BASIC, Computer Languages, and Computer Adventures

Jerry Pournelle
c/o BYTE Publications
70 Main St
Peterborough NH 03458

It's a typical Sunday afternoon here at Chaos Manor. In one room a dozen kids are playing games on the Radio Shack TRS-80, while here in the office I've been playing about with the C programming language after adding a check-writer to my accounting programs. My wife, the only practical member of the family, gently reminds me of my deadlines: galley proofs of a new novel, *King David's Spaceship* (Simon and Schuster); two chapters of the latest Niven/Pournelle collaboration, *Oath of Fealty* (Simon and Schuster, Real Soon Now); plus three columns; a speech to a librarians' convention; and inputs for a NASA study on America's fifty-year space plan. Some business people worry about cash flow; for authors it's work flow—work comes in bunches, like bananas, and sometimes it seems everything has to be done at once.

So, since it's what we've been doing here lately, I'll talk about computer games and programming languages; a disparate set of topics, but not quite as unconnected as they might seem at first glance.

Languages

One of the biggest unsolved problems in the microcomputer field is languages: which ones are going to be standard? Everyone learns BASIC, of course, because it comes with the machine, and it's a very easy language to learn. Pretty soon, though, you come to the limits of the BASIC supplied with the computer; and then what?

A few years ago there wasn't a lot of choice. You could buy FORTRAN, and perhaps COBOL; you could learn assembler; but then you were stuck. Moreover, there didn't seem to be any obvious advantages to FORTRAN and COBOL, both of which were not only hard to learn, but also difficult to connect up with the computer. Most of the books on those languages were written with big mainframe machines in mind, and the documentation for the small-system versions was, to put it kindly, rather skimpy. Moreover, the user manuals were filled with mysterious references to "logical devices" and other such nonsense, while giving almost no clear examples of how to get programs running on a home computer.

The result was a great expansion of BASICs. What was once a simple teaching language, designed largely to let new users become familiar with the way computers think, became studded with features. Every time you turned around there was a new BASIC interpreter, each one larger than the last, and almost none of them compatible with each other. Whatever portability BASIC had enjoyed vanished in a myriad of disk operations, functions, WHILE statements, new input formats, etc, etc, and, at the same time, the "free" memory left over after loading BASIC got so small that you couldn't handle much data.

The logical end of that process is Microsoft's newest BASIC-80. Understand, it's an excellent BASIC. It has features that, not long ago, the most advanced languages didn't have. It's well documented—at least the commands and functions, which are listed alphabetically, are clearly described. The general information section could be expanded with profit—at present it's written for users who are already more or less familiar with how BASIC operates. There are elaborate procedures for error trapping, and they all work. The editor has been improved. There are procedures (not very well documented) for linking in assembly-language subroutines. You can use long variable names, such as "Personal.data.1" and "Personal.data.2", and be certain the program will know they are different variables.

In other words, there's a lot going for it; but it takes up 24 K bytes of memory, and it's still BASIC. If you want to understand your program six weeks after you write it, you'll have to put in a lot of REMark statements, every one of which takes up memory space. As with all BASICs, you have to sweat blood to write well-structured code (and if you don't bother, that will come back to haunt you when you want to modify the program). And, like all BASICs, it is *slow*. Fairly simple sorts, even with efficient algorithms, take minutes; disk operations are tedious.

I suspect that Microsoft BASIC-80 is the end of the line; they have carried BASIC about as far as it can go. They've done it very well, but they've also reached the inherent limits of the language; and those limits may not be acceptable.

Of course most programmers have always known that

even the best BASIC interpreter wasn't good enough; that if you add enough features to make the language useful, you'll end up with a very slow monster that takes up far too much memory, and that even if you could tolerate those limits, the language itself forces sloppy thinking and inelegant code. However, knowing the problem didn't make the solution obvious; indeed, it's not obvious yet. We can recognize the limits to BASIC and still not agree on what to do about it.

There seem to be two fundamental paths. One is to start over: to relegate BASIC to its original function as a teaching language, and switch to some other language for serious programming. Many took this path, and came out with microcomputer versions of such languages as C, APL, ALGOL, LISP, FORTH, STOIC, and Pascal.

The other way is to *compile* BASIC. One of the first compiled BASICs, BASIC E, is in the public domain; I obtained a fairly decent version with (barely) adequate documentation from the CP/M User's Group several years ago. Then Software Systems brought out an improved BASIC E called CBASIC. It is easy to use and features really excellent documentation, some of the best I've ever seen. It has decent file structures; you are not limited to either *sequential* or *random-access* disk files, but may use sequential operations on random-access files.

There are irritants in CBASIC, particularly with regard to line-printer operations. CBASIC has only the PRINT and PRINT USING commands; there is no LPRINT. To get hard copy, you must execute a LINEPRINTER statement, then one or more PRINT statements, then do a CONSOLE statement to have the copy sent to the terminal. Every time you do the CONSOLE statement, the print buffer empties, and you can get unwanted stuff printed on your hard copy; worse, you can also get unwanted line feeds, making it tough to format hard copy (although CBASIC does allow you to output characters through a port so that, if you are clever enough, you can control the line printer directly; you could even make a CBASIC program drive a Diablo for reverse printing if you wanted to spend the time writing that program). Another needless limitation is that CBASIC allows a maximum carriage width of 133 characters, although a 12-character-per-inch printer can print lines 158 characters long.

Irritants or no, CBASIC is both well designed and well documented. It has WHILE; IF-THEN-ELSE (with chaining); long variable names; and logical operations (IF TAX >0 AND PRICE < MAXIMUM.ACCEPTABLE THEN GOSUB 234 ELSE PRINT "NO GOOD" is a perfectly valid CBASIC expression). It has the CASE (Switch or ON-GOTO) statement.

And it saves memory by compiling. To use CBASIC, one creates a program with any editor that makes ASCII (American Standard Code for Information Interchange) files (Electric-Pencil-created programs have to be put through a converter), then turns the CBASIC compiler loose on it. What comes out isn't true compilation; the compiler strips out remarks and needless line numbers, and compacts the remainder into an INT (intermediate) file; when you want to run the program, you must load in a 10 K-byte run-time package. The INT file is still interpreted; it is not a machine-language program. You can, though, include scads of remarks, put each statement on a separate line, leave lots of blank space, put in rows of asterisks, indent whole sections of the program, and thus vastly increase program readability without using up memory space. A CBASIC program can be written for legibility.

But it's still BASIC. Because a program *can* be reasonably well structured and self-documenting doesn't mean that it will be; BASIC makes it easy to write incomprehensible code and difficult not to. And CBASIC is *very slow*, no faster than Microsoft BASIC-80 and often slower.

There's another limit. It's very hard to write long programs in CBASIC. This problem is inherent in any compiled language—whether true compilation to machine code, or pseudocompiling to an INT file. For example, assume that I want to add a small feature to my accounting package (which I did in fact write in CBASIC two years ago). I load the source program into the text editor. I add the feature and hook it into the program; since I do sweat blood to write structured code, that's fairly easy. Now I must save the altered source and put it through the compiler. Since it's a long program, the compilation takes many minutes—and toward the end, I get a SYNTAX ERROR message. I've put a comma where it wants a semicolon.

Now I have to load the editor, read in the source, make the change, save, and recompile. Presuming that this time it goes without error, I may have used up half an hour just to change "," to ";"—and I still have no test of the program's *logic*. If I now test for logic and it's not right, well, I have to start all over again, hoping that this time I don't manage a new syntax error....

Thus, you can use up a whole afternoon adding something quite simple to a big program. There must be a better way. Why can't someone come up with a language that runs interpretively like normal BASIC, letting you correct both syntax and logic errors while in an interactive mode; and then allow you to compile the result? While we're at it, let's wish for the compiled program to be in true machine language, code that could be put into read-only memory, and, moreover, code that would be *fast*.

That's the route that Microsoft took. Their BASCOM compiler works just that way with their BASIC-80. It will also compile Microsoft BASIC 4.5, and, with considerable modifications to syntax, programs written in both CBASIC and BASIC E. Moreover, it's a very powerful compiler. It implements almost all the features of BASIC-80, including WHILE, IF-THEN-ELSE, CASE, logicals, and string operations, etc. It sounds like the answer to a prayer.

Of course there are problems. Random-access disk operations are unbelievably messy, and worse, a random-access file cannot be accessed sequentially. There's considerable overhead burden. For example, this program:

```
10 PRINT "Hello"
20 END
```

required 9 K bytes when compiled into a CP/M COM file; there's obviously a big run-time package built into BASCOM. Worse, present Microsoft user contracts require that anyone marketing a program compiled by

BASCOM pay a stiff 9% royalty to Microsoft on every *copy sold!* Since this is about equal to the profit margin of many software houses, it's understandable that there's been no great rush to sell programs employing BASCOM.

But let's assume much of this is fixed. Microsoft has a good reputation for responding to customer suggestions. As an example, at the West Coast Computer Faire I spoke to the Microsoft representatives about the lack of a FILES statement (a means of finding out the file names present on disk) in BASCOM; BASIC-80 supported FILES, but not the compiler. Two weeks later I received an updated version of BASCOM—and lo!—the FILES statement had been implemented, along with several features other users had suggested.

At the National Computer Conference, Microsoft representatives said they were "rethinking" their contract policy and would probably change it; that change may have been implemented by the time you read this. I have also mentioned to them the desirability of allowing sequential access to random files, and they've promised to look into that. It's not unreasonable to assume they'll tighten up the overhead-code problem. Thus, as I said, let's assume that the major problems of BASCOM are fixed. What will we have?

First, the combination of BASIC-80 and BASCOM is superb for quick and dirty jobs and for those little special-purpose programs that aren't going to be run very often (possibly only once). For example, I recently wanted to reformat some financial data files. The program had to go open the file, read the data, make a couple of changes, and write the information out in a new format. The only problem was that I also wanted to sort the data before putting it back out, and this had to be done for a *lot* of files. Doing it with interpretive BASIC would take hours and hours; while writing even that simple a program in Z80 assembler would, at best, use up an afternoon, and might take a lot longer.

The solution was to write it in BASIC-80, test syntax and logic while in interpretive mode, and compile with BASCOM. That took an hour. In another hour, I had reformatted about one hundred files. BASCOM is fast, blindingly fast; sorts that take 3.5 minutes in CBASIC are done by BASCOM (using the same algorithm) in under 20 seconds.

In other words, the combination of BASIC-80 and BASCOM has a *lot* going for it. If I'd written this review a year ago, I'd have concluded that BASIC-80/BASCOM was what the world has been waiting for, and spent the rest of the review suggesting incremental improvements to make it even better.

Now I'm not so sure.

The problem is that when all the improvements are done; when all the bugs (if any; I've found none in the latest versions of BASCOM) are eliminated; when all the new features are added; when the code is tightened; when the disk operations are simplified—when all that's done, it's still BASIC.

And there are many who believe BASIC is a dead end; that the inherent limits to the language are just too severe for it ever to be acceptable; that incremental improvements actually harm rather than help the field, because they encourage newcomers to stick with BASIC instead of learning something better. My mad friend is convinced of that. So are a number of my associates.

"But," I protested to my mad friend, "I'm interested in *using* computers. I don't *care* about elegance. What I want is something that lets me get the jobs done quickly, and BASIC-80/BASCOM does that...."

"But at a stiff price. How many times have you had to start over with a program because it just wasn't worth the effort to improve one of those BASIC routines? BASIC doesn't let you build software tools. It's like Pidgin English—you can manage to buy dinner and sell copra with Pidgin, but you'll never write *Hamlet*. Or the Declaration of Independence, or even good laws...."

And the argument starts over and goes on until we get hungry, and, at the bottom line, it's all a matter of opinion; and since my space is limited, I'll drop it for the moment. Just now the bottom line is that BASIC-80 and BASCOM work, and, if you're willing to accept the inherent limits of BASIC, they're quite splendid; but those limits are severe.

Looking Elsewhere

What, then, are the microcomputer user's best alternatives to BASIC? Once again, let me be honest: these are opinions. They're opinions based on considerable user experience, but they're opinions still; and I have found that every known language has passionate supporters, so I am bound to make someone unhappy.

The earliest alternatives to BASIC were FORTRAN and COBOL. These, in my judgment, are languages whose time has long passed. They have little to recommend them, because they have nearly all the limits of compiled BASIC without the advantage of letting you program in the interpretive mode before compiling. I've had both for years, and after an initial flurry of enthusiasm for FORTRAN (I never cared at all for COBOL, which may be all right for very large systems, but is plain crippled on microcomputers) they went on the shelf and haven't come off it. Neither FORTRAN nor COBOL lets you write structured code. True, FORTRAN with RATFOR (excellently described in Kernigan and Plauger's book *Software Tools*, Addison-Wesley, 1976) overcomes some of the limits; but to use RATFOR requires *another* compilation stage, so that it can take over an hour to find and correct a trivial error in a fairly simple program. The *Software Tools* approach to programming is excellent, and I strongly recommend the book; but in my judgment the deficiencies of FORTRAN with RATFOR are simply overwhelming, and I cannot recommend using them.

Then there's Pascal, which very well may be the wave of the future. Pascal began unfortunately: the first widespread implementation of Pascal for microcomputers was from the University of California, San Diego, and it just didn't work for most users. The hooks into the disk operating system were clumsy, and it was *very slow*.

Then came some other versions of Pascal, and they too had horrible problems; you had to be really sophisticated to use them. Bugs appeared, and, unless you knew an awful lot, you couldn't tell whether you'd made a program error or the compiler was at fault. Implementing early Pascals required a constant and fairly complex dialogue between user and publisher.

As a result, a lot of us lost interest in Pascal. The language looked great in theory, but if you couldn't run it, that hardly mattered.

There are now a lot of Pascals; Pascal for the Apple, Pascal for the TRS-80, Pascal for CP/M; Pascal that

pseudocompiles to an INT file the way CBASIC does (Pascal users call the INT file "p-code"); Pascal that truly compiles into machine language for 8080, Z80, 8086, etc. All these look good, and people I respect tell me they run; but since I haven't implemented any of them yet, I can't report on them. I can say that Pascal has many enthusiasts, and might well be the standard language of the future. Then there's Ada, a Pascal-like language heavily supported by the DOD (Department of Defense), which will certainly be around for many years. If I were preparing for a secure career in programming, I'd learn Pascal instantly and keep very close tabs on the progress of Ada.

In the next couple of months, we're adding a Pascal expert to the staff here, and I'll devote a whole column to Pascal/Ada; for now, I must simply pass them over.

Pascal has enthusiasts. So does C, a programming language developed at Bell Telephone Laboratories. The best (and indeed nearly the only) manual on C is Kernighan and Ritchie's, *The C Programming Language* (Prentice-Hall, 1978). This is an excellently written book which anyone at all interested in the C language simply must read. It succeeds in communicating a lot of enthusiasm for C. There are lots of examples of real programs that work. Kernighan, incidentally, is the same Brian Kernighan who coauthored *Software Tools*.

C is nothing like BASIC. There are far fewer commands, for one thing. On the other hand, there are a number of conventions. For example, the BASIC statements:

```
FOR I = 0 TO N - 1
NEXT I
```

would appear in C as:

```
for ( i = 0; i < N; i++)
```

which looks complex, but is, with a bit of experience, quite readable. The i++ means that i is first to be tested against N, then incremented; the expression could have been written with ++i, which would require that i be incremented *before* the test against N.

Despite (perhaps because of) the numerous time-saving conventions such as ++i, C can be learned by a BASIC user in a couple of weeks. Real facility requires practice; more practice than BASIC, precisely because there are many fewer limits in C. Programming with elegance and style takes work—but in C such programs are possible, while BASIC simply won't let you write elegant code.

I have two C compilers for microcomputers. I'm told there's also an interactive tiny-c, which I have not seen running, but which is said to be a good teaching aid, although severely limited in capability. [**Editor's note**: *See "A User's Look at Tiny-c," by Christopher O Kern, December 1979 BYTE, page 196....***RSS**]

Of my two C compilers, only one is suitable for those not already familiar with the C language. This is BDS C, available from Lifeboat Associates for $125. BDS C comes with a copy of Kernighan and Ritchie's book and quite extensive documentation on the BDS (BD Software) implementation.

The BDS compiler uses two passes. One might at first think that a disadvantage because of the time required, but in fact it is not: the first pass is done *very* fast, and checks for trivial errors, such as missing semicolons, comments improperly delimited, unmatched parentheses and brackets (C *loves* brackets, braces, and parentheses), and the like. The second pass goes a bit slower but is still much faster than the CBASIC compiler.

Like BASCOM, compiled C code must be put through a linker, and like Microsoft's, the BDS documentation tells you precisely how to do this. When it's all finished, you have a CP/M command file; and the resulting code is *very* fast. I've not yet been able to benchmark BDS C against a similar BASCOM program, because when you translate from BASIC to C you actually restructure the program; but I have two *Othello* games, one in C and the other compiled by BASCOM, and they seem to run at about the same speed. The C program, however, is about 8 K bytes compiled; the BASIC program, performing the same searches and playing at the same level, compiled to over 20 K bytes. Other programs doing similar jobs also run in comparable times, and with about the same differences in program size.

Disk operations in BDS C are fairly simple if you understand CP/M, not so simple if you don't—and CP/M's documentation is so notoriously unclear that you'll have to work for a couple of days understanding CP/M before you can write decent disk I/O (input/output) operations for BDS C. It is worth sharpening up your understanding of CP/M, though, because BDS C lets you do *everything* CP/M will: get the names and sizes of files currently on disk, make backups, rename and delete, etc, and it's no more difficult to understand than the FIELD statements in Microsoft BASIC or the dreaded FORMAT statement in FORTRAN.

String operations in C are more difficult than in BASIC. Actually, they aren't; ie: it's possible to write, in C, all the string functions of BASIC (such as LEFT$, etc), then call them as needed; and once you have written them, you can use them in any program that needs them —and leave them out if not wanted. And, in fact, that illustrates one of the fundamental differences between BASIC and C: the BASIC language provides a number of functions which you must have present whether you need them or not, and which must be used *exactly* the way BASIC wants them used. C, on the other hand, allows you to leave out functions you don't want, and rewrite those you keep to suit your precise requirements.

There is, however, one very severe limit to BDS C: it does not support floating-point data types. One can use floating-point *variables*, because BDS supplies a number of functions that can be called to do floating-point arithmetic; but the result is clumsy. If you want to learn the C language, and write games, calendar programs, and almost anything that doesn't involve crunching a lot of numbers, BDS C is highly recommended; however, it isn't suitable for writing an accounting or financial package.

The other C compiler for microcomputers is the Whitesmiths C Compiler, which is available from Lifeboat Associates for $630. This is a full implementation of the standard C described by Kernighan and Ritchie, and is highly regarded by many professionals who work with large machines like DEC's (Digital Equipment Corporation) PDP-11. In fact, Whitesmiths C was written for large machines, and it is only an accident that it could be scaled down for microcomputers. The president of Whitesmiths Ltd is P J Plauger, a fellow science fiction writer, and more important, coauthor of *Software Tools*.

Although the Whitesmiths Compiler is an excellent professional tool, I cannot recommend it to anyone who doesn't intend to program in C in a big way—and even then I'd recommend buying the BDS C compiler as well. Whitesmiths C compiles, eventually, to true machine code; but it does so by going through an intermediate assembly language called A-Natural. It's slow, and since there's no first pass to find trivial errors, the Whitesmiths compiler can grind away for half an hour before reporting a misplaced semicolon. It is certainly not what I'd choose to learn the language with—but I would get it if I were going to market programs written in C.

Ubiquitous Microsoft doesn't market a C compiler, but it does have a LISP interpreter. The Microsoft muLISP-79 is well done, if you like the LISP language. You may not care for the language, but those who like it like it a lot. *LISP* stands for list processing, and it makes creating highly complex linked lists very easy.

LISP is, however, a peculiar language. It was written in the 1950s by Dr John McCarthy, now Director of the Stanford Artificial Intelligence Laboratories (SAIL), and it's extensively used at Stanford and MIT (where McCarthy wrote it).

LISP does bit-by-bit arithmetic, meaning that there is no theoretical limit to the precision you can obtain; if you want an exact numerical expansion of, say, 2 to the 55th power, or 87 factorial, you can get it from LISP, and with only about three lines of code for a program—and you'll get the answer faster than you think. LISP is one of the fastest languages I know of, often approaching assembly-language programs in speed of operation.

LISP programs are very tight; it's almost impossible to write unstructured code in LISP. It's also very nearly impossible to understand a LISP program, even if you wrote it; at least that's been my experience. You can strain like a gearbox and produce code that runs, and which you understand just at that moment; but hours later it's gibberish. The only thing less comprehensible than a LISP program is one in APL—APL doesn't even use normal letters, but instead requires a special keyboard that can generate strangely bent arrows and other weird symbols. Both LISP and APL programmers delight in writing a whole page of instructions into one line (and you can do it, too, because both languages allow functions to call themselves). They also like to baffle fellow professionals by showing a line of code and challenging anyone to say what it does.

It's very hard to comment a LISP program—but that's all right, because it isn't traditional for LISP programmers to comment their programs anyway.

In other words, I am not a wild enthusiast for LISP as a "standard" microcomputer language. It's true that one or another LISP variant is used by just about everyone in the artificial intelligence field; for certain purposes there's nothing better. But for general-purpose programming, LISP and APL are, in my judgment, simply too obscure.

The Microsoft muLISP-79 was written by The Soft Warehouse in Hawaii; I got mine directly from the authors and haven't seen the Microsoft versions (for CP/M and the TRS-80), although they were supposed to be sent weeks ago. I am told that Microsoft has rewritten some of the documentation, which could only improve it. The problem with documenting LISP is that the language is fairly obscure; you need not only a user's manual, but an introduction to LISP itself, which is far more than the muLISP-79 manual claims to be.

The best way to learn LISP is to attend Stanford or MIT and get tutorial instruction from someone already proficient. The next best way is to get access to the MIT Macsyma Consortium computer and run the TEACH-LISP programs. There are also a couple of MIT documents which are pretty good introductions. I wish I knew of a good commercial textbook, but I don't. If you want to learn LISP, you've no choice but to play about with it; since muLISP-79 is interactive, that's not so hard to do, and there are some decent examples in the documents suppied. If you like playing with powerful languages, muLISP-79 is recommended—but don't blame me if you don't use it very often after the first wave of enthusiasm.

Which concludes my overview of languages. I haven't mentioned STOIC and FORTH, because they're really a kind of assembler language using the programmer as a parser; they make programming a bit easier, but you've got to be into assembler work before you can use them, and this is, after all, the User's Column.

Drawing Conclusions

So what's the best language to learn? I don't know. I like C. I also like what I've seen of Pascal, assuming the current crop will really run on microcomputers. And despite my misgivings, I still find myself using BASIC-80/BASCOM, particularly for quick and dirty jobs.

It seems certain—to me at least—that Pascal is going to be around a long time, especially what with all that DOD support for the Ada variant. Now that there seem to be some decent Pascal compilers available for microcomputers, we're going to see a lot of software written in Pascal, and those who want to modify their software will have to be familiar with the language.

But there may not be a real conflict between Pascal and C. Both are vastly different from BASIC; different in conception, in terminology, but more important, in the "philosophy" or style of programming employing them. Learning either will help break the BASIC habit of sloppy program structure; and having done that, you'll have little trouble learning the other, or indeed any other well-structured language.

And that can't hurt users or programmers.

Adventure and Other Games

Now, what about computer games? Well, when microcomputers first came out, games were the rage. It wasn't so much fun to *play* the games, which tended to be rather dull (you wouldn't play much tic-tac-toe with a human opponent); the fun was in writing the programs and seeing just how smart you could make the machine. With the possible exception of *Star Trek*, nobody spent much time with the games once they were written and perfected.

That's no longer true. Nowadays you can buy computer games that are fun to play. For example, at both the West Coast Computer Faire and the National Computer Conference, the most popular exhibit was Atari's. Not that so many were wild about the Atari computers, or the educational games, or that sort of thing, but boy did they stand in line to play *Star Raiders*, a real-time game in which you are a pilot of an X-wing fighter, or perhaps it's a Colonial Viper, and you go zipping about through space destroying villains and saving civilizations....

There are lots of real-time games showing up for

microcomputers. *Alien Invader, Space War,* and a whole family of games formerly available only in arcades can be your very own.

There's also an entirely different class of game available. *Adventure* is here.

The game of *Adventure* was first written in FORTRAN by Larry Crowther and Don Woods. It bore some slight resemblance to *Hunt the Wumpus,* in that the game consisted of wandering through unknown territories and encountering various hazards. Unlike *Wumpus,* though, the *Adventure* map is fixed. The game always begins at a well house, and you may continue to explore until you are killed. Actually, it doesn't end even then: the computer will resurrect you if you like.

You move about in *Adventure* by telling the computer where you want to go. The object of the game is to find treasures and bring them to the well house. On the way you encounter various obstacles and monsters, such as a large green snake, a dragon, and a ferocious bear chained to the wall. (The problem is that the bear's silver chain is a treasure.) You also find various objects: a rod, a birdcage, and other such things, some of which may be useful in solving puzzles that lead to treasure.

The game quickly became a cult object among programmers. Computer-installation supervisors estimated that when *Adventure* arrived, two weeks' work would be lost due to the staff bootlegging time to run the game. Various fixes were tried, including restricting the times at which *Adventure* could be accessed, but nothing really worked except letting the disease run its course; when all the programmers had solved the game, then and only then did they get back to work. Until then, they were driven to it as if hypnotized. To make it worse, it was customary not to tell anyone how to solve the game, although strange and misleading hints were allowed.

Adventure now exists for various microcomputers. The game itself is public domain (although programs to implement it are not), so there are many versions offered. I have one for 8-inch floppy-disk CP/M systems sold by Workman and Associates (POB 482, Pasadena CA 91102, $23.95 postpaid) and another for the Radio Shack TRS-80 Level II (Model I) by Microsoft, $24.95, and available from most dealers. Both run quite fast—faster, in fact, than the FORTRAN versions did on a DEC PDP-10. Both require 32 K bytes of memory and a single disk drive, and both are full implementations of the original Crowther and Woods *Adventure,* including the "Save" feature that allows you to store an incomplete game so that you don't have to start over every time.

The Workman version recognizes a number of commands that were not in the original *Adventure,* but the puzzles and their solutions remain unchanged.

In addition, both the Workman and the Microsoft versions store most of the game information on disk, and every time you give a command they have to go to the disk to get the response. There's no help for that, of course; the *Adventure* data base requires over 50 K bytes of ASCII (American Standard Code for Information Interchange) characters. Thus the disk gets a good workout. This presents no problem with the Workman and Associates CP/M version, because any good CP/M copy routine will allow you to make a backup; but the Microsoft TRS-80 *Adventure* has been carefully rigged to make backup copies nearly impossible. I say nearly; within

either TRSDOS or Apparat's NEWDOS it is impossible, but since I have the Omikron CP/M Mapper installed on my TRS-80, I can make backups of anything, using a CP/M sector-by-sector copy routine.

(As an aside: I've been informed that both Parasitic Engineering and Field Engineering Consultants Ltd also make memory mappers that will allow you to run CP/M on the TRS-80 Model I. I've had no chance to test either of them. My Omikron Mapper continues to work flawlessly, by the way.)

I often wonder about companies that deliberately try to keep you from copying software—especially when it's supposed to run on something as inherently flaky as a TRS-80 5-inch disk. Experienced users *never* run their primary source disks; making a backup is just common sense, even if you have excellent hardware like Percom or Matchless disk drives. (I've tested both on my TRS-80, and I'm quite happy with them.) Moreover, making it hard to copy a disk is often like waving a red flag at a bull —there are plenty of sophisticated users who will consider it a challenge, and, having with great effort found a way to make copies, will feel ethically justified in distributing them to all their friends.

In any event, the Workman and Microsoft *Adventure* implementations have provided many hours of trouble-free enjoyment, and I recommend them highly.

Just after the *Adventure* craze hit, there were rumors of another game, *Zork*, which is to *Adventure* as *Adventure* is to *Wumpus*. *Zork* was developed at the Massachusetts Institute of Technology by "the Four Implementors": Tim Anderson, Marc Blank, Bruce Daniels, and David Lebling. The game was written in MDL (or "Muddle"), a LISP-like language, and featured an *enormous* underground dungeon, dozens of clever puzzles, and a highly intelligent command parser that understands much that *Adventure* finds incomprehensible. Although *Zork* never quite caught on the way *Adventure* did, it became widespread—and where it did appear, it cost more time than ever *Adventure* had, because it was both more difficult and more interesting.

Implementors Lebling and Blank have devised a microcomputer implementation of *Zork* in two parts. *Zork: The Great Underground Empire, Part I* is being sold for the Apple II and the TRS-80 on 5-inch floppy disk by Personal Software, 1330 Bordeaux Dr, Sunnyvale CA 94086, at a price of $39.95. [**Editor's note**: *Part II is still under development, but Part I alone constitutes a complete game that can be played through to a satisfactory ending....***RSS**] Like the Microsoft *Adventure*, *Zork* requires constant access to the disk but cannot be copied by normal means. I've been just a little afraid of running the primary disk, so I haven't checked out everything; besides, the kids are still mapping *Adventure*. I've played with this *Zork* enough to know that I like it (and I wasted incredible amounts of time playing the original *Zork* on a PDP-10).

Adventure and *Zork* became popular during the *D & D* (*Dungeons and Dragons*) craze—a madness which shows little sign of peaking out even yet. It was inevitable that other *D & D* games would come forth, and sure enough, Automated Simulations Inc (ASI), POB 4232, Mountain View CA 94040, has come out with a whole series, from the introductory *Datestones* to the full four-level dungeon in *Temple of Aphsai*. These games are sold in

both tape cassette and disk versions, and they range in price from $14.95 for *Morloc's Tower* on cassette to $26.95 for the disk version of *Temple of Apshai*. ASI guarantees these games to be interesting, and I don't think they refund much money. They've been very popular around here.

There are versions for TRS-80, PET, and Apple computers; the Apple versions make extensive use of Apple's excellent graphics, so that monsters like Ant-man and the Wolf look pretty good. Unfortunately, the TRS-80 doesn't have such nice graphics, and the characters and monsters look like blobs. Unlike *Zork* and *Adventure*, these games are played in real time, and, instead of a room description, the computer draws a map, placing monsters and treasures in it where appropriate. They're very playable games, guaranteed to waste more time than you really expected to put into them.

The *real* time wasters for me, though, have been Automated Simulation's space war games, *Starfleet Orion* (two players) and *Invasion Orion* (one player against the computer). These games allow a number of different scenarios; ten or so are supplied (along with a pleasantly written background and story data including characterizations), but the user can make up his own, so that in effect either of these games has an infinite number of variants. The rulebooks also give a number of standard warship types, but once again the user can design his own, from torpedo boats to dreadnoughts to armored planets. There are Tractor and Pressor beams, something much like a phaser, torpedoes and missiles, and quite a lot of the flavor of a space battle.

I'd like the single-player version somewhat better if it were faster; in my favorite scenario, *Damocles*, it can take several minutes for the computer to plan out its move, and worse, you can't just go away, because the battle results are presented dynamically and can't be recalled once shown.

The *Orion* games are quite realistic. Classical principles of fleet warfare work, and strategy and tactics are more important than luck. Since players can modify the ships at will, it's possible to tailor the games to a balance of power so that an experienced player (or the computer in the single-player version) doesn't routinely stomp a newcomer, and the game can be changed again as the players gain experience.

All of the Automated Simulations games are implemented in BASIC. They can be copied, listed, and even modified. In theory, one ought to be able to compile *Invasion Orion* with Microsoft's TRS-80 version of BASCOM and thus speed it up. Obviously, you will have to modify the games a bit; in particular, you will have to lengthen the loops that govern how long displays stay visible. I'm anxious to try this, but so far the TRS-80 BASCOM hasn't arrived, so I can't say for certain that it will work.

Needless to say, I enjoy all the Automated Simulations games, and recommend them highly. And, needless to say, I enjoy the C language and BASIC-80, etc, etc. So what does it all mean? Well, it means that I have to get the kids away from the TRS-80 and have some computer fun of my own, here at Chaos Manor.

See text box on page 238

Items Reviewed

Microsoft
10800 NE 8th, Suite 819
Bellevue WA 98004

muLISP-79	*CP/M 8-inch disk system*	*$200*
BASIC-80	*CP/M*	*$350*
BASCOM BASIC Compiler	*CP/M, ISIS-II, TRSDOS*	*$395*
Adventure	*TRS-80 Model I Level II 32 K disk*	*$24.95*

Lifeboat Associates
1651 Third Ave
New York NY 10028

BDS C Compiler	*CP/M*	*$125*
Whitesmiths C Compiler	*CP/M*	*$630*
CBASIC2	*CP/M*	*$120*

Workman Associates
POB 482
Pasadena CA 91102

Adventure	*8-inch disk CP/M 32 K 8080 or Z80*	*$23.95*

Automated Simulations Inc
POB 4232
Mountain View CA 94040

(Following are available in disk or cassette versions)

The Temple of Apshai	*TRS-80, PET, Apple*	*$24.95*
The Tower of Morloc	*TRS-80, PET, Apple*	*$14.95*
Starfleet Orion	*TRS-80, PET, Apple*	*$19.95*
Invasion Orion	*TRS-80, PET, Apple*	*$19.95*

Personal Software
1330 Bordeaux Dr
Sunnyvale CA 94086

Zork: The Great Underground Empire, Part I	*TRS-80 or Apple disk*	*$39.95*

Books

Kernighan, Brian W and Dennis M Ritchie.
The C Programming Language.
Englewood Cliffs NJ: Prentice-Hall Software Series, Prentice-Hall, 1978, $13.95

Kernighan, Brian W and P J Plauger.
Software Tools.
Reading MA: Addison-Wesley Publishing Company, 1976, $11.95

