A C Compiler and Virtual Machine Implemented in Haskell

Submitted May 2012, in partial fulfilment of
the conditions of the award of the degree BSc (Hons) Computer Science

Thomas Busby
4099919 – txb09u
School of Computer Science and Information Technology
University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature _____

Date _____/_____/_____

Abstract

This report describes my implementation of a compiler and its associated virtual machine. The compiler takes a subset of the C programming language as its input, and returns machine code which can be run on the virtual machine. The usefulness of compilers, their various features, and the theory that makes them possible are discussed in detail. I also provide a complete functional and technical specification of the source and target languages which are implemented, along with explanations for how the compiler is actually written and commentary on the associated challenges. The compiler source code is provided as a "literate Haskell" file with some annotations and descriptions of various functions.

1. Introduction

"*Simply stated, a compiler is a program than can read a program in one language – the **source** language – and translate it into an equivalent program in another language – the **target** language"*[1] (Aho, Lam, Sethi, Ullman 2007)

For the most part, compilers are used to translate high level, human-readable source code into another language (often one that a particular machine can execute directly). There are other types of compilers, but this sort is probably the most common. They are useful for two main reasons; the first being that they allow the programmer to think in a more conceptual way about the program being written, rather than the fine details of its implementation for a given machine. The second reason is that of portability: Often a program can be written once and compiled for many different machines and architectures with minimal (or ideally) no changes to the source code by using a compiler that targets that architecture.

A virtual machine is a self-contained software implementation of a computer. It is intended to behave in like a hardware implementation of a computer. It has an internal state which can be modified by executing machine instructions, any external input it receives must be passed in via the machine hosting the virtual machine.

The source language I have chosen is a modified subset of the C programming language (which I will refer to as C-flat). The version of C i based it on is described in "The C Programming Language" by B. Kernighan and D. Ritchie.[2]

The language which I have chosen to implement my compiler and virtual machine in is the pure functional language Haskell.

My target language is largely inspired by the machine language for the "Triangle Abstract Machine". My version was inspired by a Haskell implementation by H. Nillson, itself interpreted from a Java version described in the book "Programming Language Processors in Java: Compilers and Interpreters" by D. Watt and D. Brown.[3][4] I will refer to my target language as "Rectangle."

1.1 Reasons for Choosing Each Language

The reason I have chosen a subset of C as my Source Language is mainly due to how influential that language has been in shaping most of the modern programming languages we use today. A very high percentage of languages owe a lot of their syntax and general style to C, for example PHP, JavaScript, Java, C++, Perl and many others. In fact, many of these languages are categorised as "C-style languages" and many others are influenced by it in other ways. Because of its popularity and longevity, there is a very large amount of detailed documentation about the language and how it should be implemented.

Choosing Haskell as my implementation language was, in many ways, a matter of preference. As a functional language, it is quite different from my Source and Target Languages. Due to this, it has its own interesting set of pros and cons. Given that a compiler itself can be viewed as a function (in that it takes input and maps to an output), Haskell appears to be a good match for the problem.

Its main difficulty however is that it lacks a "state." By this I mean that I cannot set a value and come back to it later, like I can by using variables in an imperative language. Because many of the algorithms involved at the various stages of compilation require a state (are "stateful") then this must be overcome using a construction called a "Monad" which I will explain later.

My target language, as I have mentioned before, is based on the instruction set for Triangle Abstract Machine. It is a stack machine, which means that memory is organised as a stack data structure, where values are pushed onto (stored), and pop from (destroyed), the stack via the use of various machine code instructions. The reason I have chosen to target a stack machine over a register machine (which is a more common real-world, hardware machine) is due to the difficulty of the analysis required for register allocation. Targeting a register machine would have left me less time to focus on other aspects of the compiler, such as the type checker.

1.2 Aims of the Project

The main goal of this project is to be able to take a piece of C-flat source code, compile it into machine language, and then run that machine code on the virtual machine. The virtual machine should run the generated code without any errors introduced at compile-time. I.e. a piece of source code which is correct in terms of the language rules, and does not contain any logical errors should compile to machine code which runs without error, and produces the expected result.

In order to guarantee this, it is important that the different phases of compilation are performed correctly. The code itself should be successfully parsed into a tree structure which accurately represents the structure of the source code, any syntax errors should be caught at this phase and terminate compilation with an error. Other than logical errors, all other errors should be caught at the type checking and contextual analysis phase. Again, errors should cause the compiler to terminate execution and give an error summary to the user. Once it's been established that the source code represents a valid program, the code generator should correctly convert it to the target language. These phases will be explained in more detail in later chapters.

It is also important to remember that the correct execution of the generated code depends on correct implementation of the virtual machine. Code which is correct, but run by an incorrect implementation of the virtual machine cannot be guaranteed to provide the expected result. As such the C-flat compiler and Rectangle Virtual Machine (RVM) can be viewed as complimentary halves of a complete programming environment.

Since the overall aim of this project is to implement established compiler theory in Haskell. As such one of the challenges of the project will be to practically apply some of the theory surrounding programming in a functional language. Tasks such as the construction of suitable monads are viewed as part of the aims of the project, in the same way as successfully implementing a code generator and type checker (both explained later) are also aims of the project.

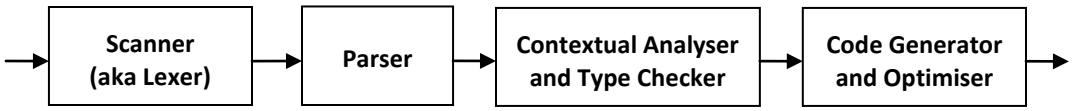
2. Motivation

As mentioned previously compilers allow problems to be solved in more abstract non-implementation-specific ways then compiled to the relevant architecture. However, compilers have many other potential uses. All mathematical problems can be phrased in terms of whether a particular object is in the language of correct solutions to that problem. Because of this, the potential for being able to translate objects expressed in one form into an equivalent form within a language of equal or greater expressive power makes compilers exceptionally powerful constructions.

However, in reality they are more commonly used for less abstract and more obvious purposes. They can be effectively used to translate documents from one kind to another. For example, this dissertation was written using MS Word 2007, but I will be submitting a PDF copy electronically. When I choose to export it to PDF, it will take the document in DOCX format, parse it into a tree structure and then compile that structure into an equivalent form that expresses the same content and layout as a PDF.

Because the same theoretical groundwork can be applied to problems of many different kinds; compilers, and languages, and approaches to implementing them are worth exploring for their own sake. This is because the insights are often useful in many other domains.

3. Overview of Research into Compiler Construction



Above is a simple diagram summarising the components of a compiler. The first stage, known as "Lexical Analysis," attempts to group the character stream into the component tokens of the language. Some examples of typical tokens would be a plus operator, a variable name, or a semi-colon. This stage is performed by the "Scanner" which is sometimes also known as a "Lexer."

Once we have a list of the actual "chunks" of syntax that make up the language, we can start trying to organise these into some kind of structure. This is what that "Parser" does. The Parser makes use of a description of the language known as a Context Free Grammar, which describes how to construct syntactically correct programs. From this model, the Parser attempts to work backwards to build a tree that represents the true structure of the program.

If the Parser succeeds then we know that we have a program which is syntactically valid, but we still don't know if it actually "makes sense." For example, a program which tried to read a variable that doesn't exist, or tried to add an integer to a Boolean, would still be considered syntactically correct. We must therefore take care to prove the absence of those kinds of errors rather than passing over to the code generator in good faith. This is the purpose of Contextual Analysis and Type Checking, which are generally performed concurrently.

Code Generation is the final step to be taken. The compiler has satisfied itself so far that the source code you have provided does indeed contain a valid program. Variables are called in scope, all expressions evaluate to their expected types, and functions are called with the correct number and types of arguments. At this stage, the code that is output will make sense as a program equivalent to the one defined in the source (even if it may contain code that will cause run-time errors).

Code generation can be quite complex as it needs to know what the address of a given variable (and due to scoping and variable shadowing is not trivial) and other contextual information to generate correct code. Having a limited number of registers on the target machine increases the complexity further still.

The whole process at its most abstract involves taking a flat stream of characters, extracting a tree structure containing the essence of its meaning, then to flatten that tree back out into a stream of characters with an equivalent meaning for a different language.

3.1 Formal Languages

Integral to the theory which underpins compiler construction is the concept of "formal" languages. Here are some basic definitions:

- *"A **language** is a (possibly infinite) set of words.*
- *A **word** is a finite sequence (or string) of symbols*
- ***ε** denotes the empty word (a sequence of 0 symbols)*
- *An **alphabet** Σ is a finite set [of symbols]."*[5] (H. Nillson, accessed May 2012)

Each word within the language must be made up of symbols from that language's alphabet. The empty word cannot be a symbol within the alphabet, because it represents the absence of symbols.

For any language L and its alphabet Σ, $L \subseteq \Sigma^*$ holds true. That is, the words that *can* be in the language are made up of a string of unbounded (but finite) length using symbols from the alphabet. The set of words that make a language are therefore a potentially infinite subset of these possibilities.

There are a number ways to define languages, each having a particular expressive power. The Lexer and the Parser together seek to identify whether the character stream passed in is a **word** within the source **language** (as defined in a purely syntactic way, disregarding context-sensitive constraints such as type rules and variable scope).

3.2 Lexical Analysis

"*As the first phase of a compiler, the main task of the lexical analyzer is to read to input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program"* [6] (Aho, Lam, Sethi, Ullman 2007)

A **lexeme** is essentially a grouping of characters that constitute a single **token**. These may be simple, for example an open brace '{', or they may be more complex. A more complex example would be that of a variable name. A common way that this token is defined is to say that it can begin with either an underscore or a letter, (but not a numeral or other symbol) followed by any number of underscores and alphanumeric characters.

A **lexeme** is a **word** within the **language** of a single **token**. In the case of variable names as we defined them, this language is infinite. In order to recognise it, we need to be able to express that rule in a more formal way, and have a systematic method of recognising it.

The way in which we express the rule, is as a regular expression. **Regular Expressions** are a way of defining a type of formal language. They have limited expressive power compared with other types of language, but they are usually more than powerful enough to define the language of a particular token:

```
( _ | [a-z] | [A-Z])( _ | [a-z] | [A-Z] | [0-9] )*
```

Brackets with vertical bars indicate choice, for example, (a | b) would indicate that this is a one-character pattern that either consists of the character 'a' or the character 'b'. The square brackets, with two values separated by a dash, are shorthand for "any one character in this range." We could remove this construct and not lose any descriptive power, but writing the entire alphabet of lowercase and uppercase letters plus numerals with brackets and vertical bars would not be convenient.

The * simply means "zero or more of the previous character." In this case the star is applied to the whole choice expression, which is valid as it returns one character. There are other constructs which represent things such as "one or more" and "between n and m occurrences of," but I won't delve too deeply into them here.

Regular Expressions are one way to specify the class of languages known as **Regular Languages**. An equivalent (and directly executable way) is a Finite Automaton. These are often used in Lexers with an implicit rule that they match the longest string possible, to answer the question $w \in L($*a given lexeme*$)$?

By making use of Haskell's pattern matching I was able to construct a scanner which functions as well as one based on Finite Automata, but is considerably more human-readable, as there are only a few tokens which have variable spelling.

3.3 Syntactic Analysis (Parsing)

"[T]he parser obtains a string of tokens from the lexical analyser ... and verifies that the string of token names can be generated by the grammar for the source language. ... [F]or well formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing."[7] (Aho, Lam, Sethi, Ullman 2007)

Because Regular Languages and the methods used to recognise them have no way to "count" the number of occurrences a particular symbol that have been read so far, they cannot recognise languages where part of the definition involves matching up brackets, or any kind of tree-like structure.

This involves a new class of language called **Context-Free Languages** which require a more powerful notation to define them. This is what's known as a **Context-Free Grammar** (**CFG)** and they are one of the most important tools used in defining programming language syntax. Because the set of Regular Languages is a subset of the set of all Context-Free Languages, there is also a CFG equivalent of all regular expressions.

You may wonder why we don't perform the Lexical Analysis as part of Parsing. It *is* possible to have one single CFG that is capable of defining the whole syntax in one set of rules, but such a grammar is almost always incredibly large and unwieldy. It is far easier to work on a set of tokens than the raw character stream.

A CFG is made up of a set of terminal symbols (the alphabet of the language), a set of non-terminal symbols (temporary placeholders that represent particular kinds of structure), a set of production rules, and a single non-terminal to act as a start point.

A production rule maps a single non-terminal to a string of symbols (which may include non-terminal symbols). When there are no more non-terminal symbols in your string left to apply production rules too, then you have reached a valid word in the language that the CFG defines. As with Regular Expressions, the vertical bar represents a choice.

Below is an example of a simple CFG defining arithmetic expressions:

```
Terminal symbols: (0-9, +, -,*, /)

Non-terminal symbols: (

Start symbol: A

A  ::= A + M | A – M | M

M  ::= M * V | M / V | V

V  ::= (A) | N

N  ::= N N' | N'

N' ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

So, for example, if we wanted to construct the expression `(1 + 2) * 31`, we would begin at `A`, which we could then derive `M`, from `M` we get `A * V`. Applying the production for `M` we get `V * V`. From this we can apply the production for `V` to the first `V` to get `(A) * V`. We can then move through: `(A + M) * V, (M + M) * V, (V + M) * V, (N + M) * V, (N' + N) * V, (1 + N) * V,` ...finally arriving at `(1 + 2) * N, (1 + 2) * NN', (1 + 2) * N'N', (1 + 2) * 3N', (1 + 2) * 31.`

This type of derivation is known as a left-most derivation, because we always chose the left-most non-terminal in the string in order to apply a production rule. It is also worth noting that operator precedence and associativity are explicitly encoded into the grammar.

If I were to try to parse the expression `3+3*4*5` using this grammar, the only way that string can be constructed is with the following implicit bracketing: `3+((3*4)*5)`. If more than one tree existed that constructed a given string, then the grammar would be ambiguous, and probably wouldn't be able to be parsed effectively.

What the parser does is start from a string of tokens, and attempt to work backwards to the start symbol, un-applying production rules as it goes. There are a number of algorithms that can do this, and they generally place their own extra restrictions upon the grammar. For example, even a completely unambiguous grammar cannot be parsed via a recursive descent parser if it contains left-recursion (i.e. a production of the type: `P ::= P P'`). However there are often ways to transform most real-world language grammars into an equivalent form that can be parsed by a given algorithm.

Usually the output of the parsing phase is a tree structure which is similar to the CFG but simplified. Because operator precedence, associativity, and bracketing are enforced by the permissible structure of the CFG, we can build the same tree without all the extra nodes and restrictions which are present in the CFG only to enforce that. In the AST, precedence, bracketing and associativity are represented implicitly in its structure. This simplified essence of the structure is known as an **Abstract Syntax Tree** (**AST**).

To use the example, supplied above, it's associated AST may combine the top three productions, and allow left or right associativity:

```
A ::= A + A | A - A | A * A | A / A | N
```

This is not something which my project focuses heavily on. I chose to use a parser generator called Happy which uses a file containing a representation of the language CFG, and fragment of code for each production that tells it how to build each node of the AST. It then uses that to output a Haskell source file which contains the constructed compiler.

3.4 Type Checking and Contextual Analysis

"Programming language designers introduce type systems so that they can specify program behaviour at a more precise level than is possible in a context-free grammar. The type system creates a second vocabulary for describing both the form and behaviour of valid programs. ... The type system should ensure that programs are well behaved – that is, the compiler and run-time system can identify all ill-formed programs before they execute an operation that causes a run-time error. In truth, the type system cannot catch ***all*** *ill-formed programs [though.]"*[8] (K. Cooper, L. Torczon 2012)

There are certain types of behaviour which are often not caught by the type system, such as division by zero, or trying to dereference an invalid pointer. The type system should generally seek to prevent the programmer from executing statements that don't have an obviously defined meaning. For example, the expression `3 + true` is clearly nonsensical, because a Boolean is not considered to be a numeric value, so the expression has no definable value that can be returned.

However, given that hardware (and therefore most virtual) machines create a numeric representation for all values, often returning 1 (or a full binary word of 1s) and accepting any non-zero as an encoding for true, and 0 for false. For a programming language without a type system the only thing we can say is that the value returned is probably going to be some value other than 3, possibly 4, but depending on the binary representation of true, it could be many other values.

These kinds of errors can be more subtle and difficult to track down if, for example, the expression returning a Boolean is a function and the programmer has accidentally used it where an integer-returning function was intended.

In order to define what kinds of values can be used together, we must first define a set of types that values can be, for example:

```
T ::= Bool | Int
```

Types can be recursive tree structures. This allows us to represent values such as a pointer to a variable of a particular type, or a function that maps arguments from a list of types to a return type. This is why the notation here looks very similar to that of CFGs and ASTs.

The example above defines a basic type system with two types: a Boolean type, and an Integer type. Now that we have the types, we must specify the types of expressions that can exist in the language:

```
e ::= b                          (Boolean Literal)
  |   i                          (Integer Literal)
  |   e > e                      (Greater than expression)
  |   e || e                     (Logical Or expression)
  |   e + e                      (Addition expression)
```

We could lump all operators together as a particular type of arity-two function and create a functional type, but for the sake of keeping this explanation brief, I will treat them all as separate functions.

Using this set of expressions, we can now define the types of each expression, using a series of rules. The first and most trivial rules are there to define that the literals correspond to their associated types:

```
b : Bool                         T-BOOLLIT
e : Int                          T-INTLIT
```

Now that we have the base cases properly typed, those expressions which are recursive need to be assigned types. These are slightly more complicated in that part of the condition of them being properly typed is that the sub-expressions contained within them are also properly typed (allowing us to prove correctness by induction). It is also necessary that those sub-expressions, if they are properly typed, return the types of values that the expression expects. These conditions are specified above the main type rule:

```
t1 : Int  t2 : Int                    T-GREATERTHAN
  t1 > t2 : Bool

t1 : Bool  t2 : Bool                  T-LOGICALOR
  t1 || t2 : Bool

t1 : Int  t2 : Int                    T-ADDITION
   t1 + t2 : Int
```

To use T-GREATERTHAN as an example, how these rules should be interpreted are "assuming t1 and t2 are properly typed, and return values with an Int type, then the applying the greater than operator to the values is considered a properly typed expression of type Bool."

What is currently missing however, is a method for us to identify the type of a variable. Due to the fact that a variable of a given name may refer to different *actual* variables of different types in different contexts within the program, we need to be able to provide a context (the current environment) in which to look up the variable's type.

This environment is usually represented by an upper case gamma Γ and we use a turnstyle symbol ⊢ to say that in the context of Γ a particular variable has a given type. This context must now be included in all type rules, so to use T-GREATERTHAN as an example again:

```
Γ ⊢ t1 : Int  Γ ⊢ t2 : Int            T-GREATERTHAN
    Γ ⊢ t1 > t2 : Bool
```

We can now specify that all these expressions have a valid type within a given context. This is important because the same expression may be properly typed in one part of the program, but not in another. This may be because the variable is not in scope or a different variable with that name and the wrong type to the one intended *is* in scope.

If we assume a new expression `x` (which represents a variable), and a new assignment operator expression `:=` (which assigns to a variable and returns the assigned value) have been added, we can now define the notion of variable look-up and assignment in terms of the type system:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T-VAR} \qquad \frac{\Gamma(x) = T \quad \Gamma \vdash t : T}{\Gamma \vdash x := t : T}\ \text{T-ASSIGN}$$

It is important to note that the assignment rule assumes that the variable has already been declared elsewhere with a specific type. I have not included this construct in the example for the sake of brevity, but, as it would be a statement rather than an expression, its type rule would not return a type.

The examples in the section are adapted from examples given in lecture slides.[9] (H. Nillson, accessed May 2012)

This phase of compilation will usually take the AST generated by the parser and annotate it with information discovered during checking. This Intermediate Representation (IR) is then passed on for Code Generation. In the case of my compiler however, it was not necessary to create an IR and the code generator works directly from the AST.

3.5 Code Generation

"The code generator takes as input an intermediate representation of the source program and maps it to the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task."[10] (Aho, Lam, Sethi, Ullman 2007)

The way this is accomplished for a stack machine is somewhat simpler than for a register machine. Register allocation is a difficult optimisation problem requiring complex analysis of variable lifetimes to determine which variables to assign to registers, which to store in memory and when to move a variable from one to the other.

The code generator must also be able to output code in a structured way which reflects the different types of variable, and must clearly separate the different functions from one another (unless of course the source language supports nested function definitions). The structure must facilitate the generation addresses for the variables which will correctly point to the values. This usually requires several types of memory, **stack** memory for addressing local variables relative to a local base, a **static** memory for globals whose addresses can be calculated at compile-time, and **heap** memory for variables whose location can't be calculated statically, but whose lifetime aren't tied to the lifetime of a particular function.

In order to be able to address local variables, we need to create a **stack frame** whenever a function is called. When this stack frame is created, the pointer to the local base is modified so that it now points to the base of the new stack frame. It is important that the previous value contained within that pointer is stored in memory so that it can be restored open returning from that function. This is known as the **dynamic link**. It is "dynamic" because it depends on the run-time behaviour of the program and must be computed on-the-fly.

The return address of the instruction which called the function must also be stored. If not, then any program that contains more than one call to that function would be unable to return from a function and continue execution from the instruction that follows the function call.

For a stack machine, the process of generating code can be expressed in terms of a set of base cases, and recursive templates. This process is somewhat analogous to constructing a string in a given language using a CFG. The process itself is very similar. The main difference that instead of having choices for how to apply productions, the tree that you are working on tells you which production to apply next after each stage, so unlike (the majority of) CFGs, the process is deterministic.

A few quick examples of this would be:

```
evaluate [ E1 * E2 ] =
       evaluate E1
       evaluate E2
       MULT

evaluate [ X ] =                                   (where X is a variable)
       LOAD addr(X)

execute [ if E then S1 else S2 ] =
       evaluate E
       JUMPIFZ l1
       execute S1
       JUMP l2
       LABEL l1
       execute S2
       LABEL l2
```

As you can see, the items in lowercase are recursive calls to generate code for the sub expressions contained within that node of the IR tree. You may also notice the `addr` function, which is used to indicate that the code generator should retrieve the calculated address for that variable from its own memory. It is also implicitly required that `l1` and `l2` within the template for the `if` statement be unique. Ensuring that these labels are "fresh" requires you to keep track of what labels have already been used.

The examples in the section are adapted from examples given in lecture slides.[11] (H. Nillson, accessed May 2012)

3.6 The State Monad

Many types of computation within this project require a lot of information about the current scope level, what variables and functions are visible, their types, whether they are initialised etc. So that we have this information, we could make sure that all this information is dragged around in the type signature of all the various functions that check expressions, declarations etc. For something like a list of variable declarations, each time one is checked, the environment would have changed after processing those new variables.

When you return from the function that changed those "state" variables however, the calling function's copy of environment will not retain those changes. Haskell functions cannot have any side effects beyond consuming arguments and returning values. This would mean that quite often, you would need to return a vast tuple of these "state" variables plus the "real" return value, then pass all those updated versions into the next function. This would be unwieldy, and would make even slightly non-trivial programs completely unreadable and error-prone to code.

What we need then, is a way to make Haskell "pretend" to have a state which is external to the functions, and hide away the "pipework" of passing variables along in large tuples. We accomplish this with something called a State monad. I won't get into the finer details of monads and how they function, but I will instead explain how this particular design pattern functions and makes itself exceptionally useful.

Firstly, the most basic form of the state monad has this form, with one variable in its "state."

```
St a = Int -> (a, Int)
```

That is, the form of this monad is a function which takes an Integer and returns a tuple containing a value of any given type, and an Int value (the state). Monads have two accompanying functions defined:

```
return :: a -> St a

return :: a -> (Int -> (a -> Int))          (equivalent to the above)

return a = \i -> (a, i)
```

```
(>>=) :: St a -> (a -> St b) -> St b
st >>= f = \i -> let (a, i') = st i in f i'
```

This may look complicated, but what we're essentially doing is taking the state's value, and performing the act of passing all those variables along behind the scenes. We've can't escape from having to pass variables along in this way, but we can abstract away the process. By defining extra functions which can modify and retrieve the right hand value, we can use these to simulate the existence of an external state:

```
inc :: Int -> St ()
inc n = \i -> ((), i+n)


get :: St Int
get = \i -> (i, i)
```

Using the "do" notation, (which is syntactic sugar for the use of multiple `>>=` operators) we can write a function of the following form:

```
addLengthToState :: [a] -> St (Int, Int)
addLengthToState as = do
      s <- get
      inc $ length as
      s' <- get
      return (s, s')
```

This function get get the current value of the state, increment that value by the number of items in the list passed to it, and then return a tuple containing both the old and the new values. Because the return type of this monad is still a function, remember to call the first function in your "stateful" functions with an extra value, which is the inital value of the state:

```
addLengthToState [1,2,3,4]    evaluates to  (\i -> (i, i + 4))
addLengthToState [1,2,3,4] 3  evaluates to  (3, 7)
```

The monads which are used in my actual implementation work in the exact same way as these except for the fact that due to the way Haskell classes work, there is some extra code to package and unpackage them into `data` values, rather than just renaming a function type using the `type` keyword. My state monads also have extra fields in the tuples and corresponding extra arguments in the lambda expressions to carry the different variables in my state.

The examples in the section are adapted from "Programming with Effects".[12] (G. Hutton, accessed May 2012)

4. Functional Specification

The final compiler should be a single executable file which can be called from the command line. It will take an input file as a command line argument. This input file will contain the code to be compiled. If compilation is successful, it will either output to a file default filename "a.out" or to a file location specified in another command line argument. There will also be the option to execute the outputted code immediately using the virtual machine.

4.1 Compiler Specification

The design philosophy of my source language is that it should essentially be a subset of ANSI C in the syntactic constructs that it supports, but an extension of ANSI C when it comes to types and enforcement of type-safety.

C-flat should support both local and global variables. Unlike the version of C that inspired it, it should have an explicit `bool` type. Conditional expressions within loops and branch structures should evaluate to this type. C-flat includes this constraint in order to provide some extra type safety that basic C does not provide. In C, you can use pass an expression of any type to a conditional statement, and true or false is encoded as whether the machine representation of that value is non-zero or zero respectively.

However, in keeping with the original design, I will consider the single quote notation for characters to be syntactic sugar for the `int` type. As such, the type checker will consider values of `int` and `char` to be of the same type, where the integer value is the corresponding ASCII code for the character.

C-flat should support user defined functions, and these should be able to be called recursively. C-flat will also support an explicit `void` type which means that the function does not return a value. When a `return` statement is called from within a function, the type system should be able to recognise whether or not a value is required or forbidden by the function declaration's return type. It should also make sure those types match up if a value is required.

When a function is called, the checker should also be able to check that the correct number of arguments is passed in, and that the types of those arguments correspond to those in that function's definition.

Variables should be considered scoped to the block which they are defined in, with globals being at the base, followed by function arguments, then locals. Function arguments are to be considered read-only in the body of that function. Variable shadowing should be supported but the checker should guard against variables being re-declared with the same name at the same scope level. The fact that arguments can be shadowed is the reason why function arguments are given a scope level between global variables and the local variables defined within the function body.

As for the generated code, the compiler should seek to minimise the usage of memory. It should keep track of the expected lifetime of values, arguments, and local variables pushed onto the stack, and should include a pop instruction to clear the values from memory at the end of their lifetime. It should also seek to do this in as few instructions as possible.

4.2 Virtual Machine Specification

The virtual machine should provide a somewhat minimal environment that allows for the execution of code. The instruction set should be composed of instructions which are very low-level.

Manipulation of stack frames and associated registers can be done implicitly via the use of call and return instructions, and memory locations should be represented as locations on a stack. Generally speaking however, the virtual machine should do as little as possible to "make life easy" for the compiler and code generator. The instruction set of the virtual machine should seek to be minimal rather than comprehensive.

The virtual machine should have a method of printing some output to the command line during execution, but it does not need to support user input.

5. Technical Specification

In this section I will formally specify both my source and target languages. This includes regular expression specification for the lexical tokens of the source language. The complete source language CFG expressed in Backus-Naur Form will also be provided along with the type rules and variable scoping rules. Finally the instruction set of the target language, along with the effect of each instruction, will be provided along the intended runtime structure of programs implemented by it.

5.1 Lexical Tokens

Below are tokens and their associated regular expressions that make up the C-flat lexical syntax. Those which have a data type after their name are tokens which have a variable spelling. That variable spelling is considered to be the "value" of the token and is stored along with it.

There are only four tokens with variable spellings. The first is T_Id which has the most complex regular expression:

```
T_Id String        ([a-z]|[A-Z]|_)([a-z]|[A-Z]|[0-9]|_)*
```

T_Id stores the names of variables and functions. It's regular expression is the same as the example in the section on lexical analysis. It consists of a letter or and underscore, followed by any number of alphanumeric characters and underscores.

The next token of variable spelling is T_IntLit, whose regular expression consists of one or more numeric characters.

```
T_IntLit Int                              [0-9]+
```

T_CharLit either consists of any single character (except backslash) enclosed in single quotes, or a backslash and a character enclosed in single quotes. The second is valid only if `\x` is a valid escape character:

```
T_CharLit Char                          '.' | '\x'
```

The final token which has variable spelling is T_BoolLit. It's spelling is simply one of the two values it can hold, true or false:

```
T_BoolLit Bool                        true | false
```

The rest of the tokens in the lexical syntax have a fixed spelling, and do not hold any extra information. Their names and spellings are listed below:

T_IntType	int	T_CharType	char
T_BoolType	bool	T_VoidType	void
T_If	if	T_Else	else
T_Do	do	T_While	while
T_For	for	T_Break	break
T_Continue	continue	T_Return	return
T_LParen	(	T_RParen	)
T_LBrace	{	T_RBrace	}
T_LSqBracket	[	T_RSqBracket	]
T_Comma	,	T_QMark	?
T_Colon	:	T_SemiColon	;
T_Not	!	T_Eq	==
T_NotEq	!=	T_GTE	>=
T_LTE	<=	T_GT	>
T_LT	<	T_And	&&
T_Or	||	T_Assign	=
T_Div	/	T_Star	*
T_Plus	+	T_Minus	-
T_Mod	%		

5.2 Language Grammar

Below is the complete CFG for the C-flat language:

```
TranslationUnit
        ::= ExternalDeclaration
        |   ExternalDeclaration TranslationUnit

ExternalDeclaration
        ::= FunctionDefinition
        |   Declaration

Declaration
        ::= TypeSpecifier InitDeclaratorList ;

DeclarationList
        ::= Declaration
        |   Declaration DeclarationList

InitDeclaratorList
        ::= InitDeclarator
        |   InitDeclarator  , InitDeclaratorList

InitDeclarator
        ::= id
        |   id = Expression

FunctionDefinition
        ::= Declarator CompoundStatement
        |   TypeSpecifier Declarator CompoundStatement

TypeSpecifier
        ::= void
        |   int
        |   char
        |   bool
```

Declarator

::= **id** '(' ')'

| **id** '(' *ParameterList* ')'

ParameterList

::= *ParameterDeclaration*

| *ParameterDeclaration* , *ParameterList*

ParameterDeclaration

::= *TypeSpecifier* **id**

CompoundStatement

::= { }

| { *StatementList* }

| { *DeclarationList* }

| { *DeclarationList StatementList* }

StatementList

::= *Statement*

| *Statement StatementList*

Statement

::= *CompoundStatement*

| *SelectionStatement*

| *IterationStatement*

| *ExpressionStatement*

| *JumpStatement*

JumpStatement

::= return ;

| return *Expression* ;

SelectionStatement

::= if (*Expression*) *Statement*

| if (*Expression*) *Statement* else *Statement*

IterationStatement

::= while (*Expression*) *Statement*

ExpressionStatement

::= ;

| *Expression* ;

Expression

::= *AssignmentExpression*

AssignmentExpression

::= *LogicalORExpression*

| **id** '=' *AssignmentExpression*

LogicalORExpression

::= *LogicalANDExpression*

| *LogicalORExpression* || *LogicalANDExpression*

LogicalANDExpression

::= *EqualityExpression*

| *LogicalANDExpression* && *EqualityExpression*

EqualityExpression

::= *RelationalExpression*

| *EqualityExpression* == *RelationalExpression*

| *EqualityExpression* != *RelationalExpression*

RelationalExpression

::= *AddativeExpression*

| *RelationalExpression* < *AddativeExpression*

| *RelationalExpression* > *AddativeExpression*

| *RelationalExpression* <= *AddativeExpression*

| *RelationalExpression* >= *AddativeExpression*

AddativeExpression

::= *MultiplicativeExpression*

| *AddativeExpression* + *MultiplicativeExpression*

| *AddativeExpression* – *MultiplicativeExpression*

MultiplicativeExpression

::= *PostfixExpression*

| *MultiplicativeExpression* * *PostfixExpression*

| *MultiplicativeExpression* / *PostfixExpression*

| *MultiplicativeExpression* % *PostfixExpression*

PostfixExpression

::= *PrimaryExpression*

| *PostfixExpression* ( )

| *PostfixExpression* (*ArgumentExpressionList*)

ArgumentExpressionList

::= *Expression*

| *Expression* , *ArgumentExpressionList*

PrimaryExpression

::= **id**

| **intlit**

| **charlit**

| **boollit**

| (*Expression*)

Note: items which are bold and underlined are tokens with a variable spelling, items which are italic are non-terminal symbols.

This CFG is a stripped down version of the one described in “The C Programming Language” by B. Kernighan and D. Ritchie[13]

5.3 Type Rules and Context-Sensitive Constraints

The types that C-Flat supports are listed below:

`T ::= IntChar`	The type of Integers and Characters
`| Bool`	The type of Booleans
`| Void`	The type of Non-returning functions
`| Comparable`	The type of Comparable types
`| Arrow Ts T`	The type of functions which map from a list of types to a type

The << operator is defined as "is a subtype of." Currently, all types are a subtype of Comparable.

The types of expressions within C-Flat:

`Γ ⊢ i : IntChar`	Where i is an integer literal	T-INTLIT
`Γ ⊢ c : IntChar`	Where c is an character literal	T-CHARLIT
`Γ ⊢ b : Bool`	Where b is an boolean literal	T-BOOLLIT

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{T-VAR}$$

$$\frac{\Gamma(x) = T \quad \Gamma \vdash t : T}{\Gamma \vdash x := t : T} \quad \text{T-ASSIGN}$$

$$\frac{\Gamma \vdash t << Comparable}{\Gamma \vdash t == t : Bool} \quad \text{T-EQ}$$

$$\frac{\Gamma \vdash t << Comparable}{\Gamma \vdash t != t : Bool} \quad \text{T-NOT-EQ}$$

$$\frac{\Gamma \vdash t1 : Bool \quad \Gamma \vdash t2 : Bool}{\Gamma \vdash t1 \,\&\&\, t2 : Bool} \quad \text{T-LOGICAL-AND}$$

$$\frac{\Gamma \vdash t1 : Bool \quad \Gamma \vdash t2 : Bool}{\Gamma \vdash t1 \,||\, t2 : Bool} \quad \text{T-LOGICAL-OR}$$

$$\frac{\Gamma \vdash t1 : IntChar \quad \Gamma \vdash t2 : IntChar}{\Gamma \vdash t1 < t2 : Bool} \quad \text{T-LESSTHAN}$$

$$\frac{\Gamma \vdash t1 : IntChar \quad \Gamma \vdash t2 : IntChar}{\Gamma \vdash t1 > t2 : Bool} \quad \text{T-GREATERTHAN}$$

$$\frac{\Gamma \vdash t1 : IntChar \quad \Gamma \vdash t2 : IntChar}{\Gamma \vdash t1 <= t2 : Bool} \quad \text{T-LT-OR-EQ}$$

$$\frac{\Gamma \vdash t1 : IntChar \quad \Gamma \vdash t2 : IntChar}{\Gamma \vdash t1 >= t2 : Bool} \text{T-GT-OR-EQ}$$

$$\frac{\Gamma \vdash t1 : IntChar \quad \Gamma \vdash t2 : IntChar}{\Gamma \vdash t1 + t2 : IntChar} \text{T-PLUS}$$

$$\frac{\Gamma \vdash t1 : IntChar \quad \Gamma \vdash t2 : IntChar}{\Gamma \vdash t1 - t2 : IntChar} \text{T-MINUS}$$

$$\frac{\Gamma \vdash t1 : IntChar \quad \Gamma \vdash t2 : IntChar}{\Gamma \vdash t1 * t2 : IntChar} \text{T-MULT}$$

$$\frac{\Gamma \vdash t1 : IntChar \quad \Gamma \vdash t2 : IntChar}{\Gamma \vdash t1 / t2 : IntChar} \text{T-DIV}$$

$$\frac{\Gamma \vdash t1 : IntChar \quad \Gamma \vdash t2 : IntChar}{\Gamma \vdash t1 \% t2 : IntChar} \text{T-MOD}$$

$$\frac{\Gamma \vdash p : Arrow\ Ts\ T \quad \Gamma \vdash es \sim Ts}{\Gamma \vdash p\ (\ es\ )\ :\ T} \text{T-CALL}$$

Note: the ~ operator is used to denote that the expressions and the list of types both line up (i.e. each expression has the same type as the type at the same index in its list).

I have not included the type rules for the statement here as they are fairly trivial. Branch statements and loops want a Bool type for the conditional expression, and return statements want to make sure their type matches up with their function's return type. It should also be assumed that the global and local declarations, along with function definitions modify the environment to include these values.

In terms of other context-sensitive constraints that are placed on the language, main must exist as a function that takes no arguments with a return type of Void.

A variable name may be reused as long as no other variables exist at the same scope level with the same name.

Functions are not permitted to be called from the initialisation expression of a global variable.

Variables must definitely have been initialised before they can be used.

5.4 Code Generator

Below are the templates used for converting the expressions and statements into Rectangle code.

```
evaluate [ X ] =                                    (where X is a variable)
      LOAD addr(X)
evaluate [ I ] =                                    (where I is an integer literal)
      LOADL I
evaluate [ C ] =                                    (where C is a character literal)
      LOADL ord(C)
evaluate [ true ] =
      LOADL 1
evaluate [ false ] =
      LOADL 0
evaluate [ X := E ] =                               (where X is a variable)
      evaluate E
      LOAD ST 0
      STORE addr(X)
evaluate [ E1 `op` E2 ] =                           (where `op` is a binary operator)
      evaluate E2
      evaluate E1
      instr(`op`)
evaluate [ F ( Es ) ] =                             (where F is a function)
      evaluate Es
      CALL name(F)

execute [ if (E) S1 else S2 ] =
      evaluate E
      JUMPIFZ l1
      execute S1
      JUMP l2
      LABEL l1
      execute S2
      LABEL l2
```

```
execute [ if (E) S ] =
        evaluate E
        JUMPIFZ l1
        execute S
        LABEL l1
execute [ while (E) S ] =
        JUMP l2
        LABEL l1
        execute S
        LABEL l2
        evaluate E
        JUMPIFNZ l1
execute [ E ; ]
        evaluate E
execute [ return ]
        RETURN args(F) 0                    (where F is the current function)
execute [ return E ]
        evaluate E
        RETURN args(F) typeSize(E)          (where F is the current function)
```

The run-time structure of the outputted code is as follows:

Outputted code should begin by pushing the global variables onto the stack. These variables can be accessed relative to the SB register at the same location throughout execution. Next there should be a CALL "main" instruction followed immediately by a HALT. This is how the program begins execution, and terminates once main has returned.

After this the declarations for all the other functions in the program should be included and the last of these should be the main function itself. When a function is called, the LB register will point to the dynamic link (LB's previous value). LB + 1 will contain the return address to resume execution from once the function returns (i.e. the instruction directly after the function call).

Local variables should be pushed to the stack before any other values and addressed from LB + 2 onwards. Any arguments passed into the function will be found between LB – n and LB - 1 where n is the number of arguments provided

6. Implementation Details

In this section I will discuss the actual details of how the compiler is implemented. I will go through the important functions in each section of the compiler and explain their workings in detail. This section will be divided up between the main control flow, lexical analysis, a short discussion on the parser, the type checker and the code generator.

6.1 (Main.lhs) The Overall Control Flow

This module is the one which is called when the program is actually run. It is the basic glue that holds the various different modules together. As such, this module is required to import functions from almost all the other modules within the project.

The first function that we will look into generates the list of default settings that the compiler runs with. The source code is a mandatory argument, so that is passed in later, but it begins as an output of "a.out", the AST is not printed, and the code is not executed after compilation:

```
defaults :: SourceCode -> (SourceCode, Output, PrintAST, Exec)

defaults s = (s, "a.out", False, NoExec)
```

With these default values defined we can now move on to looking at how the program executes from when it is first executed. Like most programming languages, Haskell requires that you implement a main function if you wish to compile it to an executable.

`main` has a type of IO () which it needs in order to stray from the pure functional environment that most Haskell functions exist within. This is because a pure mathematical function has no external state, so things cannot happen as a side-effect of function evaluation. A pure function takes input and produces output; that is all it can do. However, main needs to be able to read input from the command line, display messages to the user, and output to files.

`main` gets the command line arguments and, if none were passed in, displays the usage hint. If arguments were passed in, it takes the first as the name of the input file and the rest are passed on to `processArgs` to be read. This only happens if the file not found to exist though, otherwise a then an error is reported and the program terminates.

`processArgs` calls its auxiliary function with the set of default values.

```
processArgs :: String -> [String] -> IO ()

processArgs s ss = processArgs' (defaults s) ss
```

`processArgs'` runs through the arguments passed in matching them up with what it expects to receive. If an invalid or nonsensical argument is passed in, then an error is returned and the program terminates. It should be noted that when multiple conflicting arguments such as -Exec and -ExecTrace are passed in, then it is the last one that is actually applied. Once the arguments have been consumed, the final values for the compilation settings are passed to the compile function.

I will include the compile function in full, because it is one of the most important functions in the program, as it enforces the overall abstract structure of the compiler:

```
compile :: (SourceCode, Output, PrintAST, Exec) -> IO ()
compile (i, o, p, e) = do
     ast <- scanparse i
     if p then printAST ast else return ()
     errors <- check ast
     if length errors > 0 then do
       putStrLn "Compilation Failed:\n"
       printErrors errors
      else do
       code <- generateCode ast
       writeFile o $ showCode code
       case e of
              NoExec -> return ()
              ExecNoTrace -> runCode False code
              ExecWithTrace -> runCode True code
```

`compile` takes the list of settings produced by `processArgs` and actually performs the compilation. Firstly the scanner and parser modules are used in order to build an AST. If errors occur during this process then those modules will throw an exception and terminate compilation themselves.

If an AST is successfully produced, then it is possibly printed (depending on the supplied arguments) and passed on to the checker for contextual analysis and type checking. Unlike many contextual analysers, the one implemented here does not produce an intermediate representation. It simply returns a list of errors, where "passing" is encoded as a zero-length list.

As such, if any errors are reported, they are printed, and compilation terminates, otherwise the AST produced by the parser is passed on to the code generator, which finishes the process by generating the actual sourceCode output. This output is then written to the file specified in Output, and, depending on whether the user requested it with an argument, the generated code is then passed on for execution by the virtual machine.

6.2 (Scanner.lhs) Lexical Analysis Phase

Scanner is the first module of the compiler to actually work on the source code. It looks for the lexemes within the character stream that correspond to the tokens expected by the parser. Because there are only four tokens which do not have a fixed representation, we can write a fairly trivial scanner that does not require analysis by some of the more complex Finite Automata-based Lexical Analysis algorithms.

```
scan :: [Char] -> [Token]
```

As you would expect, the `scan` function takes in a list of characters (the entire source code of the program) and outputs a sequence of tokens that correspond to the lexemes within the source code.

The first things that the scanner checks for are whitespace and comments. White space is ignored in C-flat apart from where it is used to separate one token from another, so it is just dropped when it occurs and scanning resumes on the rest of the source code.

Single line comments are simple to remove, because you just drop all characters until you come across the end of that line. Multi-line comments require the use of the auxiliary function described below.

When the scanner detects the start of a multi-line comment, it is passed to dropComment which simply drops all the characters up to and including the comment close */ and returns the remainder of the string:

```
dropComment :: [Char] -> [Char]
dropComment [] = []
dropComment ('*':'/':cs) = cs
dropComment (c:cs) = dropComment cs
```

After this, t he main structural symbols and operators within the language are now scanned. It is very important that those tokens which contain the lexeme of another token as a prefix are matched higher up in the function definitions, as this gives them higher precedence and they will be tested first, meaning the prefix lexeme will only be matched if it is not a part of the larger token. The best examples of this are the `T_Eq` and `T_Assign` operators. If `T_Assign` was matched first, you would never be able to match T_Eq, because the scanner would match two `T_Assigns` instead. The same problem would occur with `T_GTE` and `T_LTE` also.

```
scan ('=':'=':cs) = T_Eq : scan cs
     ...
scan ('>':'=':cs) = T_GTE : scan cs
scan ('<':'=':cs) = T_LTE : scan cs
scan ('>':cs) = T_GT : scan cs
scan ('<':cs) = T_LT : scan cs
     ...
scan ('=':cs) = T_Assign : scan cs
```

Now that we have matched all of those symbols, the fact that none of the characters that make up the lexemes above are allowed to occur within any of the other keywords or variably-spelt lexemes is what allows us to scan them without worrying about the "maximal munch rule". This rule would be a potential problem because it could mean we were consuming characters that are a prefix of a different lexeme.

The rest of the scanner's definitions deal with matching valid character literals, before moving on to matching T_Id along with the rest of the keywords

Because all of the keywords in the language would also match the regular expression for T_Id, it is important that we rule them all out before we attempt to create a T_Id token from the lexeme.

6.3 (Parser.ly) Syntactic Analysis Phase (Parsing)

Because I used a parser generator to construct my parser rather than hand-coding it, very little of its actual implementation is my own. The Context-Free Grammar specified in section 5.2 of this report is exactly the same as the one specified within Parser.ly (which is the source file for the parser generator, rather than the source code for the parser itself). It uses a slightly different notation to Backus-Naur Form and it also specifies how to construct an AST representation as part of its production rule, but otherwise it's the same.

Below is the corresponding Happy source code for the JumpStatement non-terminal:

```
JumpStatement :: { Statement }

JumpStatement

     : return ';'

       { StmReturn { srExp = Nothing } }

     | return Expression ';'

       { StmReturn { srExp = Just $2 } }
```

Where-ever there are an outer set of curly braces, the code contained within is actual Haskell code, with the exception of the $ notation which I will explain shortly. As you would expect, the part at the top of the fragment that looks like a type definition *is* a type definition, just as you would expect. What it says is that the function in the outputted parser will return a result of type `Statement`. `Statement` is one of the types of node within the AST and represents instructions that are executed, rather than the other main type, which are expressions that are evaluated to return a value.

This may seem unintuitive given that the whole point of `return` is to *return* a value, but you should think of return as an instruction to the run-time environment to pop the current stack frame and restore the previous frame. It is the function call itself which returns the value as far as the higher level semantics of the language are concerned.

The $ notation is concerned with providing a way to access the parser's output when the non-terminals within the right-hand side of the production rule. In the example above, the second option contains three items, a terminal symbol `return`, an *Expression* non-terminal and then a semicolon terminal, we can use $2 to indicate that we want to keep the AST sub-tree that is constructed for that *Expression* and store it (inside a `Just`) within the `srExp` field of `StmReturn`. It is in this fashion that the AST is recursively constructed.

6.4 (Checker.lhs) Contextual Analysis Phase (inc. Type Checking)

This module is by a long way the most sophisticated module that I wrote within the compiler, both in terms of what it actually does, and how it is implemented. This is because it was the last module to be constructed, and many of the insights I'd gained during the code generation phase were put to use here.

It is precisely because the code generator was written first, that I felt there was no need to return an annotated Intermediate Representation. The IR tree usually contains information about the variable scopes and types. But, out of necessity, the code generator module had already been written to work this out for itself. The code generator also didn't need to worry about whether the types of expressions and functions were correct because it could *assume* that they were, knowing that the type checker wouldn't pass it an AST where this wasn't the case. The types were also irrelevant to the code generator because it converted all values into an Integer type, which is the only type that the target machine supports internally.

As mentioned in the technical specification, there are five things that this checker proves: correctly typed expressions, correctly typed function arguments, whether or not variables are in scope when they are referenced, and whether they have been initialised when they are referenced. Finally it also ensures that, during a function's execution, the arguments passed in are treated as constants. Any attempt to overwrite them should cause an error.

The `Type` data type defined within the module echoes the types defined in Section 5.3 of this report, except for the introduction of a new type referred to as `T_Unknown` which is the "type" of an invalid expression. Having a type which represents wrongly typed expressions allows the checker to continue on and collect up as many other errors as it can before terminating so that it can provide the most corrective information to the user.

This type is also useful in preventing one badly typed sub-expression right near the leaf of an AST from creating type errors all the way back up the tree for every expression which it is a part of. We can consider a tree node containing an expression value of type `T_Unknown` to *always* be a properly typed expression and only emit an error for the expression that first causes a type error in the first place. This means that the exact cause of the error is pinpointed and passed on to the user.

Before I get into the details of how exactly we go about implementing the type rules, I will quickly explain the values held within my state monad, and the functions which are used to manipulate them.

Held within the state throughout the checking process are three different variables: a list of error messages, a representation of the current variable, function and operator environment, and the name of the function currently being checked.

The part of the state which holds all of the accumulated error messages only has one associated function, which is `emitError`. This function is very basic in that it takes a value of type `ErrorMessage` and appends it to the list of error messages already held within the state. Because we never have to modify the error messages or retrieve them mid-calculation, we can just accumulate them in the state ready to pass them back to `compile` in Main.lhs to be displayed to the user. If it weren't for the fact that there are values in the state which we need need to retrieve and modify, a Writer monad would have been sufficient to implement it.

`Environment` is listed as a single value within the declaration of the TC monad, but in reality it's actually quite a complex structure. `Environment`'s type is actually a three-tuple, containing three separate lists of other types of tuple. So, to begin to break it down, the three lists also have their own type names. These are `VariableDef`, `BinaryOpDef` and `FunctionDef`. What these represent is fairly self-explanatory. The first of these is a tuple containing five values, the variable's type, the variable's name, its scope level, that variable's status (known initialised, known uninitialized, unknown) and finally, whether or not it is a function parameter.

It turned out later that the final value was unnecessary because, as it turns out, all (and only) function parameters end up assigned a scope level of 1. So, it would have been enough to just check the scope level to determine whether it was a parameter.

The list of available binary operations appears simpler, in that it only consists of a type and the operator itself, but operators have quite complex, recursively defined types.

Function definitions have the exact same representation as binary operators, other than being identified by a string instead of a member of the `BinOp` data type. This is because binary operators are just a special case of functions (i.e. they are all arity-two functions which for reasons of convention alone are written using infix notation).

I also provide an initial environment which contains the binary operators defined by the language specification, along with the functions which are a part of the C-flat standard library.

The environment manipulation functions which are provided for use by the checking functions are listed as follows: `getEnv`, `updateEnv`, `addVarToEnv`, `addFuncToEnv`, `varAlreadyExists`, `getVarType`, `funcAlreadyExists`, `getFuncType`, `isInitialised`, `isParameter` and `removeVarsAtScope`.

`getEnv` does precisely what you would expect, and returns the entire current environment from the state for use directly within the calling function.

`updateEnv` is the inverse of the function defined above, in that it overwrites the existing state with one passed in from the calling function.

`addVarToEnv` prepends takes a new variable definition and prepends it to the existing list of definitions within the environment.

`addFuncToEnv` prepends does the same thing as `addVarToEnv` but with a new function definition.

`varAlreadyExists` checks to see if a variable with the same name and scope level already exists.

`getVarType` takes a variable name returns `Just t` where `t` is the type of the highest scoped variable with that name, and returns `Nothing` if there is no match.

`funcAlreadyExists` and `getFuncType` are the equivalents which query the list of functions instead of the list of variables.

`isInitialised` takes a variable name and returns whether the highest scoped variable with that name is known to be initialised. This can be Yes, No, or Maybe because in the case of branches, some may initialise the variables and others not.

`initialise` takes a variable name and sets the highest scoped variable with that name's definition to be known as initialised.

`mergeEnvs` is used after two separate environments have been created as a result of checking the two different branches of a branch statement (such as an if statement) or while loop. The two environments are compared, and where they disagree on whether a variable is known to be initialised or not, the initialisation is set to `Maybe`, signifying that it is unknown.

`getBinOpType` is an equivalent function to `getVarType` and `getFuncType` but for binary operators.

`isParameter` takes a variable name, and returns whether the highest scoped variable in the environment with that name is a parameter.

`removeVarsAtScope` is used when exiting a block or function. It clears the environment of variable definitions which are no longer in scope.

The remaining item in the state variables is a stored name of the current function being checked. The two functions that get and set this value require no explanation.

Now, beginning with the `check` function which is called from `main`, this is what actually kicks off the whole process of contextual analysis. What this function essentially does call the monadic `checkAll` function on the list of AST declarations passed in from main.

When a function which uses the state monad finishes executing it returns the whole tuple containing the return value and the final values of the state. Because we want to pass the (hopefully empty) list of error messages back from main, we went to take the second item from that tuple. `checkAll` has a return type of `()` so it doesn't actually return any meaningful value anyway.

Even though check is the first to be called, and it is the one which sets the ball rolling, `checkAll` is the "real" top-level function of the type checker. `check` acts as a translator between the functions which operate in the context of the TC monad, and the functions in main, which run in IO (also a monad, hence why it can use the "do" notation). Within `checkAll`, the first thing it does is check the existence of a `main()` function in the list of declarations. If it cannot find one, or it finds one with the wrong type, it prints an error. Because main is the start point of any program that is being compiled, be want any error messages regarding its structure to be printed right at the top of the list.

Once this has been done, we move on to checking the various global variables and their associated assignment expressions.

After this, we can check all of the functions. It needs to be done in this order, otherwise there might be an attempt to read a global variable within a function, but the type checker hasn't processed that declaration yet.

Given that we begin by checking the variable definitions, it makes sense to move on to this function next. `checkVarDefs` is a function which takes a scope level and a list of definitions. The scope level is the one that is to be given to new variables when they are added to the environment. This makes the function suitable for processing both lists of global and local arguments. Because all the function definitions have been filtered out already, we don't have to pattern match for it, we can just run through the list, passing each variable definition out to `checkDecls` along with the scope level, it's data type and the actual declarations contained within the definition.

The reason that there is a list of declarations contained within a single definition is the fact that variables may be declares like so:

```
int var1 = 3, var2, anotherVar = 4 + 3;
```

All those declarations would be carried as a field within a single node `Definition` node in the AST. The data type `int` would be held in the other field of the `Definition`.

`checkDecls` is the function which inserts each new variable definition into the environment. It won't do this however until it has satisfied itself that no other variables with the same same exist at the same scope. If no variable exists at this scope, then a new one is created. If the declaration has an associated initialisation expression, then its type is checked to make sure that it matches the variable's own type. Whether or not the variable does have an initialisation list is also reflected in the "initialised" section of its environment definition.

The process for evaluating the function definitions is very similar. The `checkFunctions` function takes the list of function definitions as its only argument. It begins by passing the actual declarator for the function contained within it to `checkFuncDecl`, before sending the statement which makes up the body of the function to `checkStm`. Because of the structure of the CFG, this statement is guaranteed to be a compound statement, which means that even though a scope level of 1 is passed in, that scope will immediately be incremented to 2 at the beginning of processing the compound statement. This is why only function parameters end up with a scope level of 1. Because of this, it also needs to clear up the scope-1 variables from the environment after processing each function. This is the only place where variables are cleared from the environment other than at the end of checking a compound statement.

`checkFuncDecl` simply dismantles the function's `declarator` and adds an equivalent definition into the environment. The only type of error it can produce is if two functions are defined with the same name. It also calls `processParams` which does for function parameters what `checkVarDefs` does for other variables.

This leaves us with `checkStm` and `checkExp`, which are the real work-horses of the type checker. These cases within functions are the direct implementation of the type rules from the language specification.

Beginning with `checkStm`, its first definition checks compound statements. Compound statements are simply a list of local variable declarations, followed by a list of statements. We simply use the existing machinery to process the local variable definitions (but incrementing the current scope by one), before calling an auxiliary function (also incrementing the scope) which recursively consumes the list calling `checkStm` on each statement within the list.

If statements and while loops are somewhat interesting from the point of view of the type checker. It is when these occur that the type checker loses its determinism. If I define a block as follows, then I cannot be certain whether the variable will be initialised when I try to print it:

```
{
        char c;
        if (someBool)
        {
                c = ‘a’;
        }
        printc(c);
}
```

This is why we perform analysis whenever we have program branches which may not always be followed to check whether all paths lead to a variable being initialised. The rules of the language state that all variables must be initialised before they can be read. A variable being initialises in *some* cases before it is read is not good enough to prove safety.

The other two types of statement are expression statements, and return statements. Expression statements cannot themselves cause any errors. The sub-expressions within them may be badly typed. At the root expression in the statement's sub-tree however, that expression's value is, by definition, not being passed any further up the tree, so no errors can occur. It is sufficient to just pass the expression over to `checkExp` along with the current scope level.

The return statement is fairly straightforward to understand also. It simply checks what is being returned with the type of value it expects to be returned. If the user is trying to return something from a void type function, it outputs an error, if the user is trying to return nothing from a non-void function, it outputs its an error, and if the user is trying to return a value of the wrong type, it emits an error.

`checkExp` is probably the most complicated of the functions in the type checker. Given that type systems are concerned with what type of value things return, and expressions are the primary way that these values propagate, this is unsurprising. Starting with the base cases, the three literal types for Booleans, characters and integers are all immediately typed with their respective values, as defined by the type rules.

ID expression are slightly more complicated in that they must first retrieve their definition from the environment, then return the type stored there. However, as long as the environment and the rest of the monadic pipe-work are implemented correctly, then this is fairly trivial. If the variable cannot be found then a scope error is output. It also gives an error if the checker has realised that there is a path of execution which leaves that variable uninitialised at this point.

Assignment expressions are very similar to ID expressions: they look up their type and give an error if they can't find a matching definition. The only main difference is they also check that the type of the expression it is assigning from matches up. Assignment expressions also set that variable as definitely initialised. However, because this may be inside some kind of branch statement, it may be merged with another instance of the environment and reduced from Yes to Maybe.

Binary operators and function calls are implemented in a similar way. The expressions signifying the arguments are evaluated and their types compared with the types and number of arguments expected, if they pass the test, then the return type of the function is returned as the type of the function call expression.

There are however two special cases for function calls. One is where it checks the scope level to make sure that is not zero. If the scope level is currently zero, that means the user is attempting to call a function from the initialisation expression of a global variable. This is not permitted and will cause an error. Because the way the CFG is structured, there is also the possibility of any expression having a bracketed set of parameters after it, and this creating a function call expression within the AST. It is important to catch this, because this is nonsensical and cannot be converted into valid target code.

6.5 (Checker.lhs) Code Generation (with some Optimisation)

The code generator is implemented with a very similar structure to the type checker. Generally, where the type checker calculates or looks up the type of an expression or function, the code generator calculates or looks up the location of that variable or function. This is largely because, as I mentioned before, it was implemented prior to the type checker, and had to do some of the work that would normally be done for it.

It is also slightly more crude than the structure employed by the type checker and has a few small bugs that writing this report has caused me to notice.

Just as I did with the type checker module, I will begin by explaining the set of variables that I hold in my monadic "state." The first of these is an iterative integer that is used to provide unique labels whenever JUMP or CALL instructions are required.

After this I keep track of a stack of the numbers of un-popped items on the memory stack. (These are usually the leftover values that were returned but never got consumed by anything that occur when an expression statement is converted.) The reason a stack is required rather than just a single number of items to pop is due to the way if statements are structured in the target language.

Similarly, there is also a stack containing the number of local variables declared in the current block so that they can be popped off again once their lifetime expires. Like the type checker, I also keep track of the name of the function which is currently being evaluated.

Finally, I also keep track of those functions that are called from the C-Flat standard library. This is so that I only have to import the code for these if they are actually used.

The monadic functions which manipulate the state are described briefly below:

`freshLabel` returns a new label string constructed from the string "label" and an unused integer from the state. It also increments the integer held in the first part o the state.

`newBlock` is called when the code for a compound statement is being generated. It prepends a zero to the head of both the stack of pops and the stack of locals to prepare for new local variable declarations in that block.

`push` and `pop` are largely equivalent, where one adds and integer to what's held at the head of the stack of impending pops and one subtracts. While we don't know what values will be returned, it's possible to know that a literal expression will cause one literal to be pushed onto the stack, and overall, application binary operation will result in one pop from the stack (two pops for arguments, and one push for the return value).

`getPops` is used when we want to find out how many unused values have been returned to the memory stack so that we can remove them. This is usually done, as previously mentioned, after generating an expression statement. It also sets the current head of the stack to zero pops.

`addLocals` and `getLocals` are essentially the equivalent of push and `getPops` but are used when declaring new local variables within a block. `getLocals` also does not reset the head of its stack like getPops does.

`endBlock` is the compliment to the `newBlock` function, which returns the number of local variables to clear, and drops the head of both the `Pops` stack and the `Locals` stack.

The getter and setter for the current function name are, as in the type checker module, self explanatory.

The function which is called to get code generation started is the `generateCode` function. What this function does is start by generating the initial environment detailing the lists of variables and functions. Unlike in the type checker module, it was more convenient to explicitly pass around the environment and take advantage of the fact that changes to that environment in called expression were not preserved.

The advantage of this is that I do not require, like I did with the type checker, a method of clearing all the variable definitions at a given scope when I've finished generating all the code at the scope.

The difference in the way that this functions to the type checker, is that the type checker ran through each global declaration one at a time, constructing the environment, and checking the initialisation expressions as it went. The problem here is that I construct the entire environment, and then use it to generate the code for the variable definitions. This has one unforeseen consequence later on when it comes to local variables. Doing it this way, you would be able to define the sequence of local variables as follows:

```
int globalVar = notInScopeYet;

int notInScopeYet = 5;
```

The reason I decided to ignore this problem is that the contextual analysis phase does catch this and prevent any code that contains it from reaching this phase. The only problem is, the same philosophy is applied to local variables: that of updating the environment with the new scoped variables and their calculated addresses, and then using that to give meaning to the initialisation expressions.

The subtlety lies in the fact that when shadowing a variable, the shadowed variable is (or should be) still in scope during that variable's initialisation expression. Consider the code fragment below:

```
someFunction(int i)
{
	int i = i;
}
```

Suppose I decide that I want to create a variable that shadows the argument so that I am essentially "enabling write privileges" on that argument. The type checker will view that as a perfectly acceptable, properly scoped and typed piece of code. This will then be passed on to the code generator which, due to the subtlety of *when* the environment is updated with the new local variable definitions, will interpret the address of the variable in the initialisation expression to be the same address as the place it intends to put it. Trying to access this piece of memory that hasn't been created yet will cause the virtual machine to attempt to read a stack location that is off the top of the stack, causing a run-time error.

Other than the one subtle case above, the code generator does do a very good job of generating code in a structured and correct, and that optimises the usage of memory on target machine.

Most of the code for the code generator looks very similar to the templates defined in the language specification, so I am going to focus on those cases where interesting things occur.

As mentioned in the description of the static modifier functions, it is possible to calculate the number of leftover items on the stack when an expression has been evaluated. There are four cases where expressions can occur within statements. The most trivial of all is when evaluating the return value of a RETURN statement. Because of the way in which this statement behaves, any redundant values on the stack are disposed of along with that entire stack frame and its arguments.

Almost as trivial is that of the expression statement, where the number of leftover values are simply added up as that expression's code is generated. Any call to a binary operator will pop two items from the stack and push one, averaging out to one pop. Literals will push one value to the stack, and so will variables. At the end of this, you have simply calculated the number of values to dispose of from the top of the stack. The complexity comes in when you evaluate the conditional expression of an else-less if or a while loop. You cannot immediately clear all the values from the stack, because the returned value of that statement is then used to decide whether or not to jump. Instead you must dispose of that number minus one, and use the offset to leave only that value at the top. Because JUMPIFZ and JUMPIFNZ do not consume the value at the top the stack when they jump, it then has to be popped at the very end.

It is due to the fact that you can have this "leftover" value when entering new blocks defined within if and while statements that gives the necessity of keeping a stack of values to be popped, rather than just an individual number. In reality, a similar machine language *would* consume the value at the top for a conditional jump, but when I realised the mistake in my implementation of the virtual machine, I decided that since you can't just change how the target language functions when writing a real compiler, to just work around the machine I had built.

7 Project Evaluation

In terms of my progress with the project as a whole, I found that it involved a very steep learning curve. I was in the position of having never implemented a compiler from scratch before, and faced with the task of building a large and complicated program in a language that interested me, but I wasn't particularly experienced with.

Having studied some of the structures required to write complex algorithms in Haskell, I still found that my understanding of those structures was very limited to begin with. Broadening my understanding and teaching myself to think functionally has been an interesting but challenging experience for someone accustomed to programming in imperative languages.

I do feel now though that I am much more comfortable with Haskell as a real and usable programming language, rather than as something of an academic curiosity.

7.1 Scanner and Parser

One key thing to note about the implementation is that it is possible to use a monad, accompanied by error recovery clauses within the Happy source, file which allow the parser to continue compiling after an error, and collect up error messages.

This is something I very much would have liked to be able to implement, however, given that the Scanner/Parser were the first major components that I constructed, I didn't possess a good enough understanding of how to accomplish this. A Writer monad would have done a good job of providing that. The source code locations in lines and columns could have been collected up along with the tokens by recording how my characters I had consumed so far and the number of `'\n'` characters encountered.

It is unfortunate that my understanding of the structures required was not adequate until nearer the end of the project. I feel that being able to point the user to the exact rows and columns where the parser encounters errors is an exceptionally useful programming tool. Failing catastrophically when a parse error occurs, and being able to tell the user what token it failed at does provide some hint, but is not particularly useful.

7.2 Contextual Analyser and Code Generator

In truth, this part of the project was what I found the most enjoyable. Being able to do such an in depth analysis of the source code and prove the absence of a wide variety of different errors was very rewarding. If I were to extend the project, I would have liked to have added in pointers, arrays, and maybe more complex structures such as OO-style objects.

I feel that Strings in particular are a noticeable deficiency, and one that I would have liked to have implemented if I'd had more time.

References

[1] A. Aho, M Lam, R Sethi, J Ullman, *Compilers: Principles, Techniques and Tools* 2nd Edition. Addison-Wesley (2007) page 1.

[2] B. Kernighan, D Ritchie, *The C Programming Language (ANSI C Version)* 2nd Edition. Prentice Hall (1988)

[3] H. Nillson, *G53CMP Compilers - (Implementation and associated slides),* http://cs.nott.ac.uk/~nhn/G53CMP/ (accessed May 2012)

[4] D. Watt, D Brown, Programming Language Processors in Java: Compilers and Interpreters. 1st Edition. Prentice Hall (2000).

[5] H. Nillson, *G54FOP Mathematical Foundations of Programming (Lecture 1),* http://www.cs.nott.ac.uk/~nhn/G54FOP/LectureNotes/lecture01.pdf (accessed May 2012)

[6] A. Aho, M Lam, R Sethi, J Ullman, *Compilers: Principles, Techniques and Tools* 2nd Edition. Addison-Wesley (2007) page 109.

[7] A. Aho, M Lam, R Sethi, J Ullman, *Compilers: Principles, Techniques and Tools* 2nd Edition. Addison-Wesley (2007) page 192.

[8] K. Cooper, L. Torczon, *Engineering a Compiler*, 2nd Edition. Morgan Kaufman (2012) page 164.

[9] H. Nillson, *G54FOP Mathematical Foundations of Programming (Lecture 1),* ttp://www.cs.nott.ac.uk/~nhn/G54FOP/LectureNotes/lecture13.pdf (accessed May 2012)

[10] A. Aho, M Lam, R Sethi, J Ullman, *Compilers: Principles, Techniques and Tools* 2nd Edition. Addison-Wesley (2007) page 10.

[11] H. Nillson, *G54FOP Mathematical Foundations of Programming (Lecture 1),* http://www.cs.nott.ac.uk/~nhn/G54FOP/LectureNotes/lecture15+16.pdf (accessed May 2012)

[12] G. Hutton, *Programming with Effects*, http://www.cs.nott.ac.uk/~gmh/monads

[13] B. Kernighan, D Ritchie, *The C Programming Language (ANSI C Version)* 2nd Edition. Prentice Hall (1988) page 212.

Apendix – Test code

Below are some source files that I used to test the data. Between them, I have been sure to include some of the more unusual aspects of the language, for example the fact that assignment actually returns a value as an expression, in order to fully test all of the syntactic constructs. During testing I would change types, and deliberately attempt to modify the source to explore all the the cases in the definitions of each function:

Test1.c

```
int test = 3;
char type2 = '\n';

anotherFunction (int test, char blah)
{
             anotherVar = 4;
}

int main ()
{
             if (test % 'c' == 2)
             {
               while(5);
             }
             else if (1)
               test = 2;
}
```

Test2.c:

```
int test = 333;
char type2 = 555;
int fact(int a)
{
	if (a == 1)
		return 1;
	else
		return a * fact(a - 1);
}
void hello(int a, int b)
{
	type2 = 4;
	return;
}
main()
{
	int test2 = 3;
	test2;
	{
		int hi = 4;
		char hey = 'h';
		hi;
		{
			int there = 67;
			there;
		}
		//test = hi - 2;
	}
	test = fact(4);
	if ((test2 < 2) == false)
	{
		bool foo = true || false;
		type2 = (test = fact(5)) + 5;
	}
	else
	{
		1 + 3;
	}
	hello(212, 313);
	//function(2, 3);
}
```

Test3.c

```
int test = 2;
char type2 = test = 3;
bool booltype;

int fun(int a)
{
		if (a < 1)
		  return 0;
		else if (a == 1)
		  return 1;
		else
		  return a * fun(a - 1);
}
void hello(int a, int b)
{
		type2 = 3;
}
main()
{
		char test;
		if (true) {test = fun(5);} else {test = 4;}
		printi(test);
		exit();

}
```