The Return of Tiny Basic

An examination of the language that started it all

TOM PITTMAN

In January 1976, *Dr. Dobb's Journal of Tiny Basic Calisthenics and Orthodontia, Running Light without Overbyte* was launched on the popularity of the article "Build Your Own Basic," originally written anonymously by Dennis Allison. He was the "D" of "Dobbs" in the new magazine name. The "B" was Bob Albrecht, who ran a store-front, walk-in, computer time-share service called the "People's Computer Company" and who published a newsprint tabloid with the same moniker (*PCC*). The BYOB article first appeared in *PCC* after people began to realize that this $400 computer kit they bought was too hard to program in machine language, and Bill Gates's Altair Basic was considered too expensive at $150.

I was not the first implementor. Dick Whipple and John Arnold were there ahead of me. But while there were a number of Tiny Basic projects well underway before I started, mine was distinctive in several ways. For instance, the others all ran on Intel CPUs (mostly 8080), I only did other chips. They were all free, but I charged $5 up front. Many people fondly remember using my Tiny Basic on their 8080 or Z80 when it was most likely actually Lee Chen Wang's Palo Alto Tiny Basic that they were using.

Another significant difference is that my Tiny Basic was the only one to use Allison's original IL code (available electronically; see "Resource Center," page 4) pretty much as defined. There were several bugs I had to correct, and I added some opcodes to support a couple of Tiny Basic functions, but the BYOB interpreter was written in a special-purpose pseudocode, which Dennis recommended interpreting. I knew about interpreters, having done several of them over the years. Everybody else hand-translated the pseudocode to assembly language. So, as far as I know, my Tiny Basic was the only one with a two-level interpreter.

This technology is still interesting and useful today. In this article, I examine what's involved in recreating my original Tiny Basic in today's universal assembly language—ANSI Standard C.

The first issue of *DDJ* (which is available electronically as a PDF e-zine; see "Resource Center," page 4), reproduced the original concept article and three follow-up articles from the *PCC*, showing the progression in Allison's thinking. It's a valuable study in the formulation of a sophisticated software solution to an understandable problem. I don't have space here to recreate all the thinking he alludes to, but I do touch on some of the technical issues and trade-offs of implementing pseudocode interpreters.

Tom is a consultant and can be contacted at tpittman@ittybittycomputers.com.

Virtual Programming Languages

My introduction to virtual machines (VM) came as an off-hand remark at my first full-time job. Drexel Heater pointed out that regularly using my own library of Fortran subroutines amounted to designing my own programming language. Although never articulated so clearly, this idea pervades the programming community even today, particularly in C++ with the Standard Template Library, which by operator overloading actually changes the meaning of the Standard C operators. This concept gives you ultimate control over software design. Where Alan Kay predicts the future by inventing it, we predict software behavior by inventing the computer it runs on.

"Every programming language is the machine language of some abstract computer"

Every programming language is the machine language of some abstract computer whose machine operations are exactly that language's primitives. If you remove an operator from the language by not using it, you have effectively changed (limited) the computer. But let's increase, not reduce, the power of the machine. We do this by making the operations do more. That was Drexel's point about the subroutines.

C is more powerful than binary absolute, not because you can do more (you can't), but because the same programmer effort accomplishes more. A simple assignment—four keystrokes—hides a great deal of work the compiler does to allocate the variable: It chooses a representation of the value as well as the hardware instructions to copy those bits into that memory location. All this work is necessary, but not part of the problem the programmer is solving. Correspondingly, those four keystrokes are necessary, but not part of the specified program requirements. The most powerful programming language is where you state the requirements in a task-specific language, then the computer does it. That is the programming language you should write in.

Dennis Allison's (and my) Tiny Basic gets close to a direct implementation of a task-specific programming language.

Grammar as Programming Language

Allison encouraged his collaborators to read a compiler book. At the risk of appearing self serving, I repeat the advice. *The Art of Compiler Design: Theory and Practice*, which I cowrote with James Peters (Prentice Hall, 1991; ISBN 0130481904), emphasizes the context-free grammar as the primary design tool for the entire grammar. (You can download a free copy of my self-compiling compiler-compiler, written entirely in a grammar, at http://www.IttyBittyComputers.com/IttyBitty/TAGC/TAGinfo.html). I learned this emphasis from Allison's Tiny Basic. The important point is that the grammar is a compact notation for expressing a program as a sequence of statements, and each statement is (in the case of Tiny Basic) a keyword followed by other specified parts, and so on, as in Figure 1, the original Tiny Basic grammar. The grammar is itself written in a precise formal language. The grammar is a program; all that remains is to compile (translate) the grammar-program into machine language and run it.

Well, not quite. The grammar only specifies syntax. You also need to specify semantics, what to do when the grammatical (correct) program is accepted and run. I'll expand on the grammar somewhat, by means of English pseudocode: Tiny Basic reads lines. Each line is a statement, possibly with a line number in front (first grammar rule). If it has a line number, insert the numbered statement into the stored program; otherwise just do whatever the statement is (semantics). Repeat indefinitely.

A statement is one of the following 11 things: The keyword *PRINT* followed by a list of expressions to print, which are evaluated one by one, printed, then spaced over to the next "zone"; or else it is the keyword *IF* followed by an expression, which is evaluated, then a relational operator and another expression that is also evaluated and compared to the first expression value, followed by the keyword *THEN* and then finally another statement, which is executed if the comparison is True; or else a statement is the keyword *GOTO* followed by a line number expression, which is evaluated and becomes the next line to execute; and so on.

Similarly, an expression is a term, possibly preceded by a sign, then followed by any number of additional terms each preceded by + or –. Evaluate each term, then add or subtract each successive term. The result is the expression value.

A term is a factor, followed by any number of additional factors, each preceded by a * or /. You get the idea.

Now you have specified the whole interpreter. That is the program, more complete than the grammar alone because it also says how to do the computation. But still, even less than the grammar, it is not understandable by 1975 microcomputers. Even now, 30 years later, only people understand English—and not all that well. So let's formalize this a little and make it more like a conventional programming language.

For "Tiny Basic reads lines" I use the programming command *GETLINE* to read each line. Still buried in there are all the mechanics of accepting characters into a line buffer, stopping when users press the Enter key, and preventing buffer overrun, details that do not concern us at this level.

For "possibly with a line number in front," I use the command/test *TSTL* and the label of that part of the program to handle the line if there is no number. The English explanation: To "insert the numbered statement into the stored program," I spell *INSERT*. The English word "insert" could mean any number of things in other contexts, none of them relevant to interpreting a Tiny Basic program. Thus, this single word is sufficient to explain the whole intent of this operation. Notice also that the grammar says nothing about insertion; it is semantics—what to do with this line now that you know it has a line number.

For "Repeat indefinitely" (not in Allison's grammar, although it should be there as a star), you can write *JMP* with the name of the line where we started. That starts to look like a real programming language. That's exactly what it is: the assembly language for a VM that interprets Tiny Basic. Table 1 presents a partial correlation between grammar, English pseudocode, and IL, both Allison's original in *DDJ* #1 and my own subsequent TBIL. Listing One shows the resulting IL from *DDJ* #1, with some of the errors corrected. This program in IL will execute a Tiny Basic statement. The operators *TST, TSTV, TSTN,* and *PRS* all use a cursor to find characters of the Tiny Basic line. Other operations (*NXT, XPER*) move the cursor so it points to a Tiny Basic line. I corrected a few obvious errors. TBILasm is an assembler written in Tiny Basic. TBILasm and my extensions showing the hexadecimal byte codes are available as part of the Tiny Basic Experimenter's Kit (TBEK.txt), available at http://www.IttyBittyComputers.com/IttyBitty/TinyBasic/TBEK.txt.)

Just as the particular words of the English-language pseudocode depend on who is

(continued on page 28)

```
line::= number statement ⓒⓡ | statement ⓒⓡ
statement::=  PRINT expr-list
              IF expression relop expression THEN statement
              GOTO expression
              INPUT var-list
              LET var = expression
              GOSUB expression
              RETURN
              CLEAR
              LIST
              RUN
              END
expr-list::= (string | expression) (, (string | expression) )*
var-list::= var (, var)*
expression::= (+ | - | ∈) term ((+ | -) term)*
term::= factor ((* | /) factor)*
factor::= var | number | (expression)
var::= A | B | C ..., | Y | Z
number::= digit digit*
digit::= 0 | 1 | 2  | ...  | 8 | 9
relop::= < (> | = | ∈) | > (< | = | ∈) | =
```

Figure 1: *Tiny Basic grammar. The things in bold stand for themselves. The names in lowercase represent classes of things. "::=" is read "is defined as." The asterisk (*) denotes zero or more occurrences of the object to its immediate left. Parenthesis equal group objects.* e *is the empty set.* | *denotes the alternative (the exclusive-or).*

(continued from page 25)
saying it and how they happen to think of the problem at that moment, the particular expression of the IL is also idiosyncratic. You can see that in the difference between Allison's IL and mine. Some of it is just spelling differences (to simplify my assembler) and added features. However, there are also some differences in what the words actually do, particularly in startup and line advance, where Allison was initially somewhat vague. No two independently written programs will ever exactly match, even if they perform identically.

The point of this exercise is to express all of the relevant program operation in a concise form. In Tiny Basic, a large part of the program specification is already in a concise grammar, but we needed to add semantics. Other projects would have different requirements.

Pseudocode Trade-Offs

One of my objectives in doing this C Tiny Basic project was to investigate the mechanics for putting a graphical user interface (GUI) on a Windows program. Three years ago, when Apple killed the only commercially viable WYSIWYG operating system that ever existed, I threw away 67,000 lines of active Mac-only code. I can rewrite that program, but I need to get away from proprietary platforms the vendor can arbitrarily discontinue. C/C++ is an ANSI-standard multiplatform programming language, but there is no standard GUI. So I chose to build a virtual GUI, emulated on whatever platform is available, and implemented interpretively (in an IL) like Tiny Basic. Tiny Basic is thus a good testbed for it.

There is a performance cost to interpretation. It depends on the implementation language and the IL design, but typically costs three to 10 machine instructions or lines of source-code overhead on each VM operation, besides the code to do the operation. Very high-level operations such as the TBIL or my GUI emulator spend a lot of time doing each operation, so the overhead is a small penalty. A low-level VM such as Java bytecode, which does much less in each operation, typically runs an order of magnitude slower than native code to do the same job. Because the VM is relatively fixed and widely used, the cost of implementing a just-in-time (JIT) compiler from bytecode to native machine code is justified. Other VMs such as Forth, which has no fixed IL, can reduce the cost by careful IL design: Depending again on the host hardware, a threaded-code (Forth) interpreter costs only one to three overhead machine instructions for each VM operation, giving it performance comparable to compiled code.

Another trade-off to consider is readability (maintainability). Anybody who studied computer science in the last four decades can read grammars; however, TBIL is as hard to read, but not so widespread, as an assembly language. Forth is completely idiosyncratic to the individual programmer and application; it is truly a "write-only" language.

Against that readability fog index we trade off panorama, the ability to see in a single view everything that is going on. Allison's IL was only 110 lines, which could be viewed in less than one column of tiny font on the original *PCC* tabloid, or in two screenfulls on a modern high-res display. Furthermore, it divides nicely in half, the first half being the statement executor, and the rest is expressions and other subroutines. My own TBIL, at 230 lines, is correspondingly less readable.

Implementation Options

There are a variety of techniques to choose from in implementing intermediate-level or pseudocode languages. Most of us are familiar with the early Java runtime interpreter and its replacement on all modern platforms by a JIT compiler. Apple went through the same transition when moving its customer base from the Motorola 68000 CPU family to the IBM Power PC family, and again when dropping the (classic) Mac OS for UNIX. The 68000 instruction set was fully interpreted on the first PowerMacs, at about a 10 performance cost. Several independent developers (including myself) later got substantial performance gains by implementing a binary-to-binary JIT compiler. The "Classic" runtime environment is similarly carried along in OS X to run legacy Mac code, with comparable interpretation penalties.

Allison's (and my) TBIL was implemented in the same manner by a direct interpreter, which preserves the original pseudocode. Similarly, the TBIL interpreter itself preserves the original Basic code in

Grammar Fragment	English Explanation	Formal IL (Allison)	TBIL (Pittman)
line::=	Tiny Basic reads lines	GETLINE	GL
number	possibly with a line number in front	TSTL <label>	BN <label>
--	insert numbered statement into program	INSERT	IL
*	repeat as needed	JMP <label>	BR <label>
statement	do whatever the statement rule says to do	STMT: [label]	:STMT
PRINT	is this keyword "PRINT"?	TST S8,'PRINT'	BC SKIP "PR"
expr-list::=	this is what you can print…	--	
string	quoted string	TST S7,'"'; PRS	BC P7 '"'; PQ
expression	do the expression evaluation rule	CALL EXPR	JS EXPR
--	print the resulting value	PRN	PN
IF	is this keyword "IF"?	TST S9,'IF'	BC INPT "IF"
relop	do compare operator rule	CALL RELOP	JS RELO
THEN	error if this in not keyword "THEN"?	TST S17,'THEN'	BC I1 "THEN"
statement	do whatever the statement rule says to do	JMP STMT	J STMT
GOTO	is this keyword "GOTO"? if not, try another	TST S3,'GO'; TST S2,'TO'	BC PRNT "GO"; BC GOSB "TO"
expression	do the expression evaluation rule	CALL EXPR	JS EXPR
--	error if not end of statement	DONE	BE *
--	find that line in the program, do it next	XPER	GO
LET	is this keyword "LET"? if not, try another	TST S1,'LET'	BC GOTO "LET"
var	save variable name; error if none	TSTV S17	BV *
=	error if there is no "=" here	TST S17,'='	BC * "="
expression	do the expression evaluation rule	CALL EXPR	JS EXPR
--	error if not end of statement	DONE	BE *
--	store the value into the variable	STORE	SV
--	resume with next line	XPER	NX

Table 1: *Partial correlation between Grammar, English pseudocode, and Interpreter Language. Grammar elements in the left column, represented by English pseudocode in the next column, then IL equivalents in the last two columns, Allison's then mine.*

memory as it interprets it. Most full Basic interpreters did a simple kind of JIT translation from Basic text to condensed internal tokens. The Whipple and Arnold Tiny Basic effectively did the same by hand-translating the IL to machine code as they wrote. This removes the panorama of the original IL, but they didn't even use an assembler, choosing rather to code everything in octal absolute, a surprisingly popular Luddite methodology whose proponents have never survived long in the marketplace.

The implementation of Forth bears closer examination because of its heritage in my colleague Drexel Heater's remark. A VM consisting entirely of a library of subroutines called by the high-level problem solution is conceptually not so different from the TBIL approach: Each IL opcode can be thought of as a subroutine (for indeed it is) that is called by the IL code where it is used. Whipple and Arnold replaced those byte codes with the actual subroutine calls, which made their VM more closely resemble my Fortran VMs of 30 years ago. Forth takes that one step further and replaces all the overhead of a normal subroutine call with just the address of the subroutine. In this "threaded code," each successive subroutine call does nothing more than load the next address in sequence into the CPU program counter. The parameters are handled the same way because every operator is threaded code. It still needs to stack and unstack the current IL address entering and leaving nonprimitive subroutines, but the whole interpreter is straightforward.

"The memory bus bandwidth limits in modern computers can completely cancel the speed advantage of native code over interpretation"

Interpreter Speed

The direct interpretation technology chosen for Tiny Basic sacrifices up to an order of magnitude execution speed, but the speed of modern computers reduces the impact. Moreover, the memory bus bandwidth limits in modern computers can completely cancel the speed advantage of native code over interpretation. How can this be? Consider that the Tiny Basic virtual machine (VM) is completely defined in 443 bytes of IL code; the same code in handwritten assembly is more than 2K, a factor of 5 . Although Tiny Basic is tiny, if the IL interpreter generally fits into the primary CPU cache with its IL, while the much larger native code does not, the IL code will execute at native CPU speed while the native code thrashes much slower.

Twelve years ago, I implemented a dynamic recompiler to convert legacy 68000 code being interpreted in Apple's PowerMac computers, into native Power PC code. I expected (and got) about 4 speedup—for small programs. Larger programs actually ran *slower* than the pure interpreter. At first, I suspected my recompiler cache was thrashing, forcing unnecessary recompilation, but careful instrumentation showed that the recompiler was executing less than 10 percent of the total time. A logic analyzer showed that most of the time was spent waiting for the CPU cache to reload from main memory.

I added a large L2 cache, but got only minor improvements. Modern CPUs executing instructions 20 or more times faster than the memory bus exhibit the problem even worse. Recall that the cost of vanilla interpretation is 10 . An IL interpreter could actually make your application run *faster* than native code, depending on actual code sizes. I called this “the RISC penalty” because Reduced Instruction Set Computer (RISC) code is so much less dense than complex instruction sets; high-level ILs are even more dense, exaggerating the effect.

Ignoring the RISC penalty, what else can impact performance? The choice of implementation language is significant. Although “C/C++” are often hyphenated as if they were but one language, they really are not. Though compiler vendors have done a lot to mitigate the overhead cost of late-binding methods in support of inheritance and polymorphism, it is still significant compared to pure C.

Even in pure C, the cost of calling functions is substantial compared to inline code. If the interpreter is carefully designed as a big *switch* statement inside a *while(true)* and no function calls, modern optimizing compilers will give nearly optimal machine speed. In my first attempt at Tiny Basic IL in C, I used autoincrement on byte and *int* pointers for all memory accesses to get maximum performance. I later replaced that with function calls for compatibility to my previous implementations, which specified number stacks implemented as pairs of bytes in memory. This could still be inlined using C macros to eliminate the function call overhead, but at the cost of larger code.

A master artisan controls his tools. For a programmer, that means understanding what kind of code the compiler gives you for each statement type. Allison recommended a cascading tree of nested *IF*s in his “Build Your Own Basic” articles, but his objective was minimal space. I recommend a *switch* statement for better speed. A good compiler can select any of many individual snippets of code in a single indexed jump in constant time, whereas nested *IF* trees require *log(n)* or more individual tests. The Forth interpreter overhead is so small it can be replicated at the end of every operation code, so there isn’t even the overhead to jump to a shared interpreter loop.

Following Allison’s suggestion, my original TBIL interpreters were implemented in assembly language and optimized for minimal space. In this project, I wrote in ANSI Standard C with a modest goal (not entirely served) of tuning for speed. The C source code and some Tiny Basic programs that run in it are available electronically from *DDJ* and at http://www. IttyBittyComputers.com/IttyBitty/TinyBasic/.

DDJ

Listing One

```
;THE IL CONTROL SECTION

START:  INIT                      ;INITIALIZE
        NLINE                     ;WRITE CRLF
CO:     GETLINE                   ;WRITE PROMPT AND GET LINE
        TSTL    XEC               ;TEST FOR LINE NUMBER
        INSERT                    ;INSERT IT (MAY BE DELETE)
        JMP     CO
XEC:    XINIT                     ;INITIALIZE

;STATEMENT EXECUTOR

STMT:   TST     S1,'LET'          ;IS STATEMENT A LET
        TSTV    S17               ;YES, PLACE VAR ADDRESS ON AESTK
        TST     S17,'='           ;(This line originally omitted)
        CALL    EXPR              ;PLACE EXPR VALUE ON AESTK
        DONE                      ;REPORT ERROR IF NOT NEXT
        STORE                     ;STORE RESULT
        NXT                       ;AND SEQUENCE TO NEXT
S1:     TST     S3,'GO'           ;GOTO OT GOSUB?
        TST     S2,'TO'           ;YES...TO, OR...SUB
        CALL    EXPR              ;GET LABEL
        DONE                      ;ERROR IF CR NOT NEXT
        XPER                      ;SET UP AND JUMP
S2:     TST     S17,'SUB'         ;ERROR IF NO MATCH
        CALL    EXPR              ;GET DESTINATION
        DONE                      ;ERROR IF CR NOT NEXT
        SAV                       ;SAVE RETURN LINE
        XPER                      ;AND JUMP
S3:     TST     S8,'PRINT'        ;PRINT
S4:     TST     S7,'"'            ;TEST FOR QUOTE
        PRS                       ;PRINT STRING
S5:     TST     S6,','            ;IS THERE MORE?
        SPC                       ;SPACE TO NEXT ZONE
        JMP     S4                ;YES JUMP BACK
S6:     DONE                      ;ERROR IF CR NOT NEXT
        NLINE
        NXT
S7:     CALL    EXPR
        PRN                       ;PRINT IT
        JMP     S5                ;IS THERE MORE?
S8:     TST     S9,'IF'           ;IF STATEMENT
        CALL    EXPR              ;GET EXPRESSION
        CALL    RELOP             ;DETERMINE OPR AND PUT ON STK
        CALL    EXPR              ;GET EXPRESSION
        TST     S17,'THEN'        ;(This line originally omitted)
        CMPR                      ;PERFORM COMPARISON -- PERFORMS NXT IF FALSE
        JMP     STMT
S9:     TST     S12,'INPUT'       ;INPUT STATEMENT
S1Ø:    CALL    VAR               ;GET VAR ADDRESS
        INNUM                     ;MOVE NUMBER FROM TTY TO AESTK
        STORE                     ;STORE IT
        TST     S11,','           ;IS THERE MORE?
        JMP     S1Ø               ;YES
S11:    DONE                      ;MUST BE CR
        NXT                       ;SEQUENCE TO NEXT
S12:    TST     S13,'RETURN'      ;RETURN STATEMENT
        DONE                      ;MUST BE CR
        RSTR                      ;RESTORE LINE NUMBER OF CALL
        NXT                       ;SEQUENCE TO NEXT STATEMENT
S13:    TST     S14,'END'
        FIN
S14:    TST     S15,'LIST'        ;LIST COMMAND
        DONE
        LST
        NXT
S15:    TST     S16,'RUN'         ;RUN COMMAND
        DONE
        NXT
S16:    TST     S17,'CLEAR'       ;CLEAR COMMAND
        DONE
        JMP     START

S17:    ERR                       ;SYNTAX ERROR

EXPR:   TST     EØ,'-'
        CALL    TERM              ;TEST FOR UNARY -.
        NEG                       ;GET VALUE
        JMP     E1                ;NEGATE IT
EØ:     TST     E1A,'+'           ;LOOK FOR MORE
E1A:    CALL    TERM              ;TEST FOR UNARY +
E1:     TST     E2,'+'            ;LEADING TERM
        CALL    TERM
        ADD
        JMP     E1
E2:     TST     E3,'-'            ;ANY MORE?
        CALL    TERM              ;DIFFERENCE TERM
        SUB
        JMP     E1
E3:T2:  RTN                       ;ANY MORE?
TERM:   CALL    FACT
TO:     TST     T1,"*"
        CALL    FACT              ;PRODUCT FACTOR.
        MPY
        JMP     TØ
T1:     TST     T2,'/'
        CALL    FACT              ;QUOTIENT FACTOR.
        DIV
        JMP     TØ

FACT:   TSTV    FØ
        IND                       ;YES, GET THE VALUE.
        RTN
FØ:     TSTN    F1                ;NUMBER, GET ITS VALUE.
        RTN
F1:     TST     F2,'('            ;PARENTHESIZED EXPR.
        CALL    EXPR
        TST     F2,')'
        RTN
F2:     ERR                       ;ERROR.

RELOP:  TST     RO,'='
        LIT     Ø                 ;=
        RTN
RØ:     TST     R4,'<'
        TST     R1,'='
        LIT     2                 ;<=
        RTN
R1:     TST     R3,'>'
        LIT     3                 ;<>
        RTN
R3:     LIT     1                 ;<
        RTN
R4:     TST     S17,'>'
        TST     R5,'='
        LIT     5                 ;>=
        RTN
R5:     TST     R6,'<'
        LIT     3
        RTN                       ;(This line originally omitted)
R6:     LIT     4
        RTN
```

DDJ