David Hoare

Robinson College

Computer Science Tripos

Part II Dissertation

The implementation of a compiler from BASIC 78 to LLVM

2015

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

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 compiled by third-party compilers to target one of many instruction sets. The compiler was to be written in C#, and make would use of the LLVM API in the code generation module.

A compiler has been written in C# that correctly and efficiently compiles BASIC code to 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 those generated by similar compilers

No special difficulties

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.

# Table of Contents

Blah

# Introduction

The BASIC language was an important development in the field of Computer Science in the 60s and 70s. It was the first verbose, entry-level programming language aimed at home users, and an interpreter was included on most of the personal computers released at this time, thus introducing a generation to the basics of programming. Although not as widely distributed BASIC is still often used as an introductory programming language for beginners. It has also influenced many modern languages, most notably Visual Basic .NET, one of the two primary languages targeting the .NET framework.

LLVM is a framework for compiler construction. The principle is that compilers built with the LLVM framework output LLVM Intermediate Representation, a bytecode assembly language. This bytecode can then be compiled by an LLVM backend to native code for a target architecture. Backends exist for targeting all major modern instruction sets. Hence building a compiler with LLVM is an easy way of enabling it to build for multiple platforms. Moreover the LLVM framework performs code optimisation passes at every stage of compilation. This is made easier by the Static Single Assignment form of LLVM bytecode. 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 an LLVM frontend (ie compiler targeting LLVM IR) for the original BASIC programming language. The ANSI X.60-1978 standard was selected as an appropriate starting point for the definition of the language, with new features being added as necessary to update the language.

# Preparation

A language needed to be selected with which to implement the compiler. I chose C# .NET for this purpose because of my personal experience using the language and also its object-oriented design. Looking at the 1978 BASIC standard made it clear that it would be very advantageous to the development of the parser and compiler for object-oriented language features such as inheritance and overloading to be available.

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 tedious task of building a lexer from scratch I elected to use a lexer generator to produce this component. After exploring available tools I selected the ANTLR 4.5 parser generator for this task. ANTLR takes a standard context-free grammar and has a well-documented C# target. The system is mature and well supported and was a good choice.

The code generation module of the compiler involved use of the LLVM API. A complication to this step was the fact that there do not exist a complete set of bindings of the (very extensive) API for .NET development. There do exist several attempts at writing said bindings 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 incomplete bindings written by my project supervisor Michael Gale some years ago. With Michael’s assistance I was able to add to the bindings all functionality I needed from the LLVM API when necessary.

# Implementation

## Parser

I based a context-free grammar for my parser on the ANSI X3.60-1978 standard for BASIC. 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. 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 arsing. Once the grammar correctly interpreted the language I used ANTLR to produce C# class files that could be used to integrate the parser with the compiler

## Integrating the Parser

The next task was to use the stream of tokens returned by the parser to form an abstract syntax tree. To aid with this task I implemented the BASICListener interface produced by the ANTLR compiler. This provided stubs for all methods required for the listener. I chose an iterative approach for the development of the compiler. Hence 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.

BASIC is a very 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. The lexer is passed a single line at a time, and the base structure in the AST is the Line class. This class is inherited by classes for all different line types (Line\_Goto, Line\_For, Line\_If etc).

## Code Generation tools

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

### 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, I was able to use a single LLVM module containing a single main() function. All 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 these are stored as public static members in the Parser class so they can be accessed as necessary.

Within the main function are basic blocks which are represented by the LLVM.BasicBlock class. These contain a block of code, and are represented in LLVM IR by a labelled section of bytecode. To insert code into a BasicBlock I made use of the LLVM.IRBuilder class. This provides methods such as CreateAlloca, CreateFAdd, CreateBr to easily insert IR instructions into a BasicBlock.

### 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.

## Types

### 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. This practice is hidden from the user by use of the “%g” format specifier when outputting numbers with PRINT or WRITE. This outputs the number using the shortest possible representation, so the double 4.00000000… is output as 4, giving the illusion that it has been treated as an integer internally.

### 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 this as static members in the Parser class as an 8-bit integer, a 32-bit integer and a double.

## Expressions

### 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.

### Numeric expressions

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

numericexpression : sign? term (sign term)\*;

term : factor (multiplier factor)\*;

factor : primary (CIRCUMFLEXACCENT primary)\*;

multiplier : ASTERISK | SOLIDUS;

primary : numericvariable | numericconstant | numericfunctionref | LEFTPARENTHESIS numericexpression RIGHTPARENTHESIS;

This structure allows complex recursive numeric expressions to be parsed correctly in accordance with BIDMAS rules. As an example of how expressions parse, see the parse tree on the following page. This was exported by the ANTLR4 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. ie 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:

%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

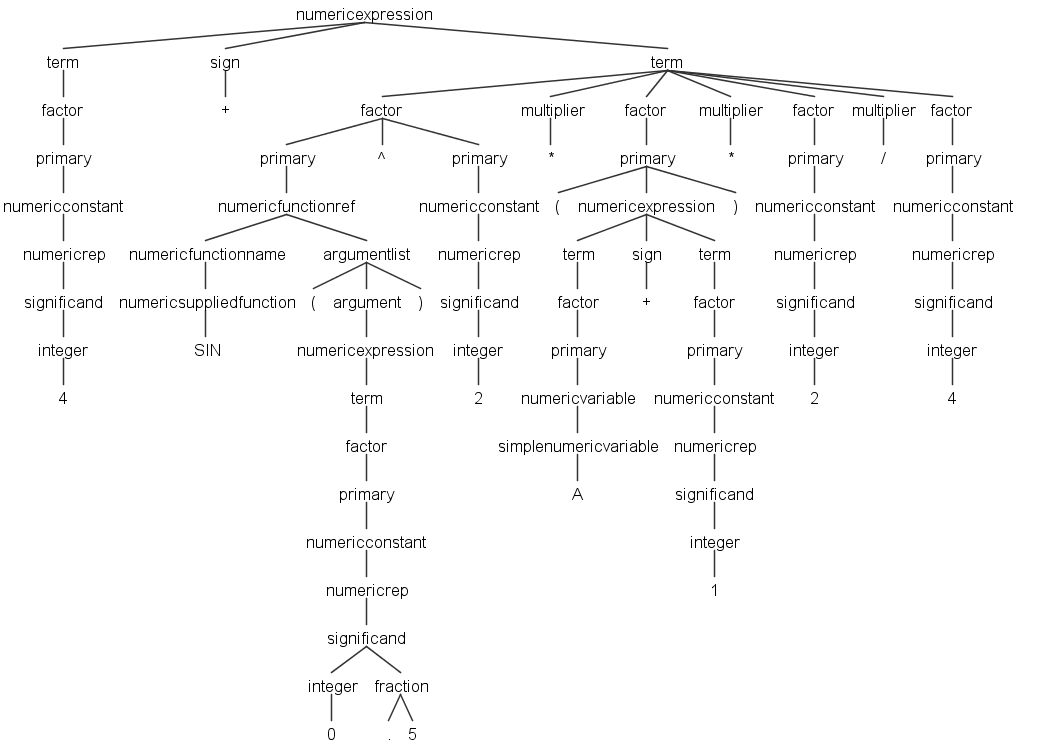
%multmp = fmul double %3, %addtmp

%multmp1 = fmul double %multmp, 2.000000e+00

%divtmp = fdiv double %multmp1, 4.000000e+00

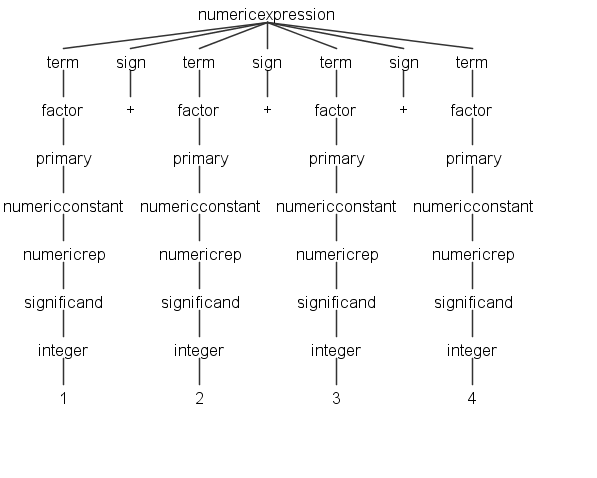
%addtmp2 = fadd double 4.000000e+00, %divtmp

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.



Parse tree for 4+SIN(0.5)^2\*(A+1)\*2/4

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 below parse tree.



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

%A = alloca double

store double 1.000000e+01, double\* %A

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.

## Compiler passes

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

### 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 AST structures used within the line to compile these structures. After each line is processed, a reference to the corresponding Line 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]

### 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 Line class and inserts a br 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 Line that represent lines where this jump is not necessary, ie lines containing control-flow statements. For example, the override jumpToNext() method for Line\_Goto is empty, as execution should not pass to the next line after a GOTO statement.

The jumpToNext() method for Line\_Next does not create a jump to the next line, but instead stores the next line in a member variable. This reference will be needed to construct the conditional jump created in the third pass. Similarly the jumpToNext() method for Line\_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.

### Third pass

The final compiler pass calls the processGoto() method for each line. The body of this method in Line 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 Line\_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.

## Error Handling

### Exceptions

I have defined a new System.Exception which I have named CompileException. I use this to represent handled runtime errors, ie errors where the compiler knows what has gone wrong. These exceptions can then be differentiated from other System.Exceptions in catch blocks to provide differing 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.

### 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 the compiler has been executed with the –debug flag set. This error information includes the line number where the error was encountered, 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 line number.

## VariableStore

### 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].

### 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 amount of memory that 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 which can be used to initialise the array. A pointer 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 11 items.

## Numeric functions

### 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).

### 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].

## File IO

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 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:

READ A filename

WRITE A filename

where A is a pre-defined array and filename is a string expression containing the file 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, 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.

## 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. Detailed 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 [3.11 ?]

## Further compilation and assembly

### 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.

### 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 is passed to gcc (the GNU Compiler Collection). This is a versatile open-source compiler. Here it is used simply to assemble and link the assembly generated by llc and produce a .exe 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]

### 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.

The target architecture is specified to llc by use of a string descriptor known as a target triple. A target triple takes the format <architecture>-<vendor>-<system>. Examples of architecture include x86,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.

### FFT for Raspberry Pi

As an example of this process, I compiled my test program for Fast Fourier Transform to run on a Raspberry Pi. I ran the source code through my compiler to generate fft.ll. I then ran this through llc using the argument –mtriple=opiqwehpfoihqwpoefi (the target triple for a Raspberry Pi. This produced fft.s which I transferred to the Raspberry Pi then ran through gcc. It was necessary at this stage to pass the –abi=blah argument to gcc. This forces the use of hard-float ABIs, a feature of the architecture.

# 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.exe tool. This is an interpreter which works directly on the LLVM IR bytecode produced by my compiler, running it in real-time 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.

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 discrete Fourier transform of a sampled signal, ie 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 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.

# Conclusions

Blah

# Bibliography

Blah

# Appendices

Blah

# Project Proposal