David Hoare

Robinson College

Computer Science Tripos

Part II Dissertation

The implementation of a compiler from BASIC 78 to LLVM

2015

Proforma

David Hoare

Robinson College

The implementation of a compiler from BASIC 78 to LLVM

Computer Science Tripos Part II (2015)

<word count>

Project Originator: Michael Gale

Project Supervisor: Michael Gale

Original Aims

The aim of the project was to implement a compiler for the ANSI X3.60-1978 standard for BASIC. The compiler was to output LLVM bytecode which can then be further compiled and assembled to target one of many instruction sets. The compiler was to be written in C#, and would make use of the LLVM API in the code generation module.

Work Completed

A compiler has been written in C# that efficiently compiles BASIC code to appropriate LLVM bytecode. The compiler adheres to the ANSI X.60-1978 standard for BASIC, with some additions to improve usability. The resultant LLVM bytecode successfully compiles to multiple different architectures, on which the executable files run faster than the equivalent interpreted code.

Special Difficulties: None

I David Hoare of Robinson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Contents

Chapter 1

Introduction

The BASIC (Beginner's All-purpose Symbolic Instruction Code) programming language was an important development in the field of Computer Science in the 60s and 70s. It was the first very human-readable, entry-level programming language aimed at beginners and home users, and a BASIC interpreter was included on most of the personal computers released at this time. Thus the BASIC programming language introduced a generation to the basics of programming. It has also influenced many modern languages, most notably Visual Basic .NET, one of the primary languages targeting Microsoft's .NET framework. Undoubtably, given the longevity of legacy code, there still exists software written in BASIC that is still in use, or would be were it not for the fact that BASIC is no longer widely distributed.

LLVM (originally Low Level Virrtual Machine) is a framework for compiler construction. The principle is that compilers built with the LLVM framework output LLVM Intermediate Representation (IR), a platform-independent bytecode assembly language. This bytecode can then be compiled by an LLVM backend to native code for a target architecture. Backends exist for targeting a wide variety of instruction sets. Hence building a compiler with LLVM is an easy way of enabling the compiler to build for multiple platforms.

The LLVM framework also performs code optimisation passes on generated bytecode. This is made easier by the Static Single Assignment (SSA) form of LLVM bytecode. SSA code produces a simple definition-use tree which simplifies dataflow optimisations. In particular, data redundancy detection can be performed immediately on SSA code, avoiding expensive reaching definition

analysis required for non-SSA code. [REF] LLVM-based compilers are therefore able to produce machine code that often runs faster than that produced by conventional compile pipelines.

The project brief was simple – to develop a compiler targeting LLVM IR for the BASIC programming language. The ANSI X.60-1978 BASIC standard was selected as an appropriate starting point for the definition of the language, with new features being added as necessary to enhance the language and increase usability. For example, the 1978 BASIC standard lacks the ability to read and write files, so I decided this would be a useful extension to implement.

The primary motivation for the project was to test whether a compiler for BASIC built on the LLVM framework would perform better than an existing compiler built in a more conventional way. The large number of optimisations offered by LLVM suggested that this should be the case. However, the simple and low-level nature of BASIC means the optimisations may not have as much of an effect as with a more complex structural language.

Chapter 2

Preparation

The BASIC specification contains a full context-free grammar for the language, along with details of how each rule should be compiled and the expected behaviour for each statement. The context-free grammar would have to be adapted to fit the format expected by the lexer/parser generator I use. I would have to refer back to the specification when writing the code generation module of my compiler to ensure the generated code behaved as required by the standard.

I chose to structure my compiler in two parts – a lexer/parser and a code generation module. The parser would take input BASIC code and produce an abstract syntax tree. This would then be passed to my code generation module to produce LLVM bytecode. The parser would be generated by a parser generator. This would then need to be adapted so it produced instances of the classes I had defined for my internal representation.

The first stage in any compiler is the lexer, which converts input code to a stream of lexical tokens to be passed to the parser. To avoid the need to build a lexer from scratch I elected to use a lexer generator to produce this component. I experimented with the lexer generator tool Flex. A lexer generated by this tool converts input code into a stream of lexical tokens, which must then be run through a parser to generate an Abstract Syntax Tree (AST). These two components could be combined by using a parser generator. A generated parser would lex code internally and then parse it into a complete AST. I selected the ANTLR 4.5 parser generator for this task. ANTLR has the advantages that it takes a standard context-free grammar as input and also has a well-documented C# target.

A language needed to be selected with which to implement the compiler. I chose C# .NET for this purpose because of its object-oriented design. This would allow the AST to be designed using principles of inheritance and overloading, making my code more coherent and maintanable. There was also the added advantage that I was already familiar with using C# .NET.

I designed a set of classes to represent the BASIC code once it had been processed by the parser. The most important class is `Statement`. Each line of BASIC would be parsed into an instance of the `Statement` class and the AST would be represented as a list of `Statements`. I designed a set of 14 subclasses of `Statement` to represent the different types of statement defined in the BASIC specification (`Statement_Goto`, `Statement_Input` etc). Lines of BASIC could therefore be parsed into an instance of the appropriate class while still being treated as an instance of `Statement`. I decided to place general methods relevant to all statements in the `Statement` class, while adding purpose-specific methods to each subclass. `Statement` would also contain a `code()` method for generation of LLVM bytecode. This could then be overridden by each subclass. I then designed classes for representing all features described in the specification. These included classes for constants, variables, value expressions and relational expressions.

From the language reference for LLVM [REF] I was able to determine how each rule in the context-free grammar would translate into LLVM bytecode. Some constructs such as simple arithmetic operations and `GOTO` statements map directly to LLVM instructions. Other structures such as assignments would need to be compiled with a combination of several instructions. Some statements such as `PRINT` would be particularly impractical to implement in LLVM bytecode. Such an implementation would, by definition, duplicate the implementation of similar functions in the C standard library. Thus a much better approach to compiling these statements would be to use the LLVM `call` instruction to invoke appropriate C library functions and pass relevant variables as arguments.

The code generation module of the compiler involved use of the LLVM API. A complication to this step was the fact that there does not exist a complete set of bindings of the (very extensive) API for .NET development. However, several attempts at writing said bindings exist albeit at differing stages of completeness. The solution was therefore to select an appropriate set of bindings and then add to them when required functionality is not implemented. After investigating several options I opted to use the bindings written by my project supervisor Michael Gale some years ago.

I chose to develop the compiler using an iterative software engineering principle; instead of completing the integration of the parser then working on the code generation, I opted to integrate portions of the parser iteratively then implement the relevant portions of code generation. This methodology has the advantage that development progress is rapid, problems are identified sooner because I could see working end-to-end results at an earlier stage and it was easier to identify components that could be reused.

Chapter 3

Implementation

3.1 Reading in file

BASIC is a rigidly line-based language; statements and expressions cannot span over multiple lines, nor can multiple statements appear on a single line. The most important control-flow statement is `GOTO` and loop blocks are enclosed by `FOR` and `NEXT` lines. For this reason I decided to parse the BASIC code on a line-by-line basis. A stream reader is used to read from the input file. As each line is read it is passed individually to the parser. This approach to parsing is reflected in the AST base structure being a list of `Statement`s, as each line of BASIC contains exactly one BASIC statement, which is parsed into a `Statement` object.

3.2 Parser

The ANSI X3.60-1978 standard for BASIC contains a Context-Free Grammar (CFG) for the language. I adapted this into the format required by ANTLR. The format of the grammar in the specification does not explicitly differentiate between parser and lexer rules, but ANTLR requires parser and lexer rules to be defined separately. To debug the CFG I used ANTLR 4.5's `TestRig` component, which produces a graphical tree representation of how an input string is parsed by the grammar [example downstairs]. This allowed me to pass lines of BASIC in and ensure that they were being parsed as expected. This was

particularly important in the writing of the rules for expression parsing, as it allowed me to verify operator grouping rules (eg BIDMAS) were being obeyed by the parser. Once the grammar correctly interpreted the language I used ANTLR to produce the C# class files that constitute the parser. These classes then had to be integrated with the code generation module of the compiler.

I chose to integrate the parser by use of a listener module. This works by a unique method being called on entering or exiting each parser rule. These methods can then construct the internal representation of the code used by the code generation module. I implemented the `BASICListener` interface produced by ANTLR. This provided stubs for all methods required for the listener. I initially integrated a small subset of the parser and then implemented the code generation module for this subset. This allowed an end-to-end demonstration of the compiler from an earlier stage.

At this stage of compilation some syntax errors are detected and C# exceptions are thrown. Errors thrown by ANTLR (i.e. input that could not be matched to a rule by the parser) are handled. Other simple errors that are detected and handled include missing statements and missing print statement subjects.

3.3 Code Generation Tools

The remainder of development was the module that actually generates LLVM bytecode from the AST.

3.3.1 LLVM API

I made extensive use of the LLVM API for the generation of the LLVM IR code. The API works by defining a global context, and then splitting an input program into modules, functions and basic blocks. Because my compiler only works with a single input source file, and does not require subroutine calls (as the 1978 BASIC specification does not allow for subroutines), I was able to use a single LLVM module containing a single function, and hence a single entry point into the code. All[DAH8] sections of code dealing with code generation make use of the LLVM context, module and main function variables. To minimise the need to pass these as arguments, I create a single instance of each variable and refer to these instances throughout compilation. This is achieved by use of

the singleton design pattern - they are stored as public static members in the Parser class.

Within the main function are LLVM structures called basic blocks. These are containers for lines of LLVM bytecode, and are represented in LLVM IR by a labelled section of code. To insert code into a basic block I made use of the LLVM `IRBuilder` class. This provides methods to insert IR instructions into a basic block. All kinds of BASIC statements can be compiled to a single basic block. Hence each line of BASIC corresponds to exactly one LLVM basic block - this fact makes compilation of control-flow statements more straightforward. The basic block corresponding to each BASIC statement is stored as a member in the relevant `Statement` object.

3.3.2 C Standard Library

To compile more complex statements such as `PRINT` and `INPUT` [see below], and arithmetic function calls such as `SIN` and `SQR` [see below], I chose to make use of C standard library functions. This avoids the need to write these procedures from scratch but requires that the standard library is available at link time for static linking, or at runtime for dynamic linking.

For example, consider the simple BASIC program in Listing 3.1. My compiler produces the bytecode seen in Listing 3.2. The C library function `double sqrt(double)` is imported at the bottom. The assignment on line 10 is compiled by allocating the space, calculating the result then storing the result in the allocated space. The calculation is performed by calling the imported library function.

Listing 3.1

```
10 LET A = SQR(9)
20 END
```

Listing 3.2

```
define i32 @main(i32, i8**) {
line10:
%A = alloca double
%2 = call double @sqrt(double 9.000000e+00)
store double %2, double* %A
br label %line20
line20:
ret i32 0
}
declare double @sqrt(double)
```

3.4 Types

3.4.1 BASIC types

The BASIC standard defines just two types – string and numeric, with no differentiation between integer and floating point values. My compiler addresses this by representing all numbers internally as doubles. This allows all operations to be seamlessly compiled regardless of the initial type of the numbers. However, this approach requires additional steps when outputting numeric values to hide decimal places when the user expects to see an integer [see 3.10.3]

This approach to storing numeric values has two potential downsides. Firstly, it uses more storage space than necessary for integers. In C# a double has a size of 64 bits. A long is also 64 bits, and a signed int is 32 bits. Thus when storing integer values it can be wasteful to use a double when an int would suffice. Secondly, the floating point nature of a double means that while a very large range of numbers can represented (much greater than that representable with either an int or long), precision can be lost when representing very large positive or negative numbers. The mantissa size of a double is 52 bits, offering a precision of between 15 and 16 significant figures. This is greater than the precision offered by an int, but less than that provided by a long. These issues notwithstanding, the use of doubles throughout was a sensible comprimise to achieve seamless usability

Listing 3.3

```
numericexpression : sign? term (sign term)*;
term              : factor (multiplier factor)*;
factor            : primary (CIRCUMFLEXACCENT primary)*;
multiplier        : ASTERISK
                  | SOLIDUS;
primary           : numericvariable
                  | numericconstant
                  | numericfunctionref
                  | LPAREN numericexpression RPAREN;
```

3.4.2 LLVM types

The code generation module makes extensive use of the LLVM API's type construct `LLVM.Type`. To minimise calls to the API all common types (`i8`, `i8*`, `i8**`, `i32`, `double`, `double*`, `void`) are initialised at the start of execution and are stored as static members in the `Parser` class. This minimises calls to the LLVM API.

The `LLVM.Constant` corresponding to the number zero is also frequently used in various types in the code generation module. Hence I have also chosen to store the zero constant as static members in the `Parser` class as an 8-bit integer, a 32-bit integer and a double.

3.5 Expressions

3.5.1 String expressions

The BASIC standard specifies that string expressions consist of either a string variable or a string literal. Thus string expressions are very simple and no operations on strings are possible.

3.5.2 Numeric expressions

Numeric expressions are much more versatile than string expressions. The parser rules for a numeric expression are as follows:

This structure allows complex recursive numeric expressions to be parsed correctly in accordance with BIDMAS rules. An example of how complex expressions are parsed is shown in figure 3.1. This was generated by the ANTLR4

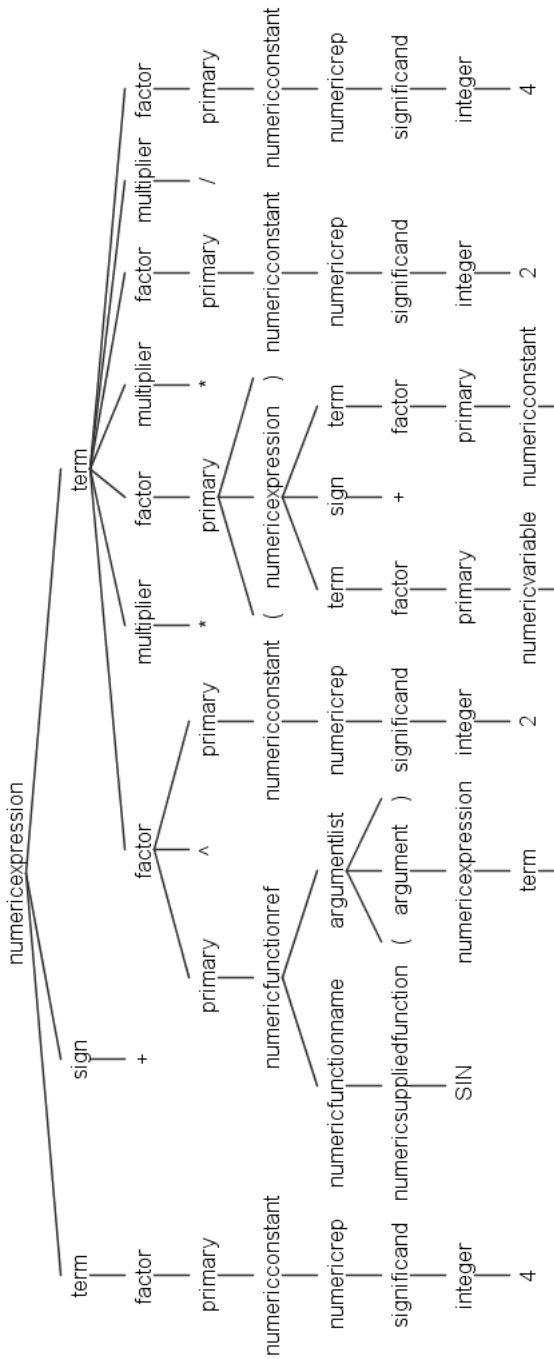

Figure 3.1: The parse tree for `4+SIN(0.5)^2*(A+1)*2/4`

Listing 3.4

```
%2 = call double @sin(double 5.000000e-01)
%3 = call double @pow(double %2, double 2.000000e+00)
%temp = load double* %A
%addtmp = fadd double %temp, 1.000000e+00
%multmp = fmul double %3, %addtmp
%multmp1 = fmul double %multmp, 2.000000e+00
%divtmp = fdiv double %multmp1, 4.000000e+00
%addtmp2 = fadd double 4.000000e+00, %divtmp
```

Listing 3.5

```
%A = alloca double
store double 1.000000e+01, double* %A
```

`TestRig` tool, which I used to debug my context-free grammar for BASIC. The tree represents the expression `4+SIN(0.5)^2*(A+1)*2/4`. The adherence to BIDMAS rules is shown by the fact that the tree branches in the correct order. i.e. first at plus signs, then by multipliers, then by indices, then by brackets, then by function calls. The LLVM bytecode for evaluation of this expression is as follows:

We can see that the `sin()` function is handled first, by calling a C library function [see 3.3.2 or 3.91]. The power is then compiled in the same way. Then the bracket is evaluated by loading variable `A` and adding `1` to it. The multiplication and division operations are then compiled, with the outer addition being performed last.

Numeric expressions also demonstrate the earliest optimisation performed by the LLVM API. Consider the trivial example expression `1+2+3+4`. This is parsed into the parse tree shown in figure 3.2

My compiler compiles this expression by emitting three `fadd` instructions, feeding in a constant double and the result of the previous instruction [see how I compile expressions][DAH9]. However, the resultant LLVM code for the statement `LET A=1+2+3+4` is as follows:

We see that the LLVM API has detected that the repeated `fadd` instructions are pointless and inefficient and has simplified the code to the equivalent to `LET A=10`.

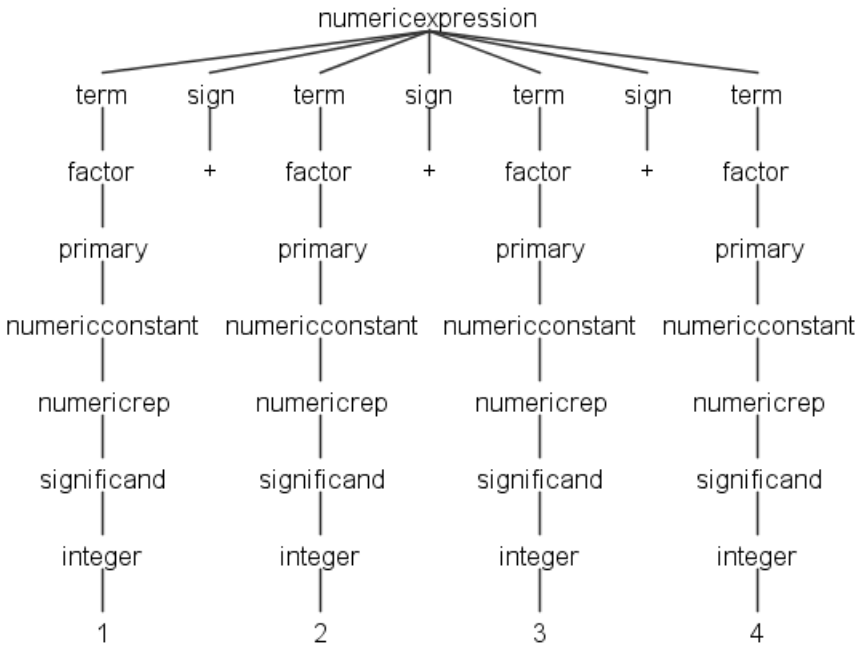

Figure 3.2: The parse tree for `1+2+3+4`

3.6 Compiler passes

My compiler makes three passes through the list of `Statement` objects returned by the parser.

3.6.1 First pass

The first pass calls the `code()` function on each line, which generates the LLVM IR for the line. Recursive `code()` calls are made to instances of AST classes representing the structures used within the line to compile these structures. After each line is processed, a reference to the corresponding `Statement` object is placed in `Parser.variables.lines`, a dictionary indexed against BASIC line numbers. This dictionary is used to compile `GOTO` and other control-flow statements in the third pass. `Parser.variables` is a static instance of `VariableStore` [see 3.8]

3.6.2 Second pass

The second compiler pass calls the `jumpToNext()` method on each line, passing a reference to the sequential next line of the program. This method is defined in the abstract `Statement` class and inserts a `br` (branch) instruction at the end of the `BasicBlock` associated with the line, causing a jump to the `BasicBlock` associated with the following line.

The `jumpToNext()` method is overridden in classes inheriting `Statement` that represent lines where this jump is not necessary, i.e. lines containing control-flow statements. For example, the override `jumpToNext()` method for `Statement_Goto` is empty, as execution should not pass to the next line after a `GOTO` statement.

The `jumpToNext()` method for `Statement_Next` differs from other statements. Here a jump to the next line is not created, but a reference to the basic block for the next line is stored in a member variable. This reference will be needed to construct the conditional jump created in the third pass. Similarly the `jumpToNext()` method for `Statement_For` stores the following line in a variable, but also creates a jump to it, as execution always passes to the next line following a `FOR` statement.

3.6.3 Third pass

The final compiler pass calls the `processGoto()` method for each line. The body of this method in `Statement` is blank, as for most lines there is no need for further compilation. This pass is only used for lines which contain control-flow statements. A case in point is the `GOTO` statement. This cannot be compiled earlier as it depends on the ability to translate an in-code line number to a `BasicBlock` reference for the target line. This cannot be done until all lines have first been processed, and the `Parser.variables.lines` dictionary has been filled [see 3.8.1 and?]. Hence in this pass a `Statement_Goto` will retrieve the `BasicBlock` associated with the in-code line number and create a `br` instruction to this block. Similar operations are performed in this pass for lines containing `FOR`, `NEXT` or `IF` statements, as these all depend on the `BasicBlock` for a target line being initialised.

3.7 Error Handling

3.7.1 Exceptions

I have defined a new `System.Exception` which I have named `CompileException`. I use this to represent handled compile-time errors, i.e. errors where the compiler knows what has gone wrong. These exceptions can then be differentiated from other `System.Exceptions` in catch blocks to provide context-sensitive debug information to the user. The constructor for `CompileException` takes a string containing a descriptive error message to be displayed to the user.

Throughout all stages of compilation, the line number currently being processed is stored statically in the `Parser` class as `Parser.lineNumber`. This means that should a `CompileException` be thrown at any point during lexing, parsing or code generation, the error message can include the line number where the error is found. This has obvious advantages for users debugging their BASIC code.

3.7.2 ANTLR error listener

When initialising the ANTLR `BASICParser` class used to parse the input code, the default ANTLR `BaseErrorListener` is removed from the `BASICParser`, and an overloaded error listener class is attached instead. I have named this class `ANTLRErrorListener`. This overrides the default error behaviour (to output any error immediately to stdout) so the more advanced error information supplied by the ANTLR system is only displayed if my compiler has been executed with the `-debug` flag set. This error information includes the line number where ANTLR encountered the error, but as the source code is fed to the parser line-by-line this number is always 1. `ANTLRErrorListener` also addresses this problem by instead outputting `Parser.currentLine`, which contains the correct BASIC line number.

3.8 VariableStore

3.8.1 Storage of line and variable addresses

The `VariableStore` class is used to contain dictionaries and lists which store allocation addresses and other key information relating to variables defined and used in the BASIC code. `VariableStore` is initialised as a public static variable

`Parser.variables` in the `Parser` class, and is thus accessible throughout the compilation process. [strings vs stringpointers + stringisPointer] [purpose for each dictionary].[DAH10]

3.8.2 Array operation methods

`VariableStore` also contains two important methods for the handling of arrays, `initialiseArray()` and `arrayItem()`. These initialise a numeric array and return a pointer to an item in the array respectively.

Initialisation involves emitting a simple `alloca` instruction but including a second argument stipulating the number of items for which memory should be allocated. The `initialiseArray()` method takes any `LLVM.Value` as the array size, allowing arrays to be initialised with arbitrary length. Thus an array can be defined with a length specified by the user at runtime. This poses a slight problem in that numeric values are stored as doubles throughout, and array indexing is one of the few times where an integer value is required. To get around this an `fptoui` instruction is emitted. This casts the supplied floating point value to an unsigned integer, casting towards zero, which can be used to initialise the array. A reference to the head of the array is stored in arrays, a dictionary indexed against the name of the array. The size of the array is stored in the dictionary `arraySizes`; this is necessary for compilation of a `WRITE` statement, where the call to helper function `writeArrayToFile` requires the length of the array.

`arrayItem()` simply retrieves the array pointer from arrays and returns a pointer to the requested item. Again, the `LLVM.Value` supplied as the array index is cast to an unsigned integer. A call to `arrayItem()` before `initialiseArray()` indicates that the array has been referenced before it is defined with a corresponding `DIM` statement. In many languages this would result in an exception, however the BASIC specification states that this is actually allowed, and the array in question should be initialised with a default length of (rather oddly) 11 items.

3.9 Numeric functions

3.9.1 Supplied numeric functions

The BASIC standard defines eleven supplied numeric functions that must be included in the implementation. These functions (`ABS`, `ATN`, `COS`, `EXP`, `INT`, `LOG`, `RND`, `SGN`, `SIN`, `SQR`, `TAN`) all map fairly directly onto C equivalents. Hence I chose to compile these by calling the relevant C function. I took a similar approach to compile `PRINT` and `INPUT` statements, making calls to `printf` and `scanf` respectively. This means the standard C library will need to be present at link time (a reasonable assumption).

3.9.2 Defined numeric functions

To easily broaden the functionality of the language, I implemented the ability to call external C functions defined in files included at link time. These function calls can either have a single double argument or none. A key example was in the fast Fourier transform algorithm I implemented as part of the evaluation process, where I used this feature to implement the modulo operator and pi [see eval].

3.10 I/O

3.10.1 stdin/stdout

stdin and stdout are read/written to with the `PRINT` and `INPUT` statements. These are compiled by calling the C functions `printf` and `scanf` respectively[DAH11].

3.10.2 File I/O

A useful feature missing from the original BASIC spec is file IO. Particularly when using algorithms which worked with large arrays of numbers it became tedious, not to mention error-prone, to type in input data at the command line and read output data from stdout. To address this I defined two additional statements – `READ` and `WRITE`. The format of these statements is:

where A is a pre-defined array and filename is a string expression containing the filename to read/write. These statements are compiled by calling two C helper functions I placed in the file `libBASICLLVM.c` [appendix]. These helper functions are passed pointers to the array and the string containing the filename,

Listing 3.6

```
READ A filename
WRITE A filename
```

Listing 3.7

```
LET A = 10
PRINT A
```

along with the length of the array being passed. They open a file pointer to the specified file and use `fgets`/`fprintf` to perform the operation.

3.10.3 Numeric representation in I/O

As mentioned in [3.4.1] numeric values are always represented as doubles in the LLVM bytecode generated by my compiler. This addresses the ambiguity of numeric data types in the BASIC specification. However this poses a problem when outputting numeric variables using `PRINT` or `WRITE`. Consider the following example code

The user would reasonably expect the output to be "10". However the number is being represented internally as the double "10.00000..." and would be output as such. I chose to address this by using the format argument accepted by `printf` and `fprintf`. This is a string argument which specifies how a numeric variable should be represented as a string. The "`%g`" format specifier outputs the number using the shortest possible representation, so the double "10.00000000..." is output as "10", while numeric values which require a higher level of precision are still returned to an appropriate number of decimal places.

3.11 Command line options

I implemented a small number of command line options to make the compiler more usable and versatile. The `-debug` flag is used to enable verbose output. Without this option the compiler will only output to stdout if there is an error. With `-debug` enabled other diagnostic information is displayed. For example, errors reported by the ANTLR parser are shown, and the entire LLVM module is dumped to stdout once code generation is complete. The `-o` option is used to specify an output file.

The `-output` option instructs the compiler to perform further processing on the generated bytecode. The options are `LL`, `S` or `EXE`, which produce a `.ll` bytecode file, a `.s` assembly file or an executable respectively. This option is discussed further in [3.12 ?]

3.12 Further compilation and assembly

3.12.1 llc

By passing my compiler the command line option `-output=S`, after compiling BASIC source to LLVM bytecode, the bytecode is further compiled into an assembly file. This works by calling the `llc` tool provided with LLVM. This is a static compiler for LLVM bytecode.

3.12.2 gcc

By passing my compiler the command line option `-output=EXE`, the output bytecode is first passed to `llc` for compilation to assembly, then the assembly code is passed to `gcc` (the GNU Compiler Collection [ref?]). This is a versatile open-source compiler. Here it is used simply to assemble and link the assembly generated by `llc` and produce an executable file. In addition to the generated assembly, `gcc` is passed the library file `libBASICLLVM.c` which contains the helper functions necessary for the File IO functions [see 3.9]

3.12.3 Compiling for other architectures

By passing the `-output` option to my compiler, an executable is produced which will run natively on the current machine. However, a key advantage of using an LLVM-based compiler is that the bytecode can be compiled for a variety of different architectures. This can be achieved by specifying `-output=LL` (or not passing an output argument) and then running the bytecode through `llc` and `gcc` separately. [see 4.2 for example]

The target architecture is specified to `llc` by use of a string descriptor known as a target triple. A target triple generally takes the format `<architecture>-<vendor>-<system>`. Examples of architecture include `i686`,`arm` and `mips`. Examples of vendor include `pc`, `apple` and `ibm`. Examples of system include `win32`, `linux` and `darwin`. So for example target triple for the machine I was developing on is

`i686-pc-win32`. The target triple is passed to `llc` by using the `-mtriple=argument`.

3.12.4 Soft vs hard float ABI

The application binary interface defines how code modules interact with each other, including the way arguments are passed to functions. In my case, this applies to the way my code passes arguments to functions contained in `libBASICLLVM` and to C library functions. The Raspberry Pi uses a "hard float" ABI (Application Binary Interface). This means floating point arithmetic is supported by the hardware. Conversely, soft float ABI systems do not have dedicated floating point hardware, and floating point arithmetic must be handled with software.

The two float passing conventions are not compatible. Thus assembly targeting a Raspberry Pi must be compiled with hard float ABI specified so it links correctly with the C libraries present on a Raspberry Pi. This is done with the `-float-abi=hard` argument when calling `llc`.

Chapter 4

Evaluation

The initial aim of the project was to produce a correctly working compiler. Therefore the obvious first step was to run a number of test programs through the compiler to ensure the expected output was produced. This was a valuable bug-finding exercise and verified that the compiler was indeed correctly compiling the BASIC code. To assist with this task I made use of LLVM's `lli` tool. This is an interpreter which works directly on the LLVM IR bytecode produced by my compiler, running it by use of a just-in-time compiler. This allowed me to rapidly test output without needing to compile the bytecode to native assembly.

Part of the reason for using LLVM was the cross-platform nature of its backend, allowing a variety of architectures to be targeted by the compiler. I was developing on a 64-bit Windows machine, on which code was being successfully compiled and run. To test the cross-platform capabilities of the compiler, I compiled some BASIC test code for the ARM platform, 32-bit Windows and 64-bit Linux architectures. I then assembled and ran the resultant assembly on machines using the respective instruction sets. To test the ARM output I used a Raspberry Pi [see 3.11.4?]. The code compiled and ran successfully on all architectures I tested it on.

4.1 LLVM optimisation passes

The level of optimisation performed by LLVM when compiling bytecode to assembly is selected by using the `-Ox` argument when calling `llc`, where `x` is a number from zero to three. To demonstrate the impact this has on assembly

Listing 4.1

```
LET A = 3
LET A = 5
LET B = A
PRINT B
END
```

Listing 4.2

```
A1 = 3
A2 = 5
B1 = A2
print(B1)
```

size, I wrote the following trivial BASIC program:

The program is clearly inefficient - the first line is not needed as the value it assigns A is superseded by the following line. Correction of these types of inefficiency are simplified by LLVM's use of static single assignment form, which transforms the code into something more like this:

We can see that variables are only assigned once, and it is easy to see that the `A1` variable is never used after definition and is therefore redundant and can be removed. It is also easier for an optimising compiler to make this observation as the use-define tree is more straightforward to calculate.

I ran this program through my compiler to compile it to bytecode, then passed it to `llc` with the `-O0` flag set. This tells `llc` not to perform any optimisation passes when compiling the code. The resultant assembly file size was 1,315 bytes. By passing the `-O3` flag (most aggressive optimisation option instructing `llc` to perform the largest number of optimisation algorithms) I produced a file of size 898 bytes. This is a substantial improvement for such a trivial inefficiency.

I performed the same experiment with my BASIC fast Fourier transform example (which from inspection appears to be efficiently coded). The assembly file without optimisation was 40KB and with 3-pass optimisation was 21KB.

4.2 FFT for Raspberry Pi

As a real-world example and to give my compiler a real workout I implemented the Fast Fourier Transform (FFT) algorithm in BASIC. This is an efficient algorithm for computing the Fourier transform of a sampled signal which decomposes the signal into its constituent frequencies and their relative sizes. I found an implementation of FFT for a different dialect of BASIC [http://www.nicholson.com/dsp.fft1.html] and adapted it to fit my specification.

The FFT program gave me an opportunity to test the `NumericDefinedFunction` feature. The algorithm makes use of pi and the modulo function, neither of which are standard 1978 BASIC "supplied" functions. I wrote two simple functions `MOD2()` and `PI()` in C which wrap around the library function `fmod()` and C constant `M_PI` respectively [appendix]. My compiler detected the non-supplied function and passed it through in the assembly. By linking the compiled code with the helper functions the code worked correctly i.e. the output data from the FFT procedure was identical to that generated by Excel's FFT function on the same data.

As an example of the ability to target multiple architectures I compiled my Fast Fourier Transform program to run on a Raspberry Pi. I ran `gcc -dumpmachine` on the Raspberry Pi which returns the appropriate target triple for the current system (in this case `arm-linux-gnueabihf`). I ran the source code through my compiler to generate `fft.ll`. I then ran this through llc using the argument `-mtriple=arm-linux-gnueabihf`. I also passed the `argument -float-abi=hard`. This instructs `llc` to generate assembly which uses "hard float ABI" (this is explained in the previous section). `llc` produced `fft.s` which I transferred to the Raspberry Pi then ran through gcc. I used the command `gcc fft.s libBASICLLVM.c additionalFunctions.c -o fft.out` which assembled and linked three source files to produce the executable `fft.out`. This ran and produced the expected output.

Chapter 5

Conclusions

Blah

Chapter 6

Bibliography

Blah

Chapter 7

Appendices

C helper functions

The following C functions are called when compiling the READ and WRITE statements.

Listing 7.1

```
blah
```

Fast Fourier Transform Algorithm

The following BASIC code for performing a fast Fourier transform is based on an implementation for a different dialect of BASIC found here: [http://www.nicholson.com/dsp.fft1.html]

Listing 7.2

```
PRINT "F?"
INPUT F
PRINT "m?"
INPUT M
PRINT ""
LET N = 2^M
DIM V(N)
DIM G(N)
READ V "v.csv"
READ G "g.csv"
PRINT V(0),V(1),V(2),V(3)
PRINT G(0),G(1),G(2),G(3)
LET Q = 0
LET P = 0
LET A = 0
FOR K=0 TO N-1
LET X = K
LET R = 0
FOR I=1 TO M
LET R = R*2
IF MOD2(X) <> 1 THEN 20
LET R = R+1
20 LET X = INT(X/2)
NEXT I
IF R <= K THEN 40
LET Q = V(R)
LET V(R) = V(K)
LET V(K) = Q
LET P = G(R)
LET G(R) = G(K)
LET G(K) = P
40 NEXT K
LET B = 2
45 IF B > N THEN 60
LET C = B/2
LET D = N/B
FOR Z = 0 TO C-1
```

```
LET A = Z * D
LET A1 = 2*PI()*A/N
LET A3 = A+N/4
LET A2 = 2*PI()*A3/N
LET H = SIN(A2)
LET W = SIN(A1)
IF F > -1 THEN 50
LET W = -W
50 FOR X = 0 TO (N-B) STEP B
LET I = Z + X
LET J = C + I
LET Q = H * V(J) - W * G(J)
LET P = H * G(J) + W * V(J)
LET Y = V(I)
LET S = G(I)
LET V(J) = Y - Q
LET G(J) = S - P
LET V(I) = Y + Q
LET G(I) = S + P
NEXT X
NEXT Z
LET B = B * 2
GOTO 45
60 IF F > -1 THEN 70
LET A = 1/N
FOR L=0 TO N-1
LET V(L) = V(L)*A
LET G(L) = G(L)*A
NEXT L
70 PRINT V(0),V(1),V(2),V(3)
PRINT G(0),G(1),G(2),G(3)
WRITE V "v2.csv"
WRITE G "g2.csv"
PRINT V(0),V(1),V(2),V(3)
PRINT G(0),G(1),G(2),G(3)
END
```

Chapter 8

Project Proposal

Introduction

From the late 70s to the early 90s, the most widely used programming language was BASIC (Beginner's All-purpose Symbolic Instruction Code). A BASIC interpreter was included on the vast majority of consumer microcomputers released during this period. As its name suggests, the language was designed with ease of use in mind – the commands are designed to be human readable and potentially confusing syntax (eg semicolons at ends of lines) was omitted. This was significant at the time as it was the first language with ease of use as a consideration – other languages used at the time (FORTRAN, COBOL etc) were much harder to understand, and hence less accessible to less technical users.

LLVM is a compilation platform written in C++. The principle is that high level languages are compiled into LLVM's bytecode language, which is then run through the LLVM compiler, which can target many different CPU architectures. The advantages of this approach are numerous. It is clearly much easier to write software to target multiple architectures, as once the code is compiled to bytecode the LLVM backend can compile this to machine code for any appropriate architecture. The bytecode is also in Static Single Assignment form. This means the LLVM framework is able to make many compile-time optimisations without needing to perform expensive tree-based operations.

The classic problem with distributing BASIC programs is that it is generally an interpreted language, so developers rely on end-users having a compatible interpreter installed. There does not currently exist a compiler from a BASIC

Listing 8.1

```
BASIC:
x% = 3
x% = x% + 4
x% = x% * 5

LLVM:
%x1 = 3
%tmp1 = 4
%x2 = add i16 %x1, %tmp1
%tmp2 = 5
%x3 = mul i16 %x2, %tmp2
```

standard to LLVM bytecode. This would allow straightforward and optimised compilation from BASIC to any system architecture supported by the LLVM framework.

An example follows of how a trivial BASIC program might be translated into LLVM code:

The project is to implement a fully functional compiler for BASIC. The compiler will output LLVM bytecode which can then be compiled by the LLVM framework to target a large array of end-user architectures. The BASIC dialect I have chosen to implement is ANSI X3.60-1978 (BASIC 78).

I have chosen to use Microsoft's C# language to implement this compiler. This provides a feature-rich, mature platform for development. C# is object-oriented and type safe, and has valuable traits for writing compilers. It is also a language which I am very familiar with, reducing the amount of studying necessary before programming can begin.

Resources Required

- Windows PC – My preferred development environment. I will use my own machine.
- Microsoft Visual Studio – This provides a mature compiler and IDE for C# in which the compiler will be written. Any edition of Visual Studio will suffice.
- Private off-site Git repository – this will provide version control and back-

ups for both the source code and the dissertation.

Starting Point

- I have a working knowledge of compilers and their structure from the IB Compiler Construction course
- I have experience working with both BASIC and C#
- The project will make use of the LLVM compilation framework. An appropriate set of C# bindings for the library will be used.
- A parser generator for C# will be used to produce a skeleton parser as a starting point for the project.

Structure and Substance of the Project

As with the development of any compiler, the project will require the production of modules for lexing, parsing and code generation. The object oriented nature of C# lends itself to this modular way of working.

The first step will be to select an appropriate parser generator. An appropriate context-free grammar to represent the syntax of BASIC must be fed into the generator such that a correct lexer/parser is returned. A set of classes must be constructed that represent all features and constructs representable in the language. This is where the use of an object-oriented language such as C# becomes advantageous, as I can use subclassing and inheritance to produce a meaningful and structured set of classes. Once these classes are designed, the parser will have to be adapted to return the appropriate instance of the classes, forming an abstract syntax tree. A pretty printer would be helpful for debugging at this stage.

Once the classes are designed, the actual code generator must be written. This involves taking the AST and returning the appropriate LLVM bytecode. Appropriate C# bindings for the LLVM library will make this process more straightforward and will optimise the LLVM code as it is generated. This is the phase of development which will take the longest, but a gradual approach to the build is acceptable. I intend to initially implement arithmetic operations and build from there.

Success Criteria

The goal is to produce a working compiler from the ANSI X.360-1978 dialect of BASIC to valid LLVM bytecode written in C#. A selling point for LLVM is compile-time optimisation so it would be interesting to evaluate the performance of programs compiled using my compiler against those compiled using an existing, more conventional BASIC compilation pipeline. The size of the executables produced by each compiler could also be compared.

Timetable and Milestones

3rd-16th November 2014

Read up on LLVM and its bytecode dialect. Write simple programs in LLVM language to explore how the Clang compiler and its optimisation techniques work. Study BASIC 78 in depth and get familiar with all the features that will have to be implemented by the compiler.

17th-30th November 2014

Write some compilers for toy languages in C#. Select an appropriate set of C# bindings for the LLVM library and get familiar with how they work/any shortcomings.

1st-21st December 2014

Further study of languages. Make any necessary changes to the C# bindings for LLVM. Begin to design and implement the classes necessary for representation of BASIC's simple data structures in C#.

22nd December 2014-11th January 2014

Complete the development of necessary classes. Continue to write compilers targeting LLVM in C#.

12th-25th January 2015

Write a context-free grammar for BASIC 78. Use a parser generator to produce an appropriate parser.

26th January-8th February 2015

Implement code generation for simple arithmetic and assignment operations. Implement classes necessary for representing flow control operators.

9th-22nd February 2015

Implement code generation for BASIC input and output, looping and GOTO operators.

23rd February-8th March 2015

Implement additional functionality. At this stage the compiler will be largely complete and optimisation/additional features can be added. Begin writing initial chapters of dissertation.

9th-22nd March 2015

Extensive testing of compiler. Clean up any untidy/inefficient portions of code. Continue to write dissertation.

23rd March-

Send first draft of dissertation to supervisor and continually redraft chapters. Send draft to DoS by 21st April for feedback and complete and submit final dissertation early May.