An ECMA-55 **Minimal BASIC** Compiler Targetting x86-64 Linux

by

John Gatewood Ham

Slides last updated on April 2, 2014.

Motivation

- `BASIC` was designed for teaching
- `BASIC` behavior is very close to real CPU behavior
- No standards-conformant free `BASIC` for 64bit Linux existed
- `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.

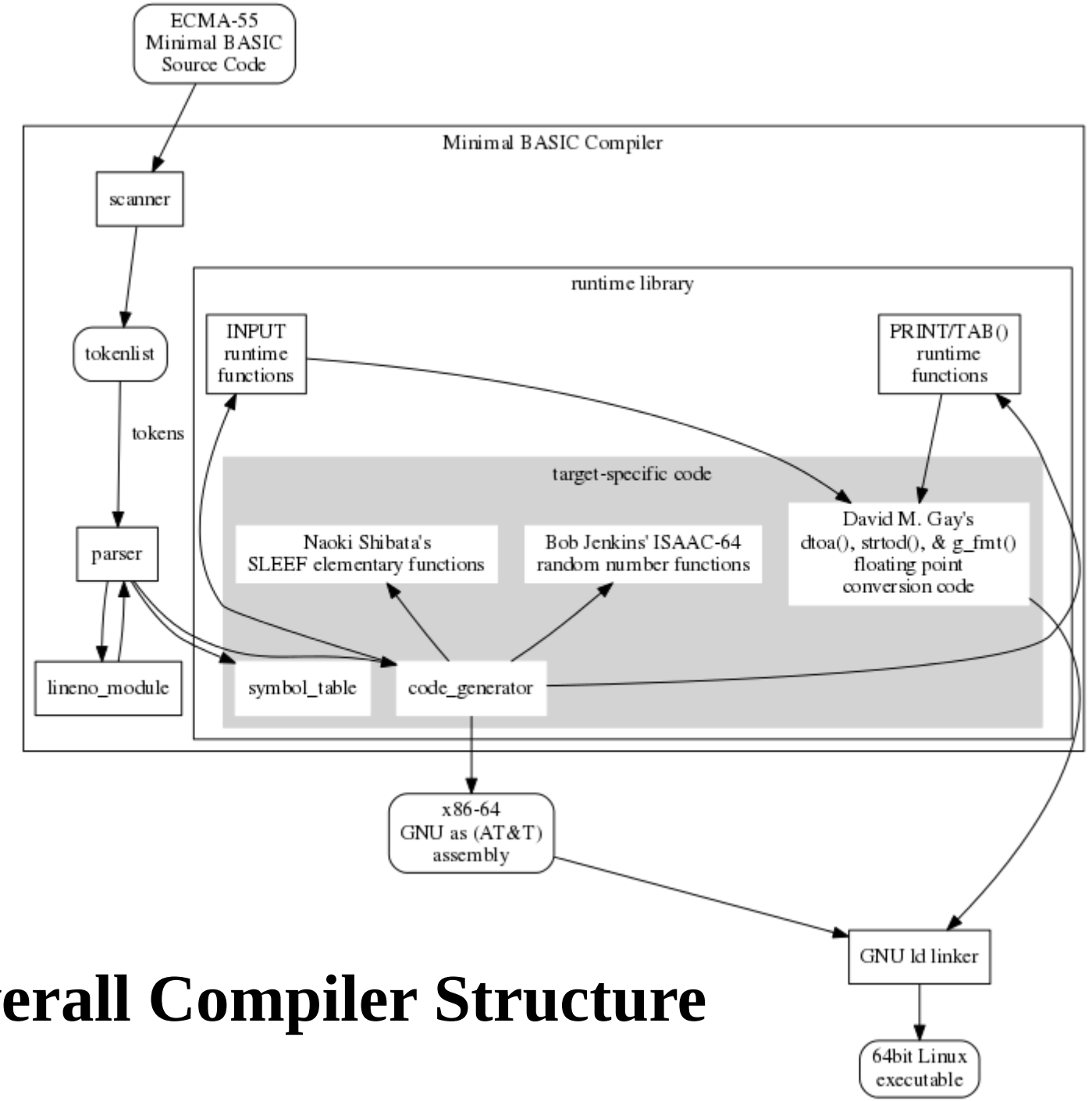
ECMA-55 Minimal BASIC
Numeric Integration
Left Reimann Sum

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
```

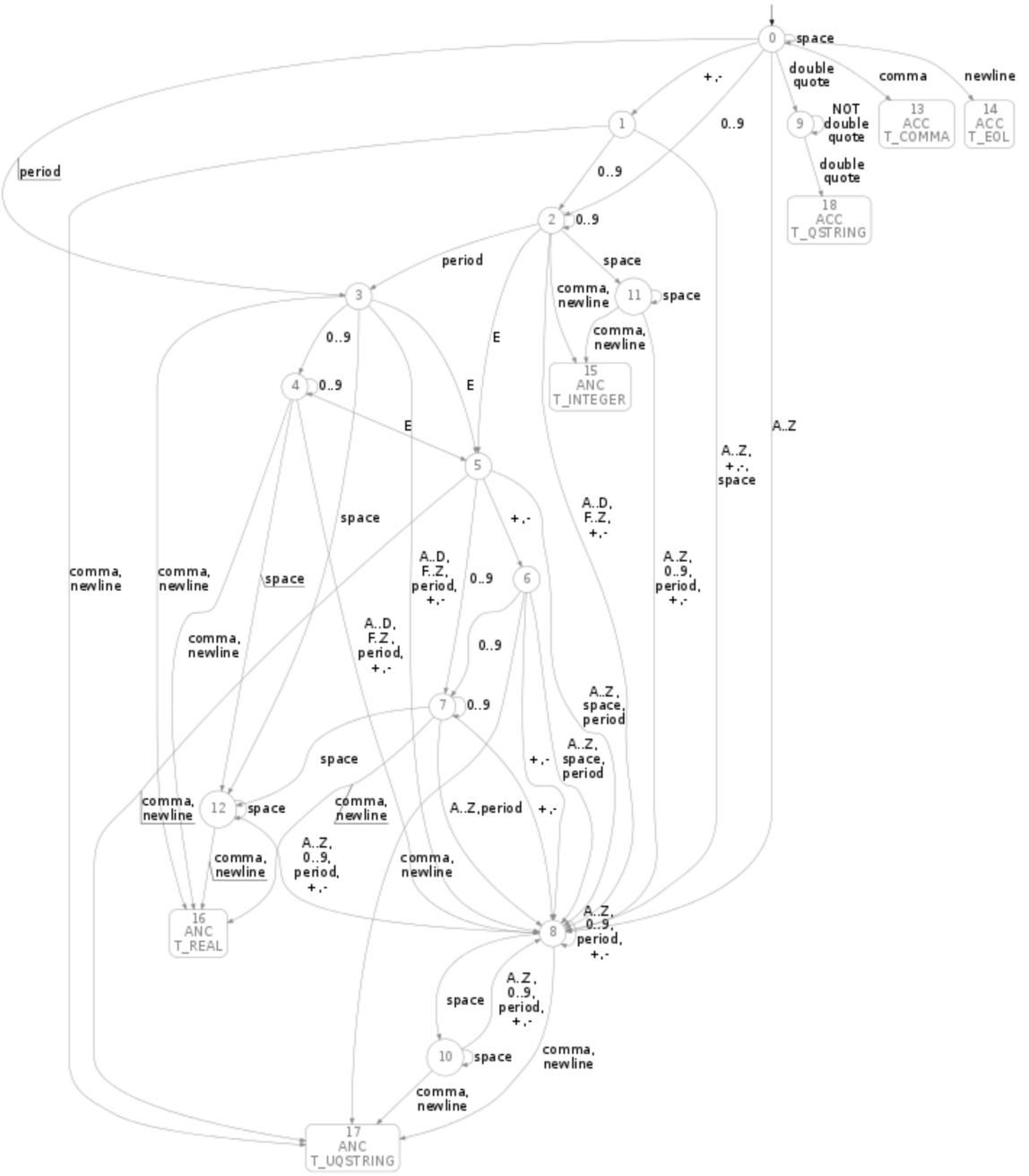
This is the beginning of the code generated for line 100 of the previous example program. It is quite involved at the machine level, and most of it is for error handling. This is typical of code compiled with full error checking, but **c** does not support this style at all. This is the reason why the standard advice to "*just look at the output of gcc or clang*" was not very helpful when developing this compiler.

```
        movabsq $S, %r15
        movsd   (%r15), %xmm0
        comisd  %xmm0, %xmm0
        jnp     1f
        # referenced scalar S was NaN, uninitialized
        movabsq $.Luninitialized_msg, %rsi
        movabsq $printstring, %rax
        callq   *%rax
        # must create string in reverse order
        xorq    %rdi,%rdi
        movb    $'S',%dil
        shl     $8,%rdi
        movb    $32,%dil
        movabsq $printvarname,%rax
        callq   *%rax
        movabsq $badxit, %rax
        jmpq    *%rax
1:      # OK
        pushxmm 0
        movabsq $D, %r15
        movsd   (%r15), %xmm0
        comisd  %xmm0, %xmm0
        jnp     1f
        # referenced scalar D was NaN, uninitialized
        movabsq $.Luninitialized_msg, %rsi
        movabsq $printstring, %rax
        callq   *%rax
```



Overall Compiler Structure

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
- Parser driven, output to temporary files
- No intermediate representation
- Almost no optimization (simple peephole pass only)
- Written in standard **`C99`** and GNU **`as`** (*AT&T dialect*)
- Uses a dedicated operand stack to implement arithmetic expressions

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

- SSE/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 have the necessary features.
- While SSE is register-based, this compiler uses an operand stack in memory to avoid having to do complex register allocation
- Full exception support required by the standard, implemented using assembly macros to keep generated code easy to read

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
- **ON** expr **GOTO** line,line,... for multi-way branch

Compiler Assembly Output

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

Generated Executables

- The generated assembly is linked with David Gay's floating point I/O code and a static executable is generated.
- Console I/O calls the kernel directly, so no libc is required by the generated code.
- *SLEEF* and *ISAAC-64* are included in the executable, so no libm is required by the generated code.
- Good style for stand-alone environments.

Automated Testing

- A simple test harness written in *GNU bash* shell using standard *UNIX* tools
- Verifies compile output of every test with standard '`make check`' invocation (64bit) and '`make check32`' (32bit)
- Verifies runtime output of every test that can be run
- Supports programs with different 32/64 bit output
- Source for all 208 *NBS* tests are provided
- 207 of 208 tests pass as of 2014/04/02, but test #131 cannot be automatically tested since it uses **`RANDOMIZE`**

Future Work

- Rewrite arithmetic expression handling to be register-based, adding the required intermediate representation and register allocation
- Add *DWARF* debugging support
- Support an alternative output style that uses RIP-relative addressing and the small code model
- Use AVX math to simplify arithmetic expression handling with the 3 operand RISC style (*requires Haswell CPU – my faculty insists I keep using a 2007 Core 2 Duo*).
- Add better string support, file handling, etc. in a move towards ECMA-116 **`BASIC-1`**.

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 20,000 total lines of code, about half of that is runtime library).

Benefits II

- People who want to know how to code x86-64 assembly, including floating point exception handling, can look at the simple, non-optimized output of this compiler and learn from it.
- Reasonable teaching compiler – people can add statements, runtime library functions, long variable names, or retarget to another processor. No "*OOP for OOP's sake*" is used.
- Written in standard C99; no exotic languages required.

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 x86-64 assembly programming, and the fact that the literature available for low-level programming on x86 architecture is almost exclusively for 32bit. 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 (with tarball downloads and mercurial VCS access):

http://sourceforge.net/projects/buraphakit

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 offence: it mutilates the mind beyond recovery*"