An ECMA-55 **`Minimal BASIC`** Compiler Targeting x86-64 Linux

by

John Gatewood Ham

Slides last updated on November 19, 2017.

Motivation

- BASIC was designed for teaching
- BASIC behavior is very close to real CPU behavior
- No free BASIC for 64bit Linux that accepts the Minimal BASIC dialect existed. Now Jorge Giner Cordero's excellent bas55 interpreter also exists.
- BASIC was designed to be compiled, although on microcomputers most people used interpreters
- Most production compilers are so complex only a genius student can understand them; a simpler compiler is needed to teach people about compilers in a first course
- Most people today use JIT and bytecode – the art of actually compiling all the way down to assembly is becoming a lost skill and is a problem for operating system and compiler development in the future

BASIC's Tarnished Reputation

Today, **BASIC** has a bad reputation which is largely due to *Dijkstra*'s famous criticisms. Those unfair criticisms, combined with the vendors of microcomputers no longer including **BASIC** for free, led to the end of the language popularity.

In computers with 64KB or less of total RAM, making everything global and coding in a machine-code style makes sense. **GOTO** is not intrinsically evil – today's current CPUs all use unconditional branches, and every C and C++ compiler emits those branches.

ECMA-55 Minimal BASIC Numeric Integration Left Riemann Sum

This is a fairly typical type of program that would be written in **BASIC** for a freshman calculus or numerical analysis course.

```
10 REM     NUMERIC INTEGRATION
20 DEF FNF(X)=COS(X)
30 LET A=0
40 LET B=1
50 LET N=100000
70 LET D=(B-A)/N
80 LET S=0
90 FOR I=A TO B STEP D
100 LET S=S+D*FNF(I)
110 NEXT I
120 PRINT S,SIN(B)-SIN(A)
130 END
```

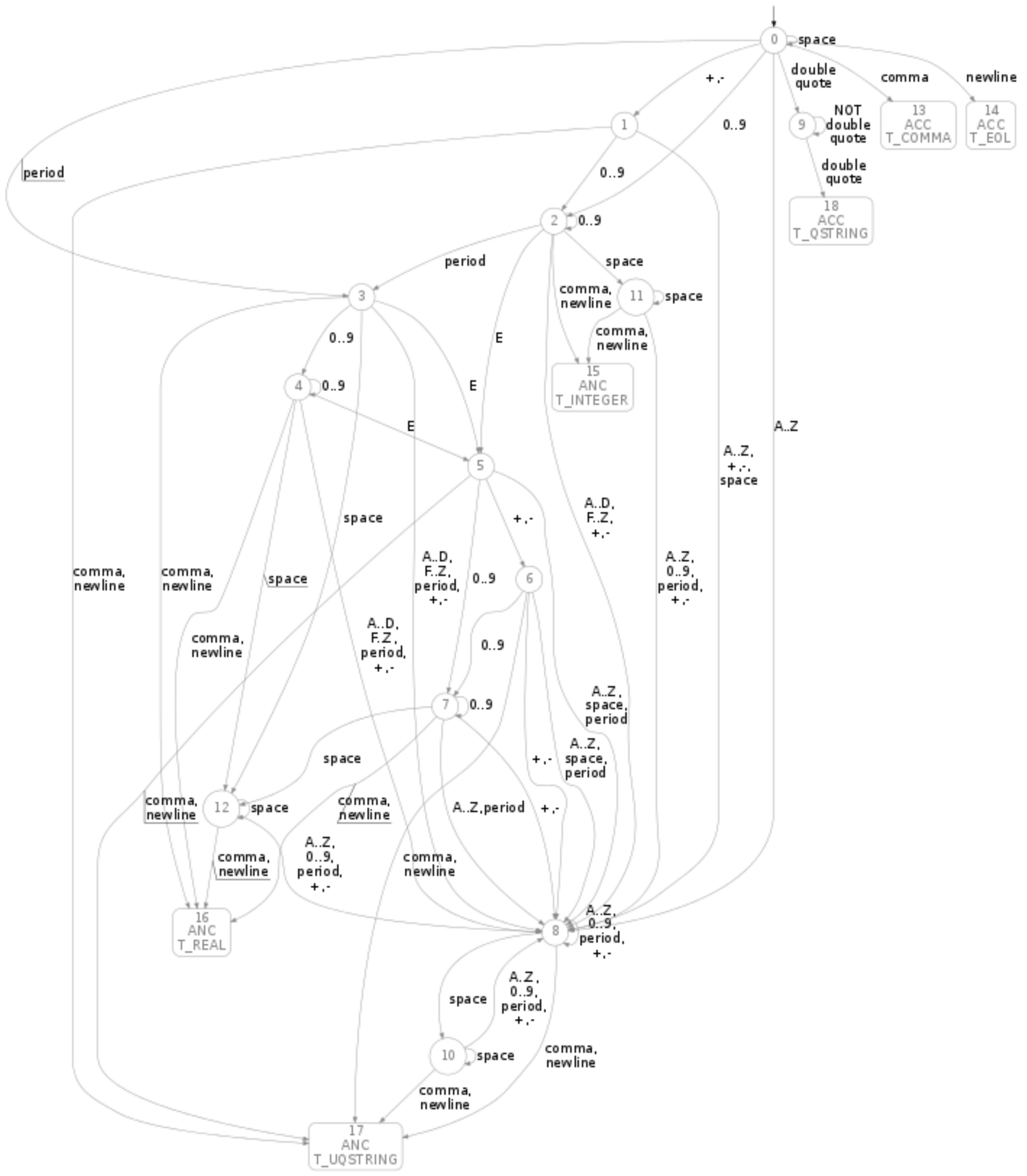
This is the code generated for line 100 of the previous example program.

```
.LLINE0100:
        movabsq $.LCURLINENO, %rax
        movq    $100, (%rax)
        floatmem_to_floatreg I,%xmm0,'I'
        movabsq $FNF,%rax
        callq   *%rax
        floatmem_to_floatreg D,%xmm1,'D'
        binary_multiply %xmm1,%xmm0
        floatmem_to_floatreg S,%xmm0,'S'
        binary_add %xmm0,%xmm1
        floatreg_to_floatmem %xmm0,S
```

Actually it is quite involved at the machine level, and most of the complexity is for error handling. **C** does not support exception checked arithmetic at all. This is the reason why the standard advice to "*just look at the output of* **gcc** *or* **clang**" was not very helpful when learning to generate arithmetic expression code for this compiler.

The use of macros makes the code easier to read, yet imposes no penalty at runtime. This allows putting the ugly complexity of exception handling for overflow, underflow, etc. in the macros and keeping the main code much simpler to follow.

Very complex rules for **`READ`** and **`INPUT`** require rather a correspondingly complex finite state machine.



Implementation

- Hand-coded deterministic finite state machine scanner
- Hand-coded top-down, recursive descent parser
- Pre-loaded symbol table with binary search lookups
- Abstract Syntax Tree intermediate representation generated by parser
- Semantic analyzer using AST
- Arithmetic expression constant folding pass using AST
- Code generated from AST
- Written in standard **C99** and GNU **as** (*AT&T dialect*)
- Scalar SSE2 floating point math with full exception checking

  (optionally scalar AVX floating point can be used)

Implementation II

- Now includes bottom-up tree pass as an alternative to top-down code generation pass.
- Now includes dag generation for common sub-expression elimination for arithmetic expressions.
- Assembly code is now in separate flat files to make it easy to read and modify.
- Now scanner mmap()'s the BASIC source file and avoids making copies of the buffer.

The Runtime Library

- Naoki Shibata's *SLEEF* for elementary functions
- Bob Jenkins' *ISAAC-64* for **`RND`** and **`RANDOMIZE`**
- David Gay's *dtoa*, *g_fmt*, and *strtod* functions for conversion between ASCII and floating point
- **`INPUT`** subsystem
- **`PRINT`**/**`TAB`** support

Arithmetic Expressions

- SIMD math used since Intel claims x87 support is deprecated and going away. This means generated code requires a CPU with SSE2 support. All known 64bit AMD64/x86-64 CPUs except Xeon Phi have the necessary features, but runtime checks for them are included in the generated programs just in case.
- Full exception support required by the standard implemented using assembly macros to keep generated code easy to read.
- Constant folding in arithmetic expressions is implemented, preserving required exceptions.

Minimal **BASIC** Control Flow

- **GOTO** for unconditional branch
- **IF** with a line number target (but no **ELSE**) for conditional branch
- **FOR** loops with index variable and **NEXT** are the only looping construct, with optional **STEP**.
- Rules to prevent jumping into a loop are complex to enforce but a scope concept makes detecting many error cases possible.
- **ON** expr **GOTO** line,line,... for multi-way branch using

  a jump table.

Compiler Assembly Output

- AT&T *UNIX*-style assembly code operands are backwards from Intel documentation, and very little easily useful example code exists on the Internet.
- Uses ***large*** model for simplicity.
- Uses macros for arithmetic operators.
- Linux AMD64 ABI requires register-based parameter passing.

Generated Executables

- The generated assembly is assembled with as, linked with ld, and a static executable is generated.
- Console I/O calls the kernel directly, so no C standard library (libc) is required by the generated code.
- *SLEEF* and *ISAAC-64* are included in the executable, so no C standard library (libm) is required by the generated code.
- Good style for stand-alone environments.

Automated Testing

- A test harness written in *GNU make* and *GNU bash* shell using standard *UNIX* tools.
- Verifies compile output of every test
- Verifies runtime output of every test that can be run
- Supports programs with different 32/64 bit output
- Source for all 208 *NBS* tests is included
- 207 of 208 tests pass as of 2014/04/02, but test #131 cannot be automatically tested since it uses **`RANDOMIZE`**
- More than 60 additional tests

Features

- -P option to pretty-print programs
- -R option to renumber programs (implies -P)
- -X option to enable extensions
- -w option to enable full double precision output and 132 column output mode (wide mode)
- Fully implements recommended uninitialized variable detection
- Generates fatal exception with line number at runtime if a NaN is generated.

Extensions

- Short-circuiting **AND**, **OR**, and **NOT** for conditional expressions.
- **EXIT FOR** statement for breaking out of a **FOR** loop early.
- Lower-case character support in quoted strings and **REM** statements.
- Add support for < <= >= > to string comparison.
- Add **LEN** function to get string length.

Future Work

- Add *DWARF* debugging support
- Support an alternative output style that uses the more common RIP-relative addressing and the small code model
- Use AVX math instructions with the 3 operand style
- Add better string support, file handling, etc.
- Optimization!

Benefits

- Students can program in traditional `BASIC` to get a better understanding of early computer programming and how a CPU works.
- Much simpler than something like Java or C++ for a ***first*** exposure to programming.
- People who want to implement true compilers (all the way to assembly) for procedural, iterative languages can start easily with this compiler.
- Simple overall structure, easy to understand, small (less than 36,000 total lines of code, and much of that is runtime library).

Benefits II

- People who want to know how to code AMD64/x86-64 assembly, including floating point exception handling, can look at the simple output of this compiler and learn from it.
- Reasonable teaching compiler – people can add or modify statements, runtime library functions, long variable names, or target another processor.
- Uses an abstract syntax tree to separate parsing, semantic analysis, and code generation.
- Written in standard C99; <u>no exotic languages (lisp, etc.) required</u>.
- Small code base, 100% FOSS.

Conclusion

Today many people get into computer science but never learn how the CPU works. This results in a shortage of people who can do *low-level* programming which is required to generate compilers, work on operating systems, and to achieve good performance that takes full advantage of hardware. It also makes learning how to debug things like a bad compiler or linker a lot more difficult since people without this knowledge cannot read assembly at all, and have no idea what correct output should be.

Conclusion II

The decline of students learning about implementing compilers is largely due to the impossibly steep learning curve required to get involved with today's production-level open source compilers, the poor documentation of non-trivial AMD64/x86-64 floating point assembly programming, and the fact that the literature available for low-level programming on x86 architecture is almost exclusively for 32bit and integers. Hopefully this project will help more people get involved in creating *low-level* code for x86-64 so they can help create the next generation of programmer tools and operating systems.

Resources

Compiler overview page:

`http://buraphakit.sourceforge.net/BASIC.shtml`

SourceForge project page:

`http://sourceforge.net/projects/buraphakit`

The bas55 project page:

`http://jorgicor.sdfeu.org/bas55/`

Some of Edsger W. Dijkstra's famous **`BASIC`** criticisms:

`https://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD498.html`

EWD claims BASIC causes “*mental mutilation*”

`https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD898.html`

EWD claims “*teaching of BASIC should be rated as a criminal offense: it mutilates the mind beyond recovery*”