

Operators	Synonyms	Associativity
function call		left
`!` (dyadic) `%` `::`	`OF`	left
`@` `!` (monadic)		left
`*` `/` `REM`		left
`+` `-` (dyadic and monadic)		left
`=` `~=` `>` `<` `>=` `<=`	`EQ` `NE` `=` `GT` `LT` `GE` `LE`	left
`<<` `>>`	`LSHIFT` `RSHIFT`	left
`NOT`	`~`	left
`&`	`/\` `LOGAND`	left
`|`	`\/` `LOGOR`	left
`EQV` `NEQV`		left
$\rightarrow$		right
`TABLE`		right
`VALOF`		left
`SLCT`		left

Figure D.1: BCPL operator precedence, associativity and common synonyms

The floating point operators (beginning with '#') have the same precedence as their integer equivalents, while the precedence for the field selector (`SLCT`) is not described in (Richards & Whitby-Strevens 1980). All operators associate left-to-right except for $\rightarrow$ and `TABLE`.

Appendix E

Integration of GENERIC and GIMPLE into GCC

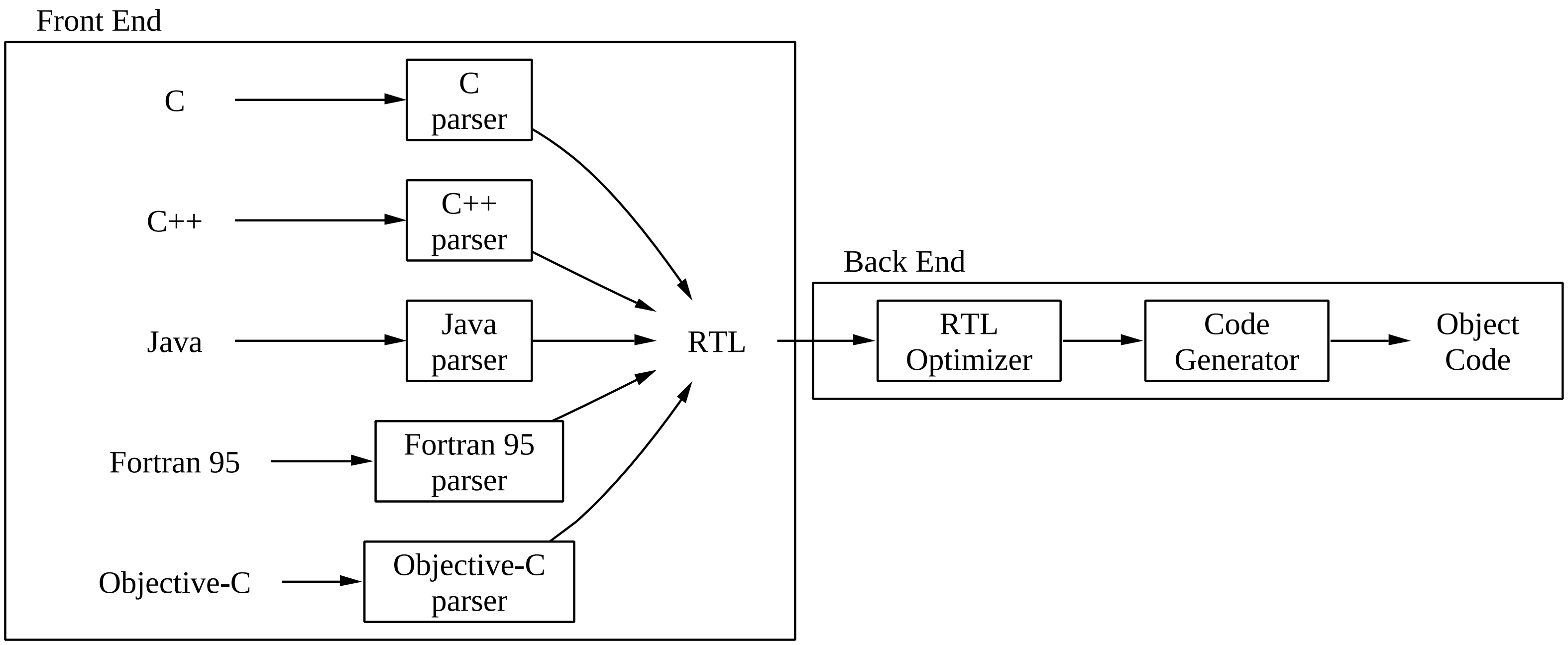

Figure E.1: Existing GCC framework (Novillo 2003*a*)

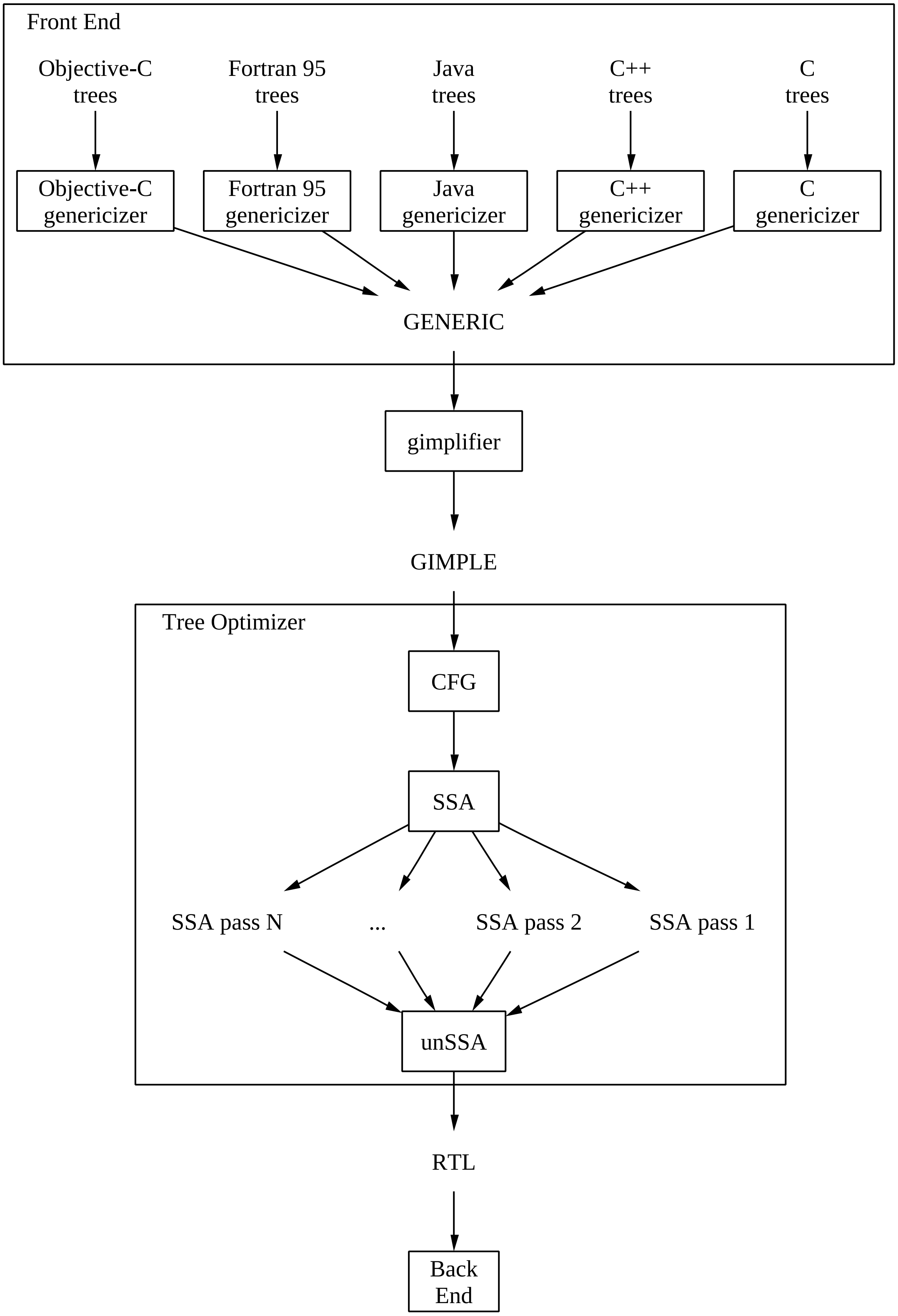

Figure E.2: Proposed integration of GENERIC and GIMPLE within the GCC framework (Novillo 2003*a*)

Appendix F

Test Results

This chapter displays a short selection of some of the test scripts used and results from running the prototype compiler. For further details, please see the attached source code CD in Appendix H.

```
[cs1tc@tcrick bcpl]$ ./bcpl
Usage: bcpl [OPTION...] file
Try ‘bcpl --help’ or ‘bcpl --usage’ for more information.
...
...
[cs1tc@tcrick bcpl]$ ./bcpl --help
Usage: bcpl [OPTION...] file
A BCPL compiler

  -o, --output=<file>        Place the output into <file>
  -v, -s, --verbose, --silent   Produce verbose output
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version

Mandatory or optional arguments to long options are also
mandatory or optional for any corresponding short options.
Report bugs to <tom@tomcrick.com>.
...
...
[cs1tc@tcrick bcpl]$ ./bcpl --usage
Usage: bcpl [-vs?V] [-o <file>] [--output=<file>] [--verbose]
            [--silent] [--help] [--usage] [--version] file
...
...
[cs1tc@tcrick bcpl]$ ./bcpl --version
bcpl (BCPL) version 0.0.1 20040513
```

```
Copyright (C) 2004 Tom Crick (tom@tomcrick.com)
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE.
...
...
[cs1tc@tcrick pre_trees]$ ./bcpl ../bcpltest/ackermann.b
../bcpltest/ackermann.b:144:20 parse error,
                        unexpected PERCENT
../bcpltest/ackermann.b:144:50 multi-line
                        string constants are not allowed

// ackermann.b
GET "LIBHDR"

LET A(M,N) = M=0 -> N+1, N=0 -> A(M-1,1), A(M-1, A(M,N-1))
AND START() BE
$(
        LET M, N = 0, 0
        $(
                WRITES("Type two arguments (zero to exit)*N")
                M := READN()
                N := READN()
                WRITEF(Ackermann(%N, %N) = %N*N", M, N, A(M,N))
        $)REPEATUNTIL M = 0
$)
...
...
[cs1tc@tcrick pre_trees]$ ./bcpl ../bcpltest/hanoi.b
../bcpltest/hanoi.b:26:7 parse error,
                        unexpected IDENTIFIER, expecting PLING

// hanoi.b
GLOBAL $(
  START:1;
  WRITEF:76
$)

LET MOVEIT(F, T) BE $(
  WRITEF("move %N --> %N*N", F, T)
$)

LET HANOI(N, T, F, U) BE $(
  IF N=0 RETURN;
```

```
  HANOI(N-1, U, F, T);
  MOVEIT(F, T);
  HANOI(N-1, T, U, F)
$)

LET START () BE $(1
  HANOI(1, 3, 1, 2)
FINISH $)1
...
...
[cs1tc@tcrick pre_trees]$ ./bcpl ../bcpltest/fastackermann.b

// fastackermann.b
GET "LIBHDR"

MANIFEST $( maxn = 100 $)

LET A(T,M,N) = VALOF
$(
        LET x = M < 4 & N < maxn -> T!(M*maxn + N), 0
        IF x = 0 THEN
        $(
                TEST M = 0 THEN x := N+1
                ELSE x := N = 0 -> A(T, M-1, 1),
                                A(T, M-1, A(T, M, N-1))
                IF M<4 & N<maxn THEN T!(M*maxn + N) := x
        $)
        RESULTIS x
$)

AND START() BE
$(
        LET V = VEC 4*maxn
        LET M, N = 0, 0
        $(
                FOR j = 0 TO 4*maxn DO V!j := 0 //clear array
                WRITES("Type two arguments (zero to exit)*N")
                M := READN()
                N := READN()
                WRITEF("Ackermann(%N, %N) = %N*N",
                                        M, N, A(V,M,N))
        $) REPEATUNTIL M=0
$)
```

Appendix G

Program Versions

The program versions of the tools used in this project are listed below.

Red Hat Linux 9 (kernel 2.4.20-31.9)
GNU Flex 2.5.4
GNU Bison 1.35
GNU Make 3.79.1
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
GNU DDD 3.3.1
GNU Emacs 21.2.1

```
GCC v3.5-tree-ssa
Reading specs from:
/usr/local/gccssa/lib/gcc/i686-pc-linux-gnu/
3.5-tree-ssa/specs
Configured with:
../gcc-tree-ssa-20040407-src/configure
--enable-shared
--enable-threads=posix
--enable-checking
--with-system-zlib
--enable-__cxa_atexit
--program-suffix=ssa
--prefix=/usr/local/gccssa
Thread model: posix
gcc version 3.5-tree-ssa 20040506 (merged 20040430)
```

For typesetting and referencing this dissertation document:
LaTeX (Web2C 7.3.1) 3.14159 (kpathsea version 3.3.1)
BibTeX (Web2C 7.3.1) 0.99c (kpathsea version 3.3.1)
Kile 1.6.1 (LaTeXfront end for KDE)

Appendix H

Source Code

As previously discussed, the code has been forked into two branches: one containing the basic scanner, parser and symbol table; the other containing the GCC development implementation, including the basic BCPL system library. The code given in the `gcc/gcc/` directory contains the GCC 3.5-tree-ssa 20040506 (merged 20040430) source. See `http://gcc.gnu.org/onlinedocs/gcc-3.4.0/gccint/Subdirectories.html` for details of the structure of the source directory. To run the code, it would be necessary to configure and build a snapshot of GCC 3.5.0, available for download from (GCC 2004*c*).

Directories:

```
docs/
pre_trees/
gcc/gcc/bcpl/
```