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 BASIC. The compiler was to output LLVM bytecode which can then be compiled by third-party compilers to target one of many computer architectures. More specifically the language implemented was to be ANSI X3.60-1978. The compiler was to be written in C#, making use of the language’s object-oriented features to simplify parser development . The compiler was to make use of a parser generator to produce the skeleton of a parser and 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 modernise the language and 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

<any 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 Microsoft Visual Basic .NET, one of the two primary languages targeting Microsoft’s .NET framework.

LLVM is a modern compiler construction infrastructure. 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 advanced 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 Microsoft 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 lexer 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.

# Implementation

## Lexing

I used the ANSI X.60-1978 standard to write a complete context-free grammar for BASIC. To debug the CFG I used ANTLR 4.5’s TestRig component, which produces a graphical representation of how an input string is parsed by the grammar. Once the grammar correctly interpreted the language I used ANTLR to produce C# class files that could be used to incorporate the lexer component into the compiler.

## Parsing

I then built a parser to produce an abstract syntax tree from the token stream returned by the lexer. To aid with this task I implemented the BASICListener interface produced by the ANTLR compiler. This provided stubs for all methods required by the parser. I chose an iterative approach for the development of the compiler. Hence I initially implemented 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 Gen

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.

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 in every code() call these are stored as static members in the Parser class so they can be accessed as necessary. Similarly, code generation methods make use of LLVM types. 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.

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 debugging the BASIC code.

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].

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.

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.

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).

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].

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.

When initialising the ANTLR BASICParser class used to lex and 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 lexer line-by-line this number is always 1. ANTLRErrorListener also addresses this problem by instead outputting Parser.currentLine, which contains the correct line number.

# 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. 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