ATLAST

Autodesk Threaded Language Application System Toolkit

Open, programmable products are superior to and displace even the best designed closed applications. A threaded language, implemented in a single portable C file, allows virtually any program, existing or newly developed, to be made programmable, extensible, and open to user enhancement.

by John Walker
Revision 1 by Duff Kurland—November 20, 1990

YOU'D THINK WE'D HAVE LEARNED by now. It was Autodesk's strategy for AutoCAD® from inception that it should be an open, extensible system. We waged a five-year uphill battle to bring such a heretical idea to eventual triumph. Today, virtually every industry analyst agrees that AutoCAD's open architecture was, more than any other single aspect of its design, responsible for its success and the success that Autodesk has experienced.

And yet, even today, we write program after program that is closed—that its users cannot program—that admits of no extensions without our adding to its source code. If we believe intellectually, from a sound understanding of the economic incentives in the marketplace, that open systems are better, and have confirmed this supposition with the success of AutoCAD, then the only question that remains is *why?* Why not make every program an open program?

Well, because it's *hard*! Writing a closed program has traditionally been much less work at every stage of the development cycle: easier to design, less code to write, simpler documentation, and far fewer considerations in the test phase. In addition, closed products are believed to be less demanding of support, although I'll argue later that this assumption may be incorrect.

The painful path to programmability

Most programs start out as nonprogrammable, closed applications, then painfully claw their way to programmability through the introduction of a limited script or macro facility, succeeded by an increasingly comprehensive interpretive macro language which grows like topsy and without a coherent design as user demands upon it grow. Finally, perhaps, the program is outfitted with bindings to existing languages such as C.

An alternative to this is adopting a standard language as the macro language for a product. After our initial foray into the awful menu macro language that still burdens us, AutoCAD took this approach, integrating David Betz' XLISP, a simple Lisp interpreter which was subsequently extended by Autodesk to add floating point, many additional Common Lisp functions, and, eventually, access to the AutoCAD database.

This approach has many attractions. First, choosing a standard language allows users to avail themselves of existing books and training resources to learn its basics. The developer of a dedicated macro language must create all this material from scratch. Second, an interpretive language, where all programs are represented in ASCII code, is inherently portable across computers and operating systems. Once the interpreter is gotten to work on a new system, all the programs it supports are pretty much guaranteed to work. Third, most existing languages have evolved to the point that most of the rough edges have been taken off their design. Extending an existing language along the lines laid down by its designers is much less likely to result in an incomprehensible disaster than growing an ad-hoc macro language feature by neat-o feature.

Unfortunately, interpreters are *slow*, *slow*, *slow*. A simple calculation of the number of instructions of overhead per instruction that furthers the execution of the program quickly demonstrates that no interpreter is suitable for serious computation. As long as the interpreter is deployed in the role of a macro language, this may not be a substantial consideration. Most early AutoLISP® programs, for example, spent most of their time submitting commands to AutoCAD with the `(command)` function. The execution time of the program was overwhelmingly dominated by the time AutoCAD took to perform the commands, not the time AutoLISP spent constructing and submitting them. However, as soon as applications tried to do substantial computation, for example the parametric object calculations in AutoCAD AEC, the overhead of AutoLISP became a crushing burden, verging on intolerable. The obvious alternative was to provide a compiled language. But that, too, has its problems.

Introducing ATLAST

ATLAST™ is a toolkit that makes applications programmable. Deliberately designed to be easy to integrate both into existing programs and newly-developed ones, ATLAST provides any program that incorporates it most of the benefits of programmability with very little explicit effort on the part of the developer. Indeed, once you begin to "think ATLAST" as part of the design cycle, you'll probably find that the way you design and build programs changes substantially. I'm coming to think of ATLAST as the "monster that feeds on programs," because including it in a program tends to shrink the amount of special-purpose code that would otherwise have to be written while resulting in finished applications that are open, extensible, and more easily adapted to other operating environments such as the event driven paradigm.

The idea of a portable toolkit, integrated into a wide variety of products, all of which thereby share a common programming language seems obvious once you consider its advantages. It's surprising that such packages aren't commonplace in the industry. In fact, the only true antecedent to ATLAST I've encountered in my whole twisted path through this industry was the universal macro package developed in the mid 1970's by Kern Sibbald and Ben Cranston at the University of Maryland. That package, implemented on Univac mainframes, provided a common macro language shared by a wide variety of University of Maryland utilities, including a text editor, debugger, file dumper, and typesetting language. While ATLAST is entirely different in structure and operation from the Maryland package, which was an interpretive string language, the concept of a cross-product macro language and appreciation of the benefits to be had from such a package are directly traceable to those roots.

So what *is* ATLAST? Well... it's FORTH, more or less. Now I'm well aware that the mere mention of FORTH stimulates a violent immune reaction in many people second, perhaps, only to that induced by the utterance of the dreaded word "LISP." Indeed, more that 12 years after my first serious encounter with FORTH, I am only now coming to feel that I am truly beginning to "get it"—to understand what it's really about, what its true strengths (and weaknesses) are, and to what problems it can offer uniquely effective solutions. PostScript had a lot to do with my coming to re-examine FORTH, as did my failed attempt in early 1988 to separate AutoCAD's user interface from the geometry engine. That project, The Leto Protocol, ended with my concluding that to succeed: to create an interface that would not grow to unbounded size, bewildering complexity, and glacial performance, it would be necessary to embed programmability within the core—to provide a set of primitives that could be composed, by the user interface module, into higher-level operators that could be invoked across the link between the two components. This programmability would, of course, have to be in a portable form and not involve linking user code into the AutoCAD core.

In looking for parallels to the problem I faced, PostScript seemed similarly motivated and reasonably effective in accomplishing its goals. (One can certainly attack PostScript on performance, although I suspect its performance problems stem more from the underlying execution speed of the graphics primitives and the inefficient ASCII representation of input than any inherent aspect of the language.) Certainly PostScript blew away its competitors, such as Impress and DDL, almost without taking notice of them. Further, it seemed apparent that PostScript's success was another example in the long list of open, programmable products that triumphed over "more comprehensive" but non-extensible ones.

Looking at PostScript inevitably brings one back to the language that inspired it, FORTH. Although FORTH has a reputation for obscurity and seems to attract an unusually high percentage of flaky adherents, it has many attributes that recommend it as a candidate for a portable tool to make any application programmable.

It is small. A minimal implementation of FORTH is a tiny thing indeed, since most of the language can be defined in itself, using only a small number of fundamental primitives. Even a rich implementation, with extensions such as floating point and mathematical functions, strings, file I/O, compiler writing facilities, user-defined objects, arrays, debugging tools, and runtime instrumentation, is still on the order of one fifth the number of source lines of a Lisp interpreter with far fewer built-in functions, and occupies less than of 70% the object code size. Runtime data memory requirements are a tiny fraction (often one or two percent) of those required by Lisp, and frequently substantially less that compiled languages such as C. It's kind of startling to discover that an entire interpretive and compiled language, including floating point, all the math functions of C, file I/O, strings, etc., can be built, in large model, into a DOS executable of 50964 bytes. It can.

It is fast. Because it is a threaded language, execution of programs consists not of source level interpretation but simple memory loads and indirect jumps. Even for compute-bound code, the speed penalty compared to true compilers is often in the range of 5 to 8. While this may seem a serious price to pay, bear in mind that tokenising Lisp interpreters often exhibit speed penalties of between 60 and 70 to 1 on similar code, and source-level interpreters, such as the macro languages found in many application programs, are often much, much worse than that. In most programs, the execution speed of FORTH and compiled code will be essentially identical, particularly when FORTH is used largely in the role of a macro language, calling primitives within an application coded in a compiled language.

It is portable. If the implementation rigidly specifies the memory architecture and data types used (and this can be done with essentially no sacrifice in speed), FORTH programs can be made 100% compatible among implementations. Programs can be transferred as ASCII files, universally interchangeable across systems. Application data types defined in FORTH, using its object creation facilities, automatically gain the portability of the underlying data types.

It is easy to extend. Because the underlying architecture is very simple (unlike, for example, that of a Lisp interpreter), any competent C programmer with a minimum of indoctrination can begin adding C-coded primitives to a C-implemented FORTH within hours. These C primitives will run at full speed, yet be able to be parameterised, placed in definitions, used in loops, etc., from any FORTH construct. This leads to a different way of building applications. Rather than programming the structure and primitives as a unified process, one builds the application-unique primitives that are needed, tests them interactively as they are built, then assembles the application with glue code written either in FORTH or C depending upon considerations of efficiency, security, and the extent to which one wishes to make the underlying primitives visible to and accessible by the user. Unlike conventional program development processes, these considerations are not yes-or-no decisions but, for the most part, continua along which the product may be positioned at the point desired and subsequently adjusted based upon market feedback.

It is interactive. While most portions of a FORTH program are compiled into a form equally compact and comparable in execution speed to machine code, direct user interaction can always be furnished simply by providing a connection from the user's keyboard to the interpreter (or conversely, blocked by denying the user that access). That such interactivity expedites program development compared to the normal edit, compile, link, debug cycle is well known. That FORTH can provide it without sacrificing execution speed is one of its major attractions.

It supports multiple operating paradigms. Once the technique of encapsulating the functionality of a product in primitives accessible from the FORTH environment is mastered, it is possible to build programs in which the core facilities (for example, database access, geometric calculations, graphical display of results, calculating mass properties) can be composed into sequences that can be invoked from a program, called interactively from a command line, triggered by a menu selection or pick of a button in a dialogue, or virtually any other form of interaction imaginable. Further, since any stimulus that affects the program simply executes a FORTH word, and such words can be easily redefined with a small amount of FORTH text, any of these operating modes can be rendered programmable by the implementor, third party developer, or user, at the discretion of the designer.

It is surprisingly modern. Although FORTH appears to be an artifact of the bygone days of 64K computers and teletype machines, many of its concepts, viewed through contemporary eyes, are remarkably up to date. For example, few languages share its ability to define new fundamental data types, along with methods that operate upon them. The multiple dictionary facility of FORTH permits one to create objects that inherit, by default, properties of their parents, and to implement such structures in an efficient manner.

Atlast and FORTH

All of these advantages do not erase some substantial shortcomings of FORTH, particularly in the modern programming environment. In defining Atlast, I have attempted to conform to FORTH wherever possible, without compromising my overall goal of creating a system that would allow a developer to factor out the programmability from an application and hand it to a standard module to manage, precisely as C programmers delegate I/O and mathematical function evaluation to library routines provided for those purposes.

Atlast is based on the FORTH-83 standard and incorporates many of the optional extensions and supplementary words defined in that standard. Once the basic differences between FORTH and Atlast have been mastered, one can use a FORTH reference manual for most user-level Atlast programming tasks. The major differences between FORTH-83 and ATLAST are as follows.

Integers are 32 bits. To bring forth another language burdened with 16 bit integers in the year 1990 is, to my mind, unthinkable. We are rapidly entering an era where the vast majority of C language environments agree that the `int` type is 32 bits, and applications may be expected to rapidly conform to this standard. Consequently, in ATLAST, all integers are 32 bits and no `short` data type is provided. Note that this does not imply incompatibility with C environments with 16 bit `int`s—ATLAST works perfectly with Turbo C on MS-DOS and Microsoft C on OS/2, for example, because all integers are explicitly declared as `long`.

Identifiers are arbitrary length. In ATLAST, you need not struggle with the tradeoff between memory efficiency and uniqueness of identifiers that plagues the FORTH programmer. Identifiers are limited in length only to the size of the built-in token assembly buffer, which defaults to 128 characters, and all characters are significant. Again, this change brings ATLAST more closely into conformance with contemporary language designs. To implement this change, symbol names were moved from the heap into dynamically allocated buffers, taking advantage of the underlying C runtime environment. This makes the task of adjusting heap size easier (and changes some of the arcana of programs that fiddle with the low-level structure of the system, but everything you could do in FORTH, you can do in ATLAST, albeit in a slightly different way).

Floating point is supported. Floating point constants, variables, operators, scanning and formatting facilities, and a rich set of mathematical functions are provided as primitives (which can be turned off at compile time, if not needed). Compatibly with C, the default floating point type is 64 bit C `double` precision numbers. The only assumption made by ATLAST about floating point format is that a floating point number is twice the size of an integer. The rational number facilities of FORTH are not provided in ATLAST.

Strings are supported. Strings are supported at a much higher level in ATLAST than in FORTH. String literals are provided in a general and explicit manner using the C syntax for escaping special characters. A rich set of string processing functions which closely follow those of C are provided (`STRCPY`, `STRCAT`, `STRLEN`...). A mechanism of cyclically allocated temporary string buffers provides more flexible manipulation of strings in interactive input. Strings continue to follow the pointer and buffer model used by both C and FORTH. String-intensive programs should run at about the same speed as their equivalents in C or FORTH.

Debugging facilities are provided. ATLAST can be configured at compile time with as much or as little error checking and debugging support as is appropriate for the application in which it is being integrated and the development status of that product. During development and test, one can configure ATLAST with an optional `TRACE` that follows program execution primitive by primitive, a `WALKBACK` that prints the active word stack when an error is detected, precise overflow and underflow checking of both the evaluation and return stacks, and close to bulletproof pointer checking that catches attempts to load or store outside the designated heap area. Although sufficiently crafty programs can still crash ATLAST, errors that slip past the checking and wreak havoc are extremely rare, even in unprotected environments such as MS-DOS. This, combined with the fundamental interactivity of ATLAST, makes for a friendly debugging environment. All the runtime error checking can be disabled to reduce memory and execution time overhead, when and where appropriate.

File I/O follows C and Unix conventions. FORTH was developed before the age of standard operating systems; in its early days, it *was* the operating system of many of the minicomputers which ran it. Now that the Unix file system interface has become a *de facto* industry standard, ATLAST conforms to that model of file system operation. `FILE` variables correspond to C language file descriptors, and a familiar set of primitives such as `FOPEN`, `FCLOSE`, `FREAD`, `FSEEK`, etc., are used in the same manner as in C. Line-level I/O is provided as well, offering AutoCAD-compatible automatic recognition of ASCII files written with any of the current end of line conventions.

Extensive support for embedding is provided. Unlike FORTH, ATLAST is intended to be invisibly embedded within application programs. Other than providing a common framework for programmability and extension, the application continues to "look like" itself, not like ATLAST or FORTH. Thus, ATLAST is not "in control" in the sense that the main loop of a FORTH system is; it is a slave, called by the application at appropriate times. Accomplishing this required inverting the control structure from that of a typical FORTH system and providing a comprehensive set of C callable linkages by which the application communicates with Atlast. In addition, primitives are provided which aid in tuning Atlast to the precise needs of the host program. The developer can monitor memory usage, note which primitives are used and which are not, and configure a custom version of Atlast ideally suited to the needs and environment of the host program.

A note on what follows

In order to illustrate Atlast, the balance of this paper employs numerous sample programs and fragments of Atlast code. A reader with a basic understanding of FORTH should, along with the definitions of the Atlast primitives given at the end of the paper, be able to figure out what is going on in the examples. If you've never encountered FORTH before, the examples may seem little more than gibberish. Don't worry—once you get the hang of it, or consult one of the many excellent FORTH books available (I recommend *Mastering Forth*, by Anderson and Tracy, New York: Brady Books/Prentice-Hall, 1984), all will become clear.

Until then, don't be put off by the examples. Just skim over them *as if* you understood them. You'll still pick up the flavour of the package, how it integrates with applications, and what you can do with it. I'd like to be able to leave my brain and fingers running overnight and find a complete Atlast reference manual that could stand by itself sitting on my machine the next day. Alas, I lack overnight batch capability and have no opportunity to undertake such a task in prime time at present. I decided to supply the documentation in this oddly incomplete form to get the essentials across to those who can understand it rather than defer the entire effort until I can complete a hundred pages or so of documentation that largely duplicates a FORTH reference manual.

Interactive Atlast

Although Atlast is intended to be embedded in application programs, for learning the language, experimenting with small programs, and using it as a desk calculator, it's handy to have an interactive stand-alone version. The Atlast source distribution includes a main program, `atlmain.c`, that can be linked with Atlast to provide such a utility. The executable, called `atlast` on Unix and `ATLAST.EXE` on MS-DOS, is built with all error checking enabled to aid in program development.

To experiment with Atlast, execute the interactive program with:

```
atlast
```

You'll be prompted with:

```
->
```

as long as Atlast is in the interpretive state. For example, you might load Atlast and experiment with various rational approximations of π.

```
% atlast
-> 22.0 7.0 f/ f.
3.14286 -> 377.0 120.0 f/ f.
3.14167 -> ^D
%
```

Note that Atlast does not explicitly return the carriage after output; use the `CR` primitive if you wish this done. Rather than printing each number and comparing it manually against π, we can define a *constant* with the value of π and a new *word* (or function) that compares a value against it and prints the error residual. Here's how we might do that:

```
% atlast
-> 1.0 atan 4.0 f* 2constant pi
-> : pierr
:>   pi f- fabs f. cr
:> ;
-> 3.0 pierr
.141593
-> 22.0 7.0 f/ pierr
0.00126449
-> 355.0 113.0 f/ pierr
2.66764e-07
-> ^D
%
```

We can also load programs from files into Interactive Atlast. Suppose we want to investigate the behaviour of Leibniz' famous 1673 series that converges (achingly slowly) to π. The series is:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots$$

We can create a file, using the text editor of our choice, containing the following:

```
\   Series approximations of Pi

\   Leibniz: pi/4 = 1 - 1/3 + 1/5 - 1/7 ...

: leibniz      ( n -- fpi )
    1.0 1.0
    4 pick 1 do
        2.0 f+  \ denom += 2
        2dup
        i 1 and if
            fnegate
        then
        1.0 2swap f/
        2rot f+
        2swap
    loop
    2drop
    rot drop
    4.0 f*
;

\   Reference value of Pi

1.0 atan 4.0 f* 2constant pi

\ Calculate and print error

: pierr
    pi f- fabs f. cr
;
```

If this seems like gibberish, don't worry! Remember the first time you looked at a Lisp or C program. If you want to decode some of the structure of this program before learning the language, refer to the definitions of Atlast primitives at the back of this manual, remember that Atlast is a reverse Polish stack language, and note that “\” is a comment delimiter that causes the rest of the line to be ignored and that “(” is a comment delimiter that ignores all text until the next “)”.

If this file is saved as `leibniz.atl`, we can load the program into Interactive Atlast with the command:

```
atlast -ileibniz
```

Atlast will compile the program in the file, report any errors, and if no errors are found enter the interactive interpretation mode. The definition of `leibniz` performs the number of iterations specified by the number on the top of the stack and leaves the resulting series approximation to π on the top of the stack.

We can play with this definition as follows:

```
% atlast -ileibniz
10 leibniz f.
3.04184 -> 100 leibniz f.
3.13159 -> 1000 leibniz f.
3.14059 -> 10000 leibniz f.
3.14149 ->
```

Well, we can see it's converging, but not very fast. Since we can define new compiled words on the fly, let's improvise a definition that will print the value and its error for increments of 10000 iterations, then run that program. Continuing our session above:

```
-> : itest 0 do i 1+ 10000 * dup .
:> leibniz 2dup f. pierr  loop ;
-> 5 itest
10000 3.14149 0.0001
20000 3.14154 5e-05
30000 3.14156 3.33333e-05
40000 3.14157 2.5e-05
50000 3.14157 2e-05
-> ^D
%
```

As you can see (even if you don't understand), we've mixed compiled code, interpreted code, and on-the fly definition of new compiled functions in a seamless manner.

You can also run an Atlast program in batch mode simply by specifying its name on the `atlast` command line. If, for example, you added the lines:

```
\   Run iteration vs. error report

: itest
    0 do
        i 1+ 10000 * dup . leibniz
        2dup f. pierr
    loop
;

10 itest
```

to the end of the `leibniz.atl` file, creating a new file called `leibbat.atl`, you could run the program in batch mode as follows:

```
% atlast leibbat
```

```
10000 3.14149 0.0001
20000 3.14154 5e-05
30000 3.14156 3.33333e-05
40000 3.14157 2.5e-05
50000 3.14157 2e-05
60000 3.14158 1.66667e-05
70000 3.14158 1.42857e-05
80000 3.14158 1.25e-05
90000 3.14158 1.11111e-05
100000 3.14158 1e-05
%
```

(By the way, as is apparent, this is clearly no way to compute π! Try this, instead, if you're serious about pumping π.)

```
\   Tamura-Kanada fast Pi algorithm

2variable a
2variable b
2variable c
2variable y

: tamura-kanada ( n -- fpi )
    1.0 a 2!
    1.0 2.0 sqrt f/ b 2!
    0.25 c 2!
    1.0
    rot 1 do
        a 2@ 2dup y 2!
        b 2@ f+ 2.0 f/ a 2!
        b 2@ y 2@ f* sqrt b 2!
        c 2@ 2over a 2@ y 2@ f-
        2dup f* f* f- c 2! 2.0 f*
    loop
    2drop
    a 2@ b 2@ f+ 2dup f* 4.0 c 2@ f* f/
;
```

Debugging

As befits an interactive language, Atlast provides debugging support. You can trace through the execution of a program word by word by enabling the TRACE facility. To turn tracing on, enter the sequence:

```
1 trace
```

If you've loaded a definition of the factorial function as follows:

```
: factorial
        dup 0= if
           drop 1
        else
           dup 1- factorial *
        then
;
```

and execute it under trace, you'll see output as follows:

```
% atlast -ifact
-> 1 trace
-> 3 factorial .

Trace: FACTORIAL
Trace: DUP
Trace: 0=
Trace: ?BRANCH
Trace: DUP
Trace: 1-
Trace: FACTORIAL
Trace: DUP
Trace: 0=
Trace: ?BRANCH
Trace: DUP
Trace: 1-
Trace: FACTORIAL
Trace: DUP
Trace: 0=
Trace: ?BRANCH
Trace: DUP
Trace: 1-
Trace: FACTORIAL
Trace: DUP
Trace: 0=
Trace: ?BRANCH
Trace: DROP
Trace: (LIT) 1
Trace: BRANCH
Trace: EXIT
Trace: *
Trace: EXIT
Trace: *
Trace: EXIT
Trace: *
Trace: EXIT
Trace: . 6 -> ^D
%
```

You can turn off tracing with "0 trace".

When an error occurs, a walkback is normally printed that lists the active words starting with the one in which the error occurred, proceeding through levels of nesting to the outermost, interpretive level. If the `WALKBACK` package is configured (see page 18), the walkback is printed by default. You can disable it with “`0 walkback`”. Here is a sample error walkback report:

```
% atlast -ileibniz
-> leibniz
Stack underflow.
Walkback:
   ROT
   LEIBNIZ
->
```

Integrating Atlast

Unlike most languages, Atlast is not structured as a main program; it is a subroutine. You can invoke it when and where you like within your application, providing as much or as little programmability as is appropriate. Before we get into the details of the interface between an application and Atlast, it’s worth showing, by example, just how simple a program can be that accesses all the facilities of Atlast mentioned so far. The following main program, linked with the Atlast object module, constitutes a fully-functional interactive Atlast interpreter. It lacks the refinements of Interactive Atlast such as console break processing, batch mode, loading definition files, prompting with compilation state, and the like, but any program that Interactive Atlast will run can be run by this program, if submitted to it by input redirection.

```
#include <stdio.h>
#include "atlast.h"
int main()
{
    char t[132];
    atl_init();
    while (printf("-> "),
           fgets(t, 132, stdin) != NULL)
        atl_eval(t);
    return 0;
}
```

Configuring `atlast.c`

The first step in integrating Atlast is building a suitable version of `atlast.c` that can be linked with your application. In order to do this, you must choose the modes with which you wish Atlast built. These modes are normally specified by compile-time definitions supplied on the C compiler call line. Unless you request individual configuration of Atlast subpackages, a fully functional version of Atlast will be built. In that case, you need only be concerned with the settings of the following compile-time variables.

`ALIGNMENT`. If double precision floating point numbers must be aligned on 8 byte boundaries in memory, define `ALIGNMENT`. If not defined, Atlast assumes that 4 byte alignment is adequate for these numbers. (Conditional code in `atldef.h` attempts to define `ALIGNMENT` on processors which require it, but its tests may omit your machine.)

`COPYRIGHT`. If you require a statement of the the public domain status of Atlast to be embedded into the binary program, define this variable. Otherwise, leave it undefined and save a few bytes.

`EXPORT`. If you are simply invoking Atlast as a macro engine and do not require access to its internal data structures, leave `EXPORT` undefined. If your program adds application-specific primitives to Atlast (as most do), define `EXPORT` and include the file `atldef.h` in all modules that require that access. The stack, return stack, and heap pointers will be made external, names of internal symbols within Atlast will be redefined to special names beginning with `atl__` to avoid conflicts with your program, and additional interface code is enabled to provide your primitives full access to the Atlast runtime environment.

`MEMSTAT`. If you want to enable the runtime memory usage monitor, accessible from the `MEMSTAT` primitive or the `atl_memstat()` function call, define `MEMSTAT`.

`NOMEMCHECK`. To disable all runtime stack, heap, and pointer checking, define `NOMEMCHECK`. This will yield a dramatic increase in execution speed, but should be enabled only in closed applications after you’re sure all the bugs are securely in hiding. When built with `NOMEMCHECK`, an Atlast program is no more secure than a pointer-mad C program.

`READONLYSTRINGS`. When the `WORDSUSED` package (see page 18) is enabled, Atlast keeps track of which primitive and user-defined words are used in a program, allowing you to determine which packages are required and whether your tests have invoked all of the words you have defined. This is done by setting a flag in the word definition which, for built-in primitive words, involves modifying a C constant string. If your C language implementation does not permit this, define `READONLYSTRINGS`, which will copy the predefined words to a dynamically allocated buffer which may be modified. Note that this is done only if the `WORDSUSED` package is enabled.

When building Atlast on MS-DOS or OS/2, you must use a large data model (32 bit data addresses). Atlast treats all integers as 32 bits and assumes that data pointers are at least that long. Attempting to build with 16 bit data addresses will cause compile errors that indicate violation of design assumptions.

Initialising: `atl_init`

Before your application makes any other calls to Atlast, you must call `atl_init` to initialise its dynamic storage and create the data structures used to evaluate Atlast expressions.

To initialise Atlast with the default memory configuration, just call:

```
atl_init();
```

The stack, return stack, heap, and initial dictionary are created and Atlast is prepared for execution. You can adjust the size of the memory allocated by Atlast by setting the following variables (defined in `atlast.h`) before calling `atl_init`.

`atl_stklen` Evaluation (data) stack length. Expressed as a number of 4 byte stack items. Default 100.

`atl_rstklen` Return stack length. Expressed as a number of 4 byte return stack pointer items. Default 100.

`atl_heaplen` Heap length. Specified as a number of 4 byte stack items. Default 1000.

`atl_ltempstr` Temporary string length. Gives the length of the buffers used to hold temporary strings entered in interpretive mode and created by certain primitives. Default 256.

`atl_ntempstr` Number of temporary strings. Specifies the number of temporary strings. Temporary strings are used in rotation; if more than `atl_ntempstr` are used without storing out the oldest result, it will be overwritten. Default 4.

Applications can allow Atlast programs they load to override default memory allocation specifications with *prologue statements*. See page 17 for details. Deeply embedded applications, such as those programmed into ROMs, may wish to assign the Atlast dynamic storage areas to predefined areas of memory instead of requesting them with `malloc()`. If the base address pointer of an area is set nonzero before `atl_init` is called, the address specified will be used for that region; no buffer will be allocated. If you take advantage of this facility, please read the code for `atl_init()` in `atlast.c` carefully and make sure the storage you supply is as long as the various length cells specify. Note in particular that the system state word, temporary string buffers, and heap are consolidated into one contiguous area of memory.

Evaluating: `atl_eval`

To evaluate a string containing Atlast program text, call:

stat `= atl_eval(`*string*`);`

where *string* is a string containing the text to be evaluated and *stat* is an integer giving the status of the evaluation. Mnemonics for evaluation status codes are defined in `atlast.h`, and have the following meanings:

`ATL_SNORM`	No error
`ATL_STACKOVER`	Stack overflow
`ATL_STACKUNDER`	Stack underflow
`ATL_RSTACKOVER`	Return stack overflow
`ATL_RSTACKUNDER`	Return stack underflow
`ATL_HEAPOVER`	Heap overflow
`ATL_BADPOINTER`	Bad heap pointer
`ATL_UNDEFINED`	Undefined word
`ATL_FORGETPROT`	Attempt to `FORGET` protected symbol
`ATL_NOTINDEF`	Compiler word outside definition
`ATL_RUNSTRING`	Runaway string
`ATL_RUNCOMM`	Runaway comment in file
`ATL_BREAK`	Asynchronous break signal received
`ATL_DIVZERO`	Attempt to divide by zero

In addition to these status codes, a program that calls `atl_eval` may determine the current state of Atlast by examining external variables. If a multi-line comment awaiting termination with a ")" is active, atl_comment will be nonzero. If the definition of a word (colon definition) is currently pending, the variable state (accessible only if EXPORT is defined and atldef.h is included) will be nonzero.

Loading files: atl_load

To load an entire file containing Atlast program text, call:

```
stat = atl_load(file);
```

where *file* is a C file descriptor (type FILE *) designating the file, currently open for input and positioned before the first byte of the Atlast program to be loaded. The program is read, and *stat* is the status resulting from loading and executing the Atlast program in that file. The status codes are the same as those given above for the atl_eval function. The atl_load function reads text files in any of the end of line conventions recognised by AutoCAD; ASCII files in any of these formats may be loaded by any implementation of Atlast. If the host system requires binary files to be identified at open time, files containing Atlast programs to be loaded with atl_load should be opened in *binary* mode, even though they nominally contain ASCII text. Binary mode permits correct interpretation of all the end of line delimiters accepted by AutoCAD.

The atl_load function uses atl_mark to save the runtime status before loading the file. If an error occurs, it attempts to restore the *status quo ante* by performing an atl_unwind. If the file loaded included interpretive mode code that modified preexisting objects on the heap, those changes will not be reversed if an error occurs whilst loading the file.

Marking: atl_mark

Applications may wish to undertake a series of Atlast operations which might result in a runtime evaluation error. In that event, the application will normally want to undo definitions made by the program that errored. To mark one's place before embarking upon a potentially perilous Atlast program, use:

```
atl_statemark mk;
atl_mark(&mk);
```

The current position of the stack, return stack, heap, and dictionary are saved in the atl_statemark structure. A subsequent atl_unwind call will roll each of those dynamic storage areas back to the position at the designated atl_mark.

Reversing changes: atl_unwind

To roll back all changes to the stack, return stack, heap allocation, and dictionary to the state saved in an atl_statemark object with atl_mark, call:

```
atl_statemark mk;
atl_unwind(&mk);
```

The allocation pointers for all the storage areas are reset to their positions at the time atl_mark was called, but changes to heap variables made by storing through pointers after the atl_mark are not reversed.

Asynchronous break: atl_break

Interactive applications of Atlast must allow the user to escape infinite loops and other accidentally initiated lengthy computations. If the system provides a facility for responding to user interrupt requests, Atlast allows execution of programs under its control to be terminated through the atl_break mechanism.

If BREAK is defined at compile time, the atl_break() function and support for asynchronous break is enabled. When the application receives an asynchronous break, it should call atl_break() to notify the currently running Atlast program of the break signal. If no Atlast program is running at the time of the signal, no harm is done. The application break routine should always call atl_break() rather than try to determine whether Atlast is active. If an Atlast program was executing at the time of the break signal, the application that invoked it, whether by atl_eval, atl_load, or atl_exec, will be notified of the abnormal termination by the return of the ATL_BREAK status.

The atl_break function simply sets a flag examined by the inner loop of the Atlast evaluator; it does not actually terminate execution. Consequently, it may safely be called at any time, even from hardware interrupt service routines.

Showing memory status: `atl_memstat`

In the final stage of optimising an application incorporating Atlast for shipment, one may wish to adjust the memory allocation parameters to eliminate wasted space while providing reasonable margins for user extensions after shipment. To set the parameters wisely, one must know the baseline memory usage of the application. If `atlast.c` is built with `MEMSTAT` defined, this can be obtained either by executing the `MEMSTAT` primitive within the Atlast program or by calling the `atl_memstat` function at an opportune time within the application. In either case, a memory usage report similar to the following example is written to the standard output stream.

```
                      Memory Usage Summary

                    Current   Maximum   Items     Percent
Memory Area          usage     used   allocated  in use
 Stack                 0         9       100        0
 Return stack          0         4       100        0
 Heap                227       227      1000       22
```

Note: to use any of the following functions, you must compile `atlast.c` and the modules that call them with `EXPORT` defined, and you must include the header file `atldef.h` in files that call them.

Looking up words: `atl_lookup`

Your application can look up words in the Atlast dictionary, using the same search order as the interpreter would, with the call:

```
dictword *dw;
char *name;

dw = atl_lookup(name);
```

Since Atlast names are matched regardless of whether letters in them are upper or lower case, the *name* may contain any combination of upper and lower case letters. If the word is defined, its dictionary entry is returned. The `dictword` structure is defined in `atldef.h`. If the word is not defined, `NULL` is returned. There may be multiple nested definitions of a word; if this is the case, only the most recent definition (the active definition) is returned. There is no way, using `atl_lookup` alone, to locate hidden definitions.

Accessing a word's text: `atl_body`

An Atlast word definition consists of several components, including its name and the C-coded method that implements it. Of most interest to applications that intercommunicate with Atlast is the *body* of the word. For a variable or constant, this is the storage that contains the word's value. To obtain the body address of a dictionary item returned by `atl_lookup` or created by `atl_vardef` (see below), use `atl_body`. The call:

```
dictword *dw;
stackitem *si;

si = atl_body(dw);
```

places the body address of dictionary item *dw* into variable *si*. If you wish to store a data type into the body of the Atlast word other than the default of `stackitem` (defined as `long`), cast the pointer to the correct pointer type. See the `atl_vardef` sample below for an example of a floating point variable being created and initialised using `atl_body`.

Defining variables: `atl_vardef`

Shared variables are a convenient way of intercommunicating between a host application and Atlast. By making the application's state visible to and changeable by the Atlast program, the program is given the information it needs and the power to direct the application. A shared variable is an Atlast variable defined by the application, the address of which is known both to Atlast (via the dictionary), and to the application (by a pointer returned when the shared variable is created). To create a shared variable, call:

```
dictword *var;

var = atl_vardef(name, size);
```

where *name* is a character pointer giving the name of the variable to be created and *size* is an integer specifying its size in bytes. Note that to create a normal Atlast integer variable *size* should be 4; for a floating point variable, *size* should be 8 bytes. Storage for the variable is reserved on the Atlast heap. If insufficient heap space is available to create the variable `NULL` is returned. Otherwise, the address of the variable's dictionary entry is returned. **Beware:** the dictionary entry *is not* the storage address of the variable's value. To obtain that address, call `atl_body`, described above.

For example, we can create a floating point variable containing a crummy approximation of π with the sequence:

```
dictword *pi;

pi = atl_vardef("Pi", sizeof(double));
if (pi == NULL) {
    printf("Can't atl_vardef PI.\n");
} else {
    *((double *) atl_body(pi)) =
        3.141596235;
}
```

We could then print the value with an Atlast program run under that application with:

```
pi 2@ f.
```

Executing words: `atl_exec`

If you've obtained the dictionary address of an Atlast word definition, your application can execute it with the sequence:

```
dictword *dw;
int stat;
stat = atl_exec(dw);
```

The status codes returned in *stat* are identical to those returned by `atl_eval`. The distinction between `atl_eval` and `atl_exec` is subtle, but important—it can make a big difference in the performance of your application. If you know the name of an Atlast word, you can execute it either by passing a string containing its name to `atl_eval` or by saving its dictionary address in a variable and executing the word directly from the dictionary address with `atl_exec`. The results of these two operations are identical, but when you pass a string to `atl_eval`, Atlast is forced to scan the string, parse its contents into the token denoting the word, look that word up in the dictionary, and only then execute the word. You can bypass all these nonproductive and time consuming preliminaries if you know the word's dictionary address and use `atl_exec`.

Creative use of `atl_lookup` and `atl_exec` provide one of the most powerful ways for Atlast to enrich an application. If you create an application to perform a relatively well-defined task, you can, before entering its main processing loop, inquire with `atl_lookup` whether the user has defined a series of words specified by the application. If so, their dictionary addresses are saved in pointers in the application code. Then, as the application executes, at each step where the user might want to interpose his own processing or replace the application's default processing with his own method, the application merely tests whether the word associated with that step has been defined in the Atlast program and, if so, runs it with `atl_exec`. If the default processing that would otherwise occur is made available as an Atlast primitive with `atl_primdef` (see below), it is extremely easy for the Atlast program to examine the data at the point it has been "hooked," perform any special processing it wishes, or inherit the default processing simply by running the primitive that does it. If the user has not requested special processing, the cost to the application to provide that opportunity is one pointer comparison against `NULL`. Compared with the benefits of open architecture, this is a small price indeed.

You can pass arguments to the definition you're invoking with `atl_exec` either by storing them in shared variables created with `atl_vardef` or, usually the best approach, pushing them on the stack before executing the definition. See the discussion of `atl_primdef` below for information on access to the stack from C.

Defining primitives: `atl_primdef`

Most of the power of Atlast derives from the ease with which C coded primitives can be added to the language. Once integrated, they may be used in conjunction with the looping, conditional execution, and other facilities already present. Atlast has been deliberately designed to make the addition of primitives simple and safe: nothing like the peril-filled nightmare of adding a function to AutoLISP. Still, to extend any language you need to learn your way around its memory architecture and control structure. So, listen up, walk through the examples, and before long you'll be adding primitives like a pro.

An Atlast primitive is a C function. When the primitive is executed, that function is called and may do whatever it likes. A primitive can be as simple as one that discards the top item on the stack, or as complex as one that prepares a ray-traced bitmap from a three dimensional geometric model. Most primitives communicate with one another via the *stack*. Some primitives also access variables stored on the *heap*. Finally, a very few primitives manipulate data stored on the *return stack*, which Atlast uses to track the nesting of execution. A user-defined primitive will rarely need to access the return stack. Definitions in `atldef.h` simplify access to each of these areas of memory. Let's look at them one by one.

Accessing the stack

The stack pointer variable is called `stk`, and always points to the next available `long` stack item. Primitives rarely reference `stk` directly, since it is usually far more convenient to use definitions that hide the complexity of indexing the stack. The following tools are provided for access to the stack.

`Sl(`*n*`)` Before you access any items on the stack, you must check that the stack actually contains at least as many items as you'll be using. If not, a *stack underflow* must be reported. At the start of your primitive, simply use the statement "`Sl(`*n*`);`", where *n* is the number of stack items you'll be referencing. If you use the topmost two stack items, `S0` and `S1`, you'd use `Sl(2);`. It's important that you use the definition rather than check the stack limit directly; if you later build your application with stack checking off, the `Sl()` statement will generate no code, automatically configuring your primitive for maximum speed.

`So(`*n*`)` Before you push any new items onto the stack, you must check that the stack will not overflow the area allocated to it when those items are added. If it would, a *stack overflow* must be reported. At the start of your primitive, simply use the statement "`So(`*n*`);`", where *n* is the number of new stack items you'll be pushing. If you are adding one new integer item to the stack, use "`So(1);`". It's important that you use the definition rather than check the stack limit directly; if you later build your application with stack checking off, the `So()` statement will generate no code, automatically configuring your primitive for maximum speed.

`S0`–`S5` The definitions `S0`, `S1`,. . . `S5` provide direct access to the top 6 integer stack items. `S0` is the top item on the stack, `S1` is the next item, and so on. These definitions may be used on either the left or right side of an assignment.

`Pop` Used as a statement, "`Pop;`", discards the topmost item from the stack.

`Pop2` Used as a statement, "`Pop2;`", discards the topmost two items from the stack.

`Npop(`*n*`)` Discards the top *n* items from the stack.

`Push` Used on the left side of an assignment, stores the value on the right side into the next free stack item and increments the stack pointer.

`Realsize` For primitives that use floating point numbers, `Realsize` gives the number of stack items occupied by one floating point number. A primitive that expects two floating point arguments on the stack and will leave them there, adding one new floating point result would begin "`Sl(2 * Realsize); So(Realsize);`".

`REAL0`–`REAL2` These definitions provide read access to the topmost three floating point numbers on the stack. The stack cells are automatically cast to type `double`. It is *essential* that you access floating point values this way—some computers require that `double`s be aligned on 8 byte boundaries, and the `REAL`*n* definitions automatically align the variable if the machine requires it.

`SREAL0(`*f*`)`, `SREAL1(`*f*`)` These definitions, used as functions, store their floating point arguments into the topmost (`SREAL0`) and next (`SREAL1`) floating point items on the stack. Because of the possible need to compensate for machine alignment restrictions, the `REAL`*n* definitions cannot be used on the left side of an assignment; use these functions instead.

`Realpop` Pops the topmost floating point value from the stack. Equivalent to `Npop(Realsize)`.

`Realpop2` Pops the two topmost floating point values from the stack. Equivalent to `Npop(2 * Realsize)`.

He said this was easy! Please bear with me—all of this is far simpler (and more compact) to use than it is to explain. If you can't stand it, skip ahead to the sample primitive definitions and see for yourself. O.K., welcome back. Probably 95% of all the primitives you'll add to ATLAST will confine themselves to accessing the stack. Heap and return stack access is far less frequent (and may indicate poor design). In any case, if you need to do it, here's how.

Accessing the heap

The *heap* is a pool of memory used to allocate static objects. Most heap is allocated by ATLAST *defining words*, such as `VARIABLE`, `CONSTANT`, and the `:` used to define new executable words, themselves stored on the heap. The ability to create defining words for new data types directly in ATLAST is one of its most powerful features and reduces the need to manipulate the heap from user primitives. The heap is accessed through a set of definitions similar to those used for the stack. The heap pointer itself is named `hptr`, but will rarely be referenced explicitly.

`Ho(`*n*`)` Before you store any new data on the heap, you must verify that doing so would not cause the heap to grow past its assigned maximum size. This event is called a *heap overflow*, and the `Ho(`*n*`)` function checks for it and terminates execution should overflow occur. The number *n* is the amount of heap you propose to allocate, *in terms of stack items*, each of four bytes. If you wish to allocate a number expressed in bytes, you must round it up to the next larger multiple of four. A portable way to do this is to use the expression: `((`*x*` + (sizeof(stackitem) - 1)) / sizeof(stackitem))` where *x* is the number of bytes of heap you require. If you configure stack and heap checking off for maximum performance, `Ho(`*n*`)` generates no code.

`Hpc(`*ptr*`)` Heap storage is normally accessed via pointers passed on the stack. Since the stack contains many other types of data, accidentally using a non-pointer as a heap address could be catastrophic. Before using any value as a pointer to the heap, call `Hpc(`*ptr*`)` where *ptr* is the pointer. If the pointer is not within the heap, a *bad pointer* error will be reported and execution terminated. If you configure stack and heap checking off, `Hpc(`*ptr*`)` generates no code.

`Hstore` Used on the left of an assignment, stores the `long` value on the right side into the next available heap cell and advances the heap allocation pointer.

Accessing the return stack

The return stack remembers the point at which one definition invoked another, tracks loop control indices, and stores other items internal to the evaluator. Messing with the return stack is generally a very bad idea. This information is presented not so much to encourage you to use the return stack as for completeness and to document the code within `atlast.c` that maintains it. The stack pointer variable is called `rstk`, and always points to the next available return stack item. Return stack items have a type of `**dictword` (got that?), which is also `typedef`ed to `rstackitem`.

Primitives rarely reference `rstk` directly, since it is usually far more convenient to use definitions that hide the complexity of indexing the return stack. The following tools provide access to the return stack.

`Rsl(`*n*`)` Before you access any items on the return stack, you must check that the return stack actually contains at least as many items as you'll be using. Otherwise, a *return stack underflow* must be reported. At the start of your primitive, simply use the statement "`Rsl(`*n*`);`", where *n* is the number of return stack items you'll be referencing. If you use the topmost two items, `R0` and `R1`, you'd use `Rsl(2);`. It's important that you use the definition rather than check the return stack limit directly; if you later build your application with stack checking off, the `Rsl()` statement will generate no code, automatically configuring your primitive for maximum speed.

`Rso(`*n*`)` Before you push any new items onto the return stack, you must check that the return stack will not overflow the area allocated to it when those items are added. If it would, a *return stack overflow* must be reported. At the start of your primitive, simply use the statement "`Rso(`*n*`);`", where *n* is the number of new return stack items you'll be pushing. If you are adding one new item to the return stack, use "`Rso(1);`". It's important that you use the definition rather than check the return stack limit directly; if you later build your application with stack checking off, the `Rso()` statement will generate no code, automatically configuring your primitive for maximum speed.

`R0`–`R2` The definitions `R0`, `R1`, and `R2` provide direct access to the top three return stack items. `R0` is the top item on the return stack, `R1` is the next item, and `R2` is the third item. These definitions may be used on either the left or the right side of an assignment.

`Rpop` Used as a statement, "`Rpop;`", discards the topmost item from the return stack.

`Rpush` Used on the left side of an assignment, stores the value on the right side into the next free return stack item and increments the return stack pointer.

Coding primitive functions

Each primitive word you define is implemented by a C function declared as "`static void`". The header file `atldef.h` defines "`prim`" as this type to more explicitly identify primitive implementing functions.

As an example of a simple primitive, let's add the ability to obtain the date and time in Unix format and to extract the hours, minutes, and seconds from the Unix date word. We'll add two new primitive functions to ATLAST: `TIME`, which leaves the number of seconds since midnight on January 1, 1970 on the top of the stack, and `HHMMSS` which, given the value returned by `TIME`, leaves the hours, minutes, and seconds represented by that time in the three top stack locations, with the seconds at the top.

Here is the C function that implements the TIME primitive word:

```
prim ptime()
{
    So(1);
    Push = time(NULL);
}
```

Since we're placing one new word on the stack, we call So(1) to check for stack overflow. That accomplished, we simply use Push on the left side of the assignment to store the long time word returned by the Unix-compatible time() function (which is supported by most non-Unix C libraries, as well).

The function for our HHMMSS primitive is more complicated, but not much. It uses the Unix-compatible localtime() function which, passed a pointer to a word containing a time in the format returned by time(), returns a pointer to an internal static structure with fields that give the day, month, year, hour, minute, second, etc. represented by that time. The primitive definition is:

```
prim phhmmss()
{
    struct tm *lt;

    Sl(1);
    So(2);
    lt = localtime(&S0);
    S0 = lt->tm_hour;
    Push = lt->tm_min;
    Push = lt->tm_sec;
}
```

This primitive expects one argument (the time word) on the stack, so it begins with Sl(1) to verify that it is present. It will replace that value with the hours and add two new items to the stack for the minutes and seconds, so it next uses So(2) to ensure those additions won't cause the stack to overflow. Now it can get down to business. It calls localtime(), passing the address of the first stack item (the time word), then stores the hours back into that word and uses Push twice to add the minutes and seconds.

Once the primitive functions are coded, the primitives are actually added to Atlast by listing them in a primitive definition table and registering that table with Atlast by calling the atl_primdef function. The primitive definition table for our two new primitives is as follows:

```
static struct primfcn timep[] = {
    {"0TIME",   ptime},
    {"0HHMMSS", phhmmss},
    {NULL,      (codeptr) 0}
};
```

The primfcn structure is declared in atldef.h. You may list as many primitives in the table as you wish. The end of the table is marked by an entry with NULL instead of a primitive name. For each primitive you define, make an entry with two components: the first a string with the first character "0" if the primitive is a normal word and "1" if it is a compile-time immediate word, the balance of which is the name of the primitive with all letters upper case. The second component is the name of the function that implements the primitive. The primitives in the table are defined by calling atl_primdef, passing the address of the table as follows:

```
atl_primdef(primt);
```

(Subtle note for MS-DOS users: to save memory, Atlast uses the actual static strings you declare in the primitive table as part of the dictionary entries it creates. Since the Atlast dictionary will contain pointers to these compiled-in strings, you must not place the data for the primitive table in an overlay which might be swapped out when Atlast later attempts to search the dictionary. If your program does not overlay its data segment, you need not worry about this.)

You can call atl_primdef any time after you've called atl_init, and you can call it as many times as you like with different primfcn tables. If a name in a primfcn table duplicates the name of a built-in Atlast primitive or a primitive defined by an previous call on atl_primdef, the earlier definition will be hidden and inaccessible.

With these new primitives installed, we can now try them out interactively from Atlast.

```
% atlast
-> time .
634539503 -> time .
634539505 -> time .
634539508 -> time .s
Stack: 634539512 -> hhmmss
-> .s
Stack: 20 58 32 -> clear time hhmmss .s
Stack: 20 58 44 -> clear
-> time hhmmss .s
Stack: 20 58 52 -> ^D
%
```

Everything seems to be behaving as we intended. Our new primitives work!

Finally, let's look at a more complicated primitive, one involving floating point. Turning again to the Leibniz series for π, here is the C language definition of a primitive function to evaluate it. The function is compatible with the one we previously implemented in Atlast: it expects the number of terms on the top of the stack and returns the approximation of π as a floating point value in the two top stack items.

```
prim pleibniz()
{
    long nterms;
    double sum = 0.0,
           numer = 1.0,
           denom = 1.0;

    Sl(1);
    nterms = S0;
    Pop;

    So(Realsize);
    Push = 0;
    Push = 0;
    while (nterms-- > 0) {
        sum += numer / denom;
        numer = -numer;
        denom += 2.0;
    }
    SREAL0(sum * 4.0);
}
```

This function begins by verifying with `Sl(1)` that its term count argument is present on the stack. It loads that argument, referenced as `S0`, and saves it in the loop count, `nterms`. The iteration count is then discarded from the stack with `Pop`. Next, `So(Realsize)` verifies that the stack will not overflow when the real result is pushed (recall that `Realsize` is the number of stack items per floating point result—this is always two, but using the definition makes for more readable code). We then immediately count on `Realsize` being two as we use two `Push` operations to allocate the stack space for the result and clear it to zero. That done, the function falls into the loop that sums the requested number of terms of the series. Finally, `SREAL0()` is used to store the result into the top floating point value on the stack: the one we created with the two `Push`es.

This primitive is declared and registered with Atlast with the sequence:

```
static struct primfcn pip[] = {
    {"0LEIBNIZ", pleibniz},
    {NULL, (codeptr) 0}
};
atl_primdef(pip);
```

With a C coded primitive implementation, we can explore the outer reaches of this awful series. For example, here it's used to print the error after the first half million terms.

```
% atlast
-> 2variable pi
-> 1.0 atan 4.0 f* pi 2!
-> pi 2@ f. c
3.14159
-> 500000 leibniz pi 2@ f- f. cr
-2e-06
-> ^D
%
```

As you can see from the brevity and straightforwardness of these sample primitives, there's nothing complicated or difficult about adding a primitive to Atlast. The overhead in executing a primitive function from Atlast rather than calling it from a C program is a matter of a few instructions. If you need guidance in implementing primitives that interact with Atlast in more intricate ways, the best source of information is the source code of `atlast.c`; find a standard primitive with arguments and results similar to the one you're planning to add, and look up its implementing function. That should abate any confusion about the fine points of stack and heap manipulation.

Package configuration

In addition to the global configuration parameters described on page 8, you can choose precisely which components of Atlast are included when building a version for your application by creating a custom configuration file named `custom.h`, then compiling `atlast.c` with the `-DCUSTOM` compiler flag. A custom configuration file has the following format:

`#define INDIVIDUALLY`
`#define` $\mathit{Package}_1$
`#define` $\mathit{Package}_2$
$\vdots$
`#define` $\mathit{Package}_n$

The *Package$_n$* definitions select which ATLAST subpackages you wish included in your application. The individual subpackages are described in the following paragraphs. The WORDSUSED and WORDSUNUSED primitives, available as part of the WORDSUSED package, let you determine which primitives are used within an ATLAST program and, consequently, which packages are required to execute it.

The ARRAY package. Provides declaration of n dimensional arrays of arbitrary data types and runtime subscript calculation for such arrays. Primitives: ARRAY.

The BREAK package. Enables asynchronous break processing via the atl_break function. Disabling this package saves an insignificant amount of memory but increases execution speed by about 10%. Primitives: none.

The COMPILERW package. Enables primitives used to define new compiler words. Primitives: [COMPILE], LITERAL, COMPILE, <MARK, <RESOLVE, >MARK, >RESOLVE.

The CONIO package. Enables primitives that display interactive output. These primitives may be disabled in applications that provide no interaction with the user. Primitives: ., ?, CR, .S, .", .(, TYPE, WORDS.

The DEFFIELDS package. Enables low level primitives used to manipulate dictionary items. These primitives are rarely used except in very ambitious language extensions coded in ATLAST. Primitives: FIND, >NAME, >LINK, BODY>, NAME>, LINK>, N>LINK, L>NAME, NAME>S!, S>NAME!.

The DOUBLE package. Enables double word operations. These operations can be used with any stack data, but are heavily used in floating point code, since floating point numbers occupy pairs of stack items. Primitives: 2DUP, 2DROP, 2SWAP, 2OVER, 2ROT, 2VARIABLE, 2CONSTANT, 2!, 2@.

The FILEIO package. Enables the C language-like file primitives. If your application does not require access to files, this package may be disabled. Primitives: FILE, FOPEN, FCLOSE, FDELETE, FGETS, FPUTS, FREAD, FWRITE, FGETC, FPUTC, FTELL, FSEEK, FLOAD. In addition, FILE variables STDIN, STDOUT, and STDERR are defined, automatically bound to the Unix I/O streams with the same names.

The MATH package. Enables the mathematical functions. MATH can be enabled only if REAL is also enabled. Primitives: ACOS, ASIN, ATAN, ATAN2, COS, EXP, LOG, POW, SIN, SQRT, TAN.

The MEMMESSAGE package. Controls whether messages are printed when runtime errors (such as stack overflow and underflow, bad pointers, etc.) occur. Disabling these messages doesn't save time or significant memory: it's intended for deeply embedded applications where returning the error status to the caller of atl_eval or atl_exec is all the error notification that is appropriate. Primitives: none.

The PROLOGUE package. The amount of memory allocated to the stack, return stack, heap, and temporary string buffers can be controlled by setting the external variables governing those areas as described on page 9. You can allow the ATLAST program text to override the default settings you make by enabling the PROLOGUE package. If this package is enabled, special statements of the form:

\ **area size*

are recognised by the evaluator when encountered before the first line containing executable ATLAST text. To permit processing of the prologue, *do not* explicitly call atl_init; it will be called automatically by atl_eval after the prologue is processed. The following *area* specifications are recognised in the prologue:

STACK Specifies the stack size in terms of long stack items.

RSTACK Specifies the return stack size in items.

HEAP Specifies the heap size as a number of long stack items.

TEMPSTRL Specifies the length of each temporary string buffer in characters.

TEMPSTRN Specifies the number of temporary string buffers.

The REAL package. Enables floating point operations. If you enable the REAL package, you should also enable the DOUBLE package; without it you won't be able to accomplish much. Primitives: (FLIT), F+, F-, F*, F/, FMIN, FMAX, FNEGATE, FABS, F=, F<>, F>, F<, F>=, F<=, F., FLOAT, FIX.

The SHORTCUTA package. Enables shortcut integer arithmetic operations. Primitives: 1+, 2+, 1-, 2-, 2*, 2/.

The SHORTCUTC package. Enables shortcut integer comparison operations. Primitives: 0=, 0<>, 0<, 0>.

The STRING package. Enables string operations. Primitives: (STRLIT), STRING, STRCPY, S!, STRCAT, S+, STRLEN, STRCMP, STRCHAR, SUBSTR, COMPARE, STRFORM, STRINT, STRREAL. If the REAL package is also enabled, the FSTRFORM primitive is available, as well.

The SYSTEM package. Enables submission of commands in strings to the operating system for execution. This package may be enabled only if the implementation of C used to build ATLAST provides the system() function. Primitives: SYSTEM.

The TRACE package. Enables runtime word execution trace. Primitives: TRACE.

The WALKBACK package. Enables the walkback through nested invocation of words when an error is detected at runtime. Primitives: WALKBACK.

The WORDSUSED package. Enables the collection of information on which words are used and not used by a program, and the primitives that list words used and words not used. This facility allows you to determine, in the development phase of an ATLAST application, which packages are needed and which can be safely dispensed with. Primitives: WORDSUSED, WORDSUNUSED.

Benchmarks

To give a rough idea of the kind of performance you can expect from ATLAST when it is pressed into service for compute-intensive tasks, I tested it against C and AutoLISP with two benchmarks, both involving the computation of square roots.

The first benchmark, CSQRT, calculates the square root of 2 with the iterative Newton-Raphson algorithm used by AutoCAD's HMATH.C module, also used in the AutoLISP sample program SQR.LSP. This benchmark is representative of extremely compute-bound code which represents misuse of a macro language—any such computation should normally be moved into a primitive written in C. Still, it's interesting to know what the worst case is.

The second benchmark, SSQRT, is identical to CSQRT, except that the system math library's sqrt() function is called instead of one coded in the language under test. Since all three languages are calling the same underlying system function, this test demonstrates relative performance in an environment still more compute-bound than a typical macro language application, but one where the language overhead is less than 100%. All of these benchmarks were run on a Sun 3/260 under SunOS 4.0.3, and listings of the benchmark programs are given at the end of this paper. The ATLAST timings were made on a version of ATLAST compiled with the "-O4 -f68881" flags, and stack and heap checking disabled in the ATLAST configuration. The C programs were also compiled with "-O4 -f68881" flags, while the AutoLISP tests were run on a NONPRODUCTION version of Z.0.65 in which AutoLISP was built with "-O -f68881". All timings in the following table have been normalised so that the native C language times are 1.

	C	ATLAST	AutoLISP
CSQRT	1.00	7.41	67.08
SSQRT	1.00	1.00	1.52

Summary and Conclusions

Everything should be programmable. *Everything!* I have come to the conclusion that to write almost any program in a closed manner is a mistake that invites the expenditure of uncounted hours "enhancing" it over its life cycle. Further tweaks, "features," and "fixes" often result in a product so massive and incomprehensible that it becomes unlearnable, unmaintainable, and eventually unusable.

Far better to invest the effort up front to create a product flexible enough to be adapted at will, by its users, to their immediate needs. If the product is programmable in a portable, open form, user extensions can be exchanged, compared, reviewed by the product developer, and eventually incorporated into the mainstream of the product.

It is far, far better to have thousands of creative users expanding the scope of one's product in ways the original developers didn't anticipate—in fact, working for the vendor without pay, than it is to have thousands of frustrated users writing up wish list requests that the vendor can comply with only by hiring people and paying them to try to accommodate the perceived needs of the users. Open architecture and programmability not only benefits the user, not only makes a product better in the technical and marketing sense, but confers a direct economic advantage upon the vendor of such a product—one mirrored in a commensurate disadvantage to the vendor of a closed product.

The chief argument against programmability has been the extra investment needed to create open products. Atlast provides a way of building open products in the same, or less, time than it takes to construct closed ones. Just as no C programmer in his right mind would sit down and write his own buffered file I/O package when a perfectly fine one was sitting in the library, why re-invent a macro language or other parameterisation and programming facility when there's one just sitting there that's as fast as native C code for all but the most absurd misapplications, takes less than 51K with every gew-gaw and optional feature at its command enabled all at once, is portable to any machine that supports C by simply recompiling a single file, and can be integrated into a typical application at a basic level in less than 15 minutes?

Am I proposing that every application suddenly look like FORTH? Of course not; no more than output from PostScript printers looks like PostScript, or applications that run on 80386 processors resemble 80386 assembly language. Atlast is an intermediate language, seen only by those engaged in implementing and extending the product. Even then, Atlast is a chameleon which, with properly defined words, can look like almost anything you like, even at the primitive level of the interpreter.

Again and again, I have been faced with design situations where I knew that I really needed programmability, but didn't have the time, the memory, or the fortitude to face the problem squarely and solve it the right way. Instead, I ended up creating a kludge that continued to burden me through time. This is just a higher level manifestation of the nightmares perpetrated by old-time programmers who didn't have access to a proper dynamic memory allocator or linked list package. Just because programmability is the magic smoke of computing doesn't mean we should be spooked by the ghost in the machine or hesitant to confer its power upon our customers.

Don't think of Atlast as FORTH. Don't think of it as a language at all. The best way to think of Atlast is as a library routine that gives you *programmability*, in the same sense other libraries provide file access, window management, or graphics facilities. The whole concept of "programmability in a can" is odd—it took me two years from the time I first thought about it in connection with The Leto Protocol until I really got my end effector around it and crushed it into submission. I urge you to think about it, play with it, and examine how it will be applied in the Atlast-enhanced programs I will be demonstrating in the near future.

Open is better. Atlast lets you build open programs in less time than you used to spend writing closed ones. Programs that inherit their open architecture from Atlast will share, across the entire product line and among all hardware platforms that support it, a common, clean, and efficient means of user extensibility. The potential benefits of this are immense.

John Walker
Muir Beach, California
January 22–February 11, 1990
4072 lines of code

Atlast Primitives: Alphabetical Reference

`+`	n1 n2 → n3	**n3 = n1 + n2** Adds *n1* and *n2* and leaves sum on stack.	
`-`	n1 n2 → n3	**n3 = n1 − n2** Subtracts *n2* from *n1* and leaves difference on stack.	
`*`	n1 n2 → n3	**n3 = n1 × n2** Multiplies *n1* and *n2* and leaves product on stack.	
`/`	n1 n2 → n3	**n3 = n1 ÷ n2** Divides *n1* by *n2* and leaves quotient on stack.	
`'` *word*	→ caddr	**Obtain compilation address** Places the compilation address of the following word on the stack.	
`,`	n →	**Store in heap** Reserves four bytes of heap space, initialising it to *n*.	
`.`	n →	**Print top of stack** Prints the number on the top of the stack.	`CONIO`
`.(` *str*	→	**Print constant string** Immediately prints the string that follows in the input stream.	`CONIO`
`.S`	→	**Print stack** Prints entire contents of stack.	`CONIO`
`."` *str*	→	**Print immediate string** Prints the string literal that follows in line.	`CONIO`
`:` *w*	→	**Begin definition** Begins compilation of a word named *w*.	
`;`	→	**End definition** Ends compilation of word.	
`<`	n1 n2 → flag	**Less than** Returns −1 if *n1*<*n2*, 0 otherwise.	
`<=`	n1 n2 → flag	**Less than or equal** Returns −1 if *n1*≤*n2*, 0 otherwise.	
`<>`	n1 n2 → flag	**Not equal** Returns −1 if *n1*≠*n2*, 0 otherwise.	
`=`	n1 n2 → flag	**Equal** Returns −1 if *n1*=*n2*, 0 otherwise.	
`>`	n1 n2 → flag	**Greater** Returns −1 if *n1*>*n2*, 0 otherwise.	
`>=`	n1 n2 → flag	**Greater than or equal** Returns −1 if *n1*≥*n2*, 0 otherwise.	
`?`	addr →	**Print indirect** Prints the value at the address at the top of the stack.	`CONIO`
`!`	n addr →	**Store into address** Stores the value *n* into the address *addr*.	
`+!`	n addr →	**Add indirect** Adds *n* to the word at address *addr*.	
`@`	addr → n	**Load**	

Loads the value at *addr* and leaves it at the top of the stack.

`[`	→	**Set interpretive state**		Within a compilation, returns to the interpretive state.
`['] word`	→ caddr	**Push next word**		Places the compile address of the following word in a definition onto the stack.
`]`	→	**End interpretive state**		Restore compile state after temporary interpretive state.
`0<`	n1 → flag	**Less than zero**	`SHORTCUTC`	Returns −1 if *n1* less than zero, 0 otherwise.
`0<>`	n1 → flag	**Nonzero**	`SHORTCUTC`	Returns −1 if *n1* is nonzero, 0 otherwise.
`0=`	n1 → flag	**Equal to zero**	`SHORTCUTC`	Returns −1 if *n1* is zero, 0 otherwise.
`0>`	n1 → flag	**Greater than zero**	`SHORTCUTC`	Returns −1 if *n1* greater than zero, 0 otherwise.
`1+`	n1 → n2	**Add one**	`SHORTCUTA`	Adds one to top of stack.
`1-`	n1 → n2	**Subtract one**	`SHORTCUTA`	Subtracts one from top of stack.
`2+`	n1 → n2	**Add two**	`SHORTCUTA`	Adds two to top of stack.
`2-`	n1 → n2	**Subtract two**	`SHORTCUTA`	Subtracts two from top of stack.
`2*`	n1 → n2	**Times two**	`SHORTCUTA`	Multiplies the top of stack by two.
`2/`	n1 → n2	**Divide by two**	`SHORTCUTA`	Divides top of stack by two.
`2!`	n1 n2 addr →	**Store two words**	`DOUBLE`	Stores the two words *n1* and *n2* at addresses *addr* and *addr*+4.
`2@`	addr → n1 n2	**Load two words**	`DOUBLE`	Places the two words starting at *addr* on the top of the stack
`2CONSTANT x`	n1 n2 →	**Double word constant**	`DOUBLE`	Declares a double word constant *x*. When *x* is executed, *n1* and *n2* are placed on the stack.
`2DROP`	n1 n2 →	**Double drop**	`DOUBLE`	Discards the two top items from the stack.
`2DUP`	n1 n2 → n1 n2 n1 n2	**Duplicate two**	`DOUBLE`	Duplicates the top two items on the stack.
`2OVER`	n1 n2 n3 n4 → n1 n2 n3 n4 n1 n2	**Double over**	`DOUBLE`	Copies the second pair of items on the stack to the top of stack.
`2ROT`	n1 n2 n3 n4 n5 n6 → n3 n4 n5 n6 n1 n2	**Double rotate**	`DOUBLE`	

Atlast Primitives: Alphabetical Reference

		Rotates the third pair on the stack to the top, moving down the first and second pairs.
`2SWAP`	n1 n2 n3 n4 → n3 n4 n1 n2	**Double swap** `DOUBLE` Swaps the first and second pairs on the stack.
`2VARIABLE` *x*	→	**Double variable** `DOUBLE` Creates a two cell (8 byte) variable named *x*. When *x* is executed, the address of the 8 byte area is placed on the stack.
`ABORT`	→	**Abort** Clears the stack and performs a `QUIT`.
`ABORT"` *str*	→	**Abort with message** Prints the string literal that follows in line, then aborts, clearing all execution state to return to the interpreter.
`ABS`	n1 → n2	$\mathbf{n2} = \lvert\mathbf{n1}\rvert$ Replaces top of stack with its absolute value.
`ACOS`	f1 → f2	$\mathbf{f2} = \arccos \mathbf{f1}$ `MATH` Replaces floating point top of stack with its arc cosine.
`AGAIN`	→	**Indefinite loop** Marks the end of an indefinite loop opened by the matching `BEGIN`.
`ALLOT`	n →	**Allocate heap** Allocates *n* bytes of heap space. The space allocated is rounded to the next higher multiple of 4.
`AND`	n1 n2 → n3	**Bitwise AND** Stores the bitwise AND of *n1* and *n2* on the stack.
`ARRAY` *x*	$s_1\ s_2 \ldots s_n$ n esize →	**Declare array** `ARRAY` Declares an array *x* of elements of *esize* bytes each with *n* subscripts, each ranging from 0 to $s_n - 1$
`ASIN`	f1 → f2	$\mathbf{f2} = \arcsin \mathbf{f1}$ `MATH` Replaces floating point top of stack with its arc sine.
`ATAN`	f1 → f2	$\mathbf{f2} = \arctan \mathbf{f1}$ `MATH` Replaces floating point top of stack with its arc tangent.
`ATAN2`	f1 f2 → f3	$\mathbf{f3} = \arctan \mathbf{f1}/\mathbf{f2}$ `MATH` Replaces the two floating point numbers on the top of the stack with the arc tangent of their quotient, properly handling zero denominators.
`BEGIN`	→	**Begin loop** Begins an indefinite loop. The end of the loop is marked by the matching `AGAIN`, `REPEAT`, or `UNTIL`.
`BODY>`	pfa → cfa	**Body to word** `DEFFIELDS` Given body address of word, return the compile address of the word.
`>BODY`	cfa → pfa	**Body address** Given the compile address of a word, return its body (parameter) address.
`BRANCH`	→	**Branch**

Jump to the address that follows in line.

`?BRANCH`	flag →	**Conditional branch** If the top of stack is zero, jump to the address which follows in line. Otherwise skip the address and continue execution.	
`C!`	n addr →	**Store byte** The 8 bit value *n* is stored in the byte at address *addr*.	
`C@`	addr → n	**Load byte** The byte at address *addr* is placed on the top of the stack.	
`C,`	n →	**Compile byte** The 8 bit value *n* is stored in the next free byte of the heap and the heap pointer is incremented by one.	
`C=`	→	**Align heap** The heap allocation pointer is adjusted to the next four byte boundary. This must be done following a sequence of `C,` operations.	
`CLEAR`	→	**Clear stack** All items on the stack are discarded.	
`COMPARE`	s1 s2 → n	**Compare strings** The two strings whose addresses are given by *s1* and *s2* are compared. If *s1* is less than *s2*, -1 is returned; if *s1* is greater than *s2*, 1 is returned. If *s1* and *s2* are equal, 0 is returned.	`STRING`
`COMPILE` *w*	→	**Compile word** Adds the compile address of the word that follows in line to the definition currently being compiled.	`COMPILERW`
`[COMPILE]` *word*	→	**Compile immediate word** Compiles the address of *word*, even if *word* is marked `IMMEDIATE`.	`COMPILERW`
`CONSTANT` *x*	n →	**Declare constant** Declares a constant named *x*. When *x* is executed, the value *n* will be left on the stack.	
`COS`	f1 → f2	**Cosine** The floating point value on the top of the stack is replaced by its cosine.	`MATH`
`CR`	→	**Carriage return** The standard output stream is advanced to the first character of the next line.	`CONIO`
`CREATE`	→	**Create object** Create an object, given the name which appears next in the input stream, with a default action of pushing the parameter field address of the object when executed. No storage is allocated; normally the parameter field will be allocated and initialised by the defining word code that follows the `CREATE`.	
`DEPTH`	→ n	**Stack depth** Returns the number of items on the stack before `DEPTH` was executed.	

Atlast Primitives: Alphabetical Reference

`DO` limit n →

Definite loop
Executes the loop from the following word to the matching `LOOP` or `+LOOP` until *n* increments past the boundary between *limit*−1 and *limit*. Note that the loop is always executed at least once (see `?DO` for an alternative to this).

`?DO` limit n →

Conditional loop
If *n* equals *limit*, skip immediately to the matching `LOOP` or `+LOOP`. Otherwise, enter the loop, which is thenceforth treated as a normal `DO` loop.

`DOES>` →

Run-time action
Sets the run-time action of a word created by the last `CREATE` to the code that follows. When the word is executed, its body address is pushed on the stack, then the code that follows the `DOES>` will be executed.

`DROP` n →

Discard top of stack
Discards the value at the top of the stack.

`DUP` n → n n

Duplicate
Duplicates the value at the top of the stack.

`?DUP` n → 0 / n n

Conditional duplicate
If top of stack is nonzero, duplicate it. Otherwise leave zero on top of stack.

`ELSE` →

Else
Used in an `IF`—`ELSE`—`THEN` sequence, delimits the code to be executed if the if-condition was false.

`EXECUTE` addr →

Execute word
Executes the word with compile address *addr*.

`EXIT` →

Exit definition
Exit from the current definition immediately. Note that `EXIT` cannot be used within a `DO`—`LOOP`; use `LEAVE` instead.

`EXP` f1 → f2

$\mathbf{f2 = e^{f1}}$ `MATH`
The floating point value on the top of the stack is replaced by its natural antilogarithm.

`F+` f1 f2 → f3

$\mathbf{f3 = f1 + f2}$ `REAL`
The two floating point values on the top of the stack are added and their sum is placed on the top of the stack.

`F-` f1 f2 → f3

$\mathbf{f3 = f1 - f2}$ `REAL`
The floating point value *f2* is subtracted from the floating point value *f1* and the result is placed on the top of the stack.

`F*` f1 f2 → f3

$\mathbf{f3 = f1 \times f2}$ `REAL`
The two floating point values on the top of the stack are multiplied and their product is placed on the top of the stack.

`F/` f1 f2 → f3

$\mathbf{f3 = f1 \div f2}$ `REAL`
The floating point value *f1* is divided by the floating point value *f2* and the quotient is placed on the top of the stack.

`F.` f →

Print floating point `REAL`
The floating point value on the top of the stack is printed.

`F<` f1 f2 → flag

Floating less than `REAL`
The top of stack is set to −1 if *f1* is less than *f2* and 0 otherwise.

`F<=` f1 f2 → flag

Floating less than or equal `REAL`
The top of stack is set to −1 if *f1* is less than or equal to *f2* and 0 otherwise.

`F<>` f1 f2 → flag

Floating not equal `REAL`
The top of stack is set to −1 if *f1* is not equal to *f2* and 0 otherwise.

`F=` f1 f2 → flag

Floating equal `REAL`
The top of stack is set to −1 if *f1* is equal to *f2* and 0 otherwise.

`F>` f1 f2 → flag

Floating greater than `REAL`
The top of stack is set to −1 if *f1* is greater than *f2* and 0 otherwise.

`F>=` f1 f2 → flag

Floating greater than or equal `REAL`
The top of stack is set to −1 if *f1* is greater than or equal to *f2* and 0 otherwise.

`FABS` f1 → f2

f2 = |f1|
Replaces floating point top of stack with its absolute value.

`FCLOSE` file →

Close file `FILEIO`
The specified file is closed.

`FDELETE` s1 → flag

Delete file `FILEIO`
The file named by the string *s1* is deleted. If the file was successfully deleted, −1 is returned. Otherwise, 0 is returned.

`FGETC` file → char

Read next character `FILEIO`
The next byte is read from the specified *file* and placed on the top of the stack. If end of file is encountered, −1 is returned.

`FGETS` file string → flag

Read string `FILEIO`
The next text line (limited to a maximum of 132 characters) is read from *file* and stored into the buffer at *string*. Input lines are recognised in all the end of line conventions accepted by AutoCAD. The end of line delimiter is deleted from the input line and is not stored in the *string*. If end of file is encountered 0 is returned; otherwise −1 is placed on the top of the stack.

`FILE` *f* →

Declare file `FILEIO`
A file descriptor named *f* is declared. This descriptor may subsequently be associated with a file with `FOPEN`.

`FIND` s → word flag

Look up word `DEFFIELDS`

The word with name given by the string *s* is looked up in the dictionary. If a definition if not found, *word* will be left as the address of the string and *flag* will be set to zero. If the word is present in the dictionary, its compilation address is placed on the stack, followed by a *flag* that is 1 if the word is marked for immediate execution and −1 otherwise.

`FIX` f → n
Floating to integer `REAL`
The floating point number on the top of the stack is replaced by the integer obtained by truncating its fractional part.

`(FLIT)` → f
Push floating point literal `REAL`
Pushes the floating point literal that follows in line onto the top of the stack.

`FLOAD` file → stat
Load file `FILEIO`
The source program starting at the current position in *file* is loaded as if its text appeared at the current character position in the input stream. The status resulting from the evaluation is left on the stack, zero if normal, negative in case of error.

`FLOAT` n → f
Integer to floating `REAL`
The integer value on the top of the stack is replaced by the equivalent floating point value.

`FMAX` f1 f2 → f3
Floating point maximum `FLOAT`
The greater of the two floating point values on the top of the stack is placed on the top of the stack.

`FMIN` f1 f2 → f3
Floating point minimum `FLOAT`
The lesser of the two floating point values on the top of the stack is placed on the top of the stack.

`FNEGATE` f1 → f2
f2 = −f1 `FLOAT`
The negative of the floating point value on the top of the stack replaces the floating point value there.

`FOPEN` fname fmodes file → flag
File open `FILEIO`
The previously declared *file* is opened with the specified file name *fname* given by the string address on the stack in the mode given by *fmodes*. The bits in *fmodes* are 1 for read, 2 for write, 4 for binary, and 8 to create a new file. If the file is opened successfully, −1 is returned; otherwise 0 is returned. The Unix standard streams, `STDIN`, `STDOUT`, and `STDERR` are predefined and automatically opened.

`FORGET` *w* →
Forget word
The most recent definition of word *w* is deleted, along with all words declared more recently than the named word.

`FPUTC` char file → stat
Write character `FILEIO`
The character *char* is written to *file*. If the character is written successfully, *char* is returned; otherwise −1 is returned.

`FPUTS` s file → flag
Write string `FILEIO`

The string *s* is written to *file*, followed by the end of line delimiter used on this system. If the line is written successfully, −1 is returned; otherwise 0 is returned.

`FREAD` file len buf → length

Read file `FILEIO`
Len bytes are read into buffer *buf* from *file*. The number of bytes actually read is returned on the top of the stack.

`FSEEK` offset base file →

Set file position `FILEIO`
The current position of *file* is set to *offset*, relative to the specified *base*: if 0, the beginning of the file; if 1, the current file position; if 2, the end of file.

`FSTRFORM` f format str →

Floating point edit `REAL`
Edits a floating point number *f* into string *str*, using the `sprintf` format given by the string *format*.

`FTELL` file → pos

File position `FILEIO`
Returns the current byte position *pos* for file *file*.

`FWRITE` len buf file → length

File write `FILEIO`
Writes *len* bytes from the buffer at address *buf* to *file*. The number of bytes written is returned on the top of the stack.

`HERE` → addr

Heap address
The current heap allocation address is placed on the top of the stack.

`I` → n

Inner loop index
The index of the innermost `DO`—`LOOP` is placed on the stack.

`IF` flag →

Conditional statement
If *flag* is nonzero, the following statements are executed. Otherwise, execution resumes after the matching `ELSE` clause, if any, or after the matching `THEN`.

`IMMEDIATE` →

Mark immediate
The most recently defined word is marked for immediate execution; it will be executed even if entered in compile state.

`J` → n

Outer loop index
The loop index of the next to innermost `DO`—`LOOP` is placed on the stack.

`L>NAME` lfa → nfa

Link to name field `DEFFIELDS`
Given the link field address of a word on the top of the stack, its name pointer field address is returned.

`LEAVE` →

Exit `DO`—`LOOP`
The innermost `DO`—`LOOP` is immediately exited. Execution resumes after the `LOOP` statement marking the end of the loop.

`LINK>` lfa → cfa

Link field to compile address `DEFFIELDS`
Given the link field address of a word on the top of the stack, the compile address of the word is returned.

`>LINK`	cfa → lfa	**Link address** `DEFFIELDS` Given the compile address of a word, return its link field address.
`(LIT)`	→ n	**Push literal** Pushes the integer literal that follows in line onto the top of the stack.
`LITERAL`	n →	**Compile literal** `COMPILERW` Compiles the value on the top of the stack into the current definition. When the definition is executed, that value will be pushed onto the top of the stack.
`LOG`	f1 → f2	$\mathbf{f2} = \ln \mathbf{f1}$ `MATH` The floating point value on the top of the stack is replaced by its natural logarithm.
`LOOP`	→	**Increment loop index** Adds one to the index of the active loop. If the limit is reached, the loop is exited. Otherwise, another iteration is begun.
`+LOOP`	n →	**Add to loop index** Adds *n* to the index of the active loop. If the limit is reached, the loop is exited. Otherwise, another iteration is begun.
`<MARK`	→ addr	**Backward jump mark** `COMPILERW` Saves the current compilation address on the stack.
`>MARK`	→ addr	**Forward mark** `COMPILERW` Compiles a place-holder offset for a forward jump and saves its address for later backpatching on the stack.
`MAX`	n1 n2 → n3	**Maximum** The greater of *n1* and *n2* is left on the top of the stack.
`MEMSTAT`	→	**Print memory status** `MEMSTAT` The current and maximum memory usage so far are printed on standard output. The sizes allocated for the stack, return stack, and heap are edited, as well as the percentage in use.
`MIN`	n1 n2 → n3	**Minimum** The lesser of *n1* and *n2* is left on the top of the stack.
`MOD`	n1 n2 → n3	**Modulus (remainder)** The remainder when *n1* is divided by *n2* is left on the top of the stack.
`/MOD`	n1 n2 → n3 n4	$\mathbf{n3} = \mathbf{n1} \bmod \mathbf{n2}, \mathbf{n4} = \mathbf{n1} \div \mathbf{n2}$ Divides *n1* by *n2* and leaves quotient on top of stack, remainder as next on stack.
`N>LINK`	nfa → lfa	**Name to link field** `DEFFIELDS` Given the name field pointer address of a word on the top of the stack, leaves the link field address of the word on the top of stack.
`>NAME`	cfa → nfa	**Name address** `DEFFIELDS` Given the compile address of a word, return its name pointer field address.

Atlast Primitives: Alphabetical Reference

`NAME>` nfa → cfa

Name field to compile address `DEFFIELDS`
Given the address of the name pointer field of a word on the top of the stack, leaves the compile address of the word on the top of the stack.

`NAME>S!` nfa string →

Get name field `DEFFIELDS`
Stores the name field of the word pointed to by *nfa* into *string*.

`NEGATE` n1 → n2

$n2 = -n1$
Negates the value on the top of the stack.

`(NEST)` →

Invoke word
Pushes the instruction pointer onto the return stack and sets the instruction pointer to the next word in line.

`NOT` n1 → n2

Logical not
Inverts the bits in the value on the top of the stack. This performs logical negation for truth values of −1 (True) and 0 (False).

`OR` n1 n2 → n3

Bitwise OR
Stores the bitwise OR of *n1* and *n2* on the stack.

`OVER` n1 n2 → n1 n2 n1

Duplicate second item
The second item on the stack is copied to the top.

`PICK` $\ldots n_2\ n_1\ n_0$ index → $\ldots n_0\ n_{index}$

Pick item from stack
The *index*th stack item is copied to the top of the stack. The top of stack has *index* 0, the second item *index* 1, and so on.

`POW` f1 f2 → f3

$f3 = f1^{f2}$ `MATH`
The second floating point value on the stack is taken to the power of the top floating point stack value and the result is left on the top of the stack.

`QUIT` →

Quit execution
The return stack is cleared and control is returned to the interpreter. The stack is not disturbed.

`>R` n →

To return stack
Removes the top item from the stack and pushes it onto the return stack.

`R>` → n

From return stack
The top value is removed from the return stack and pushed onto the stack.

`R@` → n

Fetch return stack
The top value on the return stack is pushed onto the stack. The value is not removed from the return stack.

`REPEAT` →

Close `BEGIN`—`WHILE`—`REPEAT` loop
Another iteration of the current `BEGIN`—`WHILE`—`REPEAT` loop having been completed, execution continues after the matching `BEGIN`.

`<RESOLVE` addr →

Backward jump resolve `COMPILERW`
Compiles the address saved by the matching `<MARK`.

`>RESOLVE` addr →

Forward jump resolve `COMPILERW`

Backpatches the address left by the matching `>MARK` to jump to the next word to be compiled.

`ROLL` $\ldots n_2\ n_1\ n_0$ index → $\ldots n_0\ n_{index}$

Rotate *index*th item to top

The stack item selected by *index*, with 0 designating the top of stack, 1 the second item, and so on, is moved to the top of the stack. The intervening stack items are moved down one item.

`ROT` n1 n2 n3 → n2 n3 n1

Rotate 3 items

The third item on the stack is placed on the top of the stack and the second and first items are moved down.

`-ROT` n1 n2 n3 → n3 n1 n2

Reverse rotate

Moves the top of stack to the third item, moving the third and second items up.

`S!` s1 s2 →

Store string `STRING`

The string at address *s1* is copied into the string at *s2*.

`S+` s1 s2 →

String concatenate `STRING`

The string at address *s1* is concatenated to the string at address *s2*.

`S>NAME!` string nfa →

Store name field `DEFFIELDS`

Stores the *string* into the name field of the word given by name pointer field *nfa*.

`SHIFT` n1 n2 → n3

Shift n1 by n2 bits

The value *n1* is logically shifted the number of bits specified by *n2*, left if *n2* is positive and right if *n2* is negative. Zero bits are shifted into vacated bits.

`SIN` f1 → f2

Sine `MATH`

The floating point value on the top of the stack is replaced by its sine.

`SQRT` f1 → f2

$\mathbf{f2} = \sqrt{\mathbf{f1}}$ `MATH`

The floating point value on the top of the stack is replaced by its square root.

`STATE` → addr

System state variable

The address of the system state variable is pushed on the stack. The state is zero if interpreting, nonzero if compiling.

`STRCAT` s1 s2 →

String concatenate `STRING`

The string at address *s1* is concatenated to the string at address *s2*.

`STRCHAR` s1 s2 →

String character search `STRING`

The string at address *s1* is searched for the first occurrence of the first character of string *s2*. If that character appears nowhere in *s1*, 0 is returned. Otherwise, the address of the first occurrence in *s1* is left on the top of the stack.

`STRCMP` s1 s2 → n

String compare `STRING`

The string at address *s1* is compared to the string at address *s2*. If *s1* is less than *s2*, −1 is returned. If *s1* and *s2* are equal, 0 is returned. If *s1* is greater than *s2*, 1 is returned.

`STRCPY` s1 s2 →

Store string `STRING`
The string at address *s1* is copied into the string at *s2*.

`STRFORM` n format str →

Integer edit `STRING`
Edits the number *n* into string *str*, using the `sprintf` format given by the string *format*. Note: the reference to the number in the format must be as a `long` value, for example `"%ld"`.

`STRING` ***x*** size →

Declare string `STRING`
Declares a string named *x* of a maximum of *size*−1 characters.

`STRINT` s1 → s2 n

String to integer `STRING`
Scans an integer from *s1*. The integer scanned is placed on the top of the stack and the address of the character that terminated the scan is stored as the next item on the stack.

`STRLEN` s → n

String length `STRING`
The length of string *s* is placed on the top of the stack.

`(STRLIT)` → s

String literal `STRING`
Pushes the address of the string literal that follows in line onto the stack.

`STRREAL` s1 → s2 f

String to real `STRING`
Scans a floating point number from *s1*. The floating point number scanned is placed on the top of the stack and the address of the character that terminated the scan is stored as the next item on the stack.

`SUBSTR` s1 start length s2 →

Extract substring `STRING`
The substring of string *s1* that begins at character *start*, with the first character numbered 0, extending for *length* characters, with −1 designating all characters to the end of string, is stored into the string *s2*.

`SWAP` n1 n2 → n2 n1

Swap top two items
The top two stack items are interchanged.

`SYSTEM` s → n

Execute system command `SYSTEM`
The operating system command given in the string *s* is passed to the system's command interpreter (shell). The system result status returned after the command completes is left on the top of the stack.

`TAN` f1 → f2

Tangent `MATH`
The floating point value on the top of the stack is replaced by its tangent.

`THEN` →

End if
Used in an `IF`—`ELSE`—`THEN` sequence, marks the end of the conditional statement.

`TRACE` n →

Trace mode `TRACE`
If *n* is nonzero, trace mode is enabled. If *n* is zero, trace mode is turned off.

`TYPE` s →

Print string `CONIO`

The string at address *s* is printed on standard output.

`UNTIL` flag →

End `BEGIN`—`UNTIL` loop

If *flag* is zero, the loop continues execution at the word following the matching `BEGIN`. If *flag* is nonzero, the loop is exited and the word following the `UNTIL` is executed.

`VARIABLE` ***x*** →

Declare variable

A variable named *x* is declared and its value is set to zero. When *x* is executed, its address will be placed on the stack. Four bytes are reserved on the heap for the variable's value.

`WALKBACK` n →

Walkback mode `WALKBACK`

If *n* is nonzero, a walkback trace through active words will be performed whenever an error occurs during execution. If *n* is zero, the walkback is suppressed.

`WHILE` flag →

Decide `BEGIN`—`WHILE`—`REPEAT` loop

If *flag* is nonzero, execution continues after the `WHILE`. If *flag* is zero, the loop is exited and execution resumed after the `REPEAT` that marks the end of the loop.

`WORDS` →

List words defined `CONIO`

Defined words are listed, from the most recently defined to the first defined. If the system supports keystroke trapping, pressing any key will pause the display of defined words; pressing carriage return will abort the listing—any other key resumes it. On other systems, only the 20 most recently defined words are listed.

`WORDSUSED` →

List words used `WORDSUSED`

The words used by this program are listed on standard output. If the system supports keystroke trapping, the listing may be aborted by pressing a key while the output is in progress. The words used report is useful in configuring a custom version of ATLAST that includes just the words needed by the program it executes.

`WORDSUNUSED` →

List words not used `WORDSUSED`

The words not used by this program are listed on standard output. If the system supports keystroke trapping, the listing may be aborted by pressing a key while the output is in progress. The words not used report is useful in configuring a custom version of ATLAST that includes just the words needed by the program it executes.

`XOR` n1 n2 → n3

Bitwise exclusive OR

Stores the bitwise exclusive or of *n1* and *n2* on the stack.

`(XDO)` limit n →

Execute loop

At runtime, enters a loop that will step until *n* increments and becomes equal to *limit*

`(X?DO)` limit n →

Execute conditional loop
At runtime, tests if *n* equals *limit.* If so, skips until the matching `LOOP` or `+LOOP`. Otherwise, enters the loop.

`(XLOOP)` →

Increment loop index
At runtime, adds one to the index of the active loop and exits if equal to the limit. Otherwise returns to the matching `DO` or `?DO`.

`(+XLOOP)` incr →

Add to loop index
At runtime, increments the loop index by the top of stack. If the loop is not done, begins the next iteration.

Atlast Primitives: Alphabetical Summary

+	n1 n2 → n3	n3 = n1 + n2
-	n1 n2 → n3	n3 = n1 − n2
*	n1 n2 → n3	n3 = n1 × n2
/	n1 n2 → n3	n3 = n1 ÷ n2
’ *word*	→ caddr	Obtain compilation address
,	n →	Store in heap
.	n →	Print top of stack
.(*str*	→	Print constant string
.S	→	Print stack
." *str*	→	Print immediate string
: *w*	→	Begin definition
;	→	End definition
<	n1 n2 → flag	Less than
<=	n1 n2 → flag	Less than or equal
<>	n1 n2 → flag	Not equal
=	n1 n2 → flag	Equal
>	n1 n2 → flag	Greater
>=	n1 n2 → flag	Greater than or equal
?	addr →	Print indirect
!	n addr →	Store into address
+!	n addr →	Add indirect
@	addr → n	Load
[	→	Set interpretive state
[’] *word*	→ caddr	Push next word
]	→	End interpretive state
0<	n1 → flag	Less than zero
0<>	n1 → flag	Nonzero
0=	n1 → flag	Equal to zero
0>	n1 → flag	Greater than zero
1+	n1 → n2	Add one
1-	n1 → n2	Subtract one
2+	n1 → n2	Add two
2-	n1 → n2	Subtract two
2*	n1 → n2	Times two
2/	n1 → n2	Divide by two
2!	n1 n2 addr →	Store two words
2@	addr → n1 n2	Load two words
2CONSTANT *x*	n1 n2 →	Double word constant
2DROP	n1 n2 →	Double drop
2DUP	n1 n2 → n1 n2 n1 n2	Duplicate two
2OVER	n1 n2 n3 n4 → n1 n2 n3 n4 n1 n2	Double over
2ROT	n1 n2 n3 n4 n5 n6 → n3 n4 n5 n6 n1 n2	Double rotate
2SWAP	n1 n2 n3 n4 → n3 n4 n1 n2	Double swap
2VARIABLE *x*	→	Double variable
ABORT	→	Abort
ABORT" *str*	→	Abort with message
ABS	n1 → n2	$n2 = \lvert n1 \rvert$
ACOS	f1 → f2	$f2 = \arccos f1$
AGAIN	→	Indefinite loop
ALLOT	n →	Allocate heap
AND	n1 n2 → n3	Bitwise AND
ARRAY *x*	s_1 s_2 … s_n n esize →	Declare array

ATLAST Primitives: Alphabetical Summary

ASIN	f1 → f2	f2 = arcsin f1
ATAN	f1 → f2	f2 = arctan f1
ATAN2	f1 f2 → f3	f3 = arctan f1/f2
BEGIN	→	Begin loop
BODY>	pfa → cfa	Body to word
>BODY	cfa → pfa	Body address
BRANCH	→	Branch
?BRANCH	flag →	Conditional branch
C!	n addr →	Store byte
C@	addr → n	Load byte
C,	n →	Compile byte
C=	→	Align heap
CLEAR	→	Clear stack
COMPARE	s1 s2 → n	Compare strings
COMPILE *w*	→	Compile word
[COMPILE] *word*	→	Compile immediate word
CONSTANT *x*	n →	Declare constant
COS	f1 → f2	Cosine
CR	→	Carriage return
CREATE	→	Create object
DEPTH	→ n	Stack depth
DO	limit n →	Definite loop
?DO	limit n →	Conditional loop
DOES>	→	Run-time action
DROP	n →	Discard top of stack
DUP	n → n n	Duplicate
?DUP	n → 0 / n n	Conditional duplicate
ELSE	→	Else
EXECUTE	addr →	Execute word
EXIT	→	Exit definition
EXP	f1 → f2	f2 = e^{f1}
F+	f1 f2 → f3	f3 = f1 + f2
F-	f1 f2 → f3	f3 = f1 − f2
F*	f1 f2 → f3	f3 = f1 × f2
F/	f1 f2 → f3	f3 = f1 ÷ f2
F.	f →	Print floating point
F<	f1 f2 → flag	Floating less than
F<=	f1 f2 → flag	Floating less than or equal
F<>	f1 f2 → flag	Floating not equal
F=	f1 f2 → flag	Floating equal
F>	f1 f2 → flag	Floating greater than
F>=	f1 f2 → flag	Floating greater than or equal
FABS	f1 → f2	f2 = |f1|
FCLOSE	file →	Close file
FDELETE	s1 → flag	Delete file
FGETC	file → char	Read next character
FGETS	file string → flag	Read string
FILE *f*	→	Declare file
FIND	s → word flag	Look up word
FIX	f → n	Floating to integer
(FLIT)	→ f	Push floating point literal
FLOAD	file → stat	Load file

Atlast Primitives: Alphabetical Summary

FLOAT	n → f	Integer to floating
FMAX	f1 f2 → f3	Floating point maximum
FMIN	f1 f2 → f3	Floating point minimum
FNEGATE	f1 → f2	f2 = −f1
FOPEN	fname fmodes file → flag	File open
FORGET *w*	→	Forget word
FPUTC	char file → stat	Write character
FPUTS	s file → flag	Write string
FREAD	file len buf → length	Read file
FSEEK	offset base file →	Set file position
FSTRFORM	f format str →	Floating point edit
FTELL	file → pos	File position
FWRITE	len buf file → length	File write
HERE	→ addr	Heap address
I	→ n	Inner loop index
IF	flag →	Conditional statement
IMMEDIATE	→	Mark immediate
J	→ n	Outer loop index
L>NAME	lfa → nfa	Link to name field
LEAVE	→	Exit DO—LOOP
LINK>	lfa → cfa	Link field to compile address
>LINK	cfa → lfa	Link address
(LIT)	→ n	Push literal
LITERAL	n →	Compile literal
LOG	f1 → f2	f2 = ln f1
LOOP	→	Increment loop index
+LOOP	n →	Add to loop index
<MARK	→ addr	Backward jump mark
>MARK	→ addr	Forward mark
MAX	n1 n2 → n3	Maximum
MEMSTAT	→	Print memory status
MIN	n1 n2 → n3	Minimum
MOD	n1 n2 → n3	Modulus (remainder)
/MOD	n1 n2 → n3 n4	n3 = n1 mod n2, n4 = n1 ÷ n2
N>LINK	nfa → lfa	Name to link field
>NAME	cfa → nfa	Name address
NAME>	nfa → cfa	Name field to compile address
NAME>S!	nfa string →	Get name field
NEGATE	n1 → n2	n2 = −n1
(NEST)	→	Invoke word
NOT	n1 → n2	Logical not
OR	n1 n2 → n3	Bitwise OR
OVER	n1 n2 → n1 n2 n1	Duplicate second item
PICK	$\ldots n_2\ n_1\ n_0$ index → $\ldots n_0\ n_{index}$	Pick item from stack
POW	f1 f2 → f3	$f3 = f1^{f2}$
QUIT	→	Quit execution
>R	n →	To return stack
R>	→ n	From return stack
R@	→ n	Fetch return stack
REPEAT	→	Close BEGIN—WHILE—REPEAT loop
<RESOLVE	addr →	Backward jump resolve
>RESOLVE	addr →	Forward jump resolve

Atlast Primitives: Alphabetical Summary

`ROLL`	$\ldots n_2\ n_1\ n_0$ index → $\ldots n_0\ n_{index}$	Rotate *index*th item to top
`ROT`	n1 n2 n3 → n2 n3 n1	Rotate 3 items
`-ROT`	n1 n2 n3 → n3 n1 n2	Reverse rotate
`S!`	s1 s2 →	Store string
`S+`	s1 s2 →	String concatenate
`S>NAME!`	string nfa →	Store name field
`SHIFT`	n1 n2 → n3	Shift n1 by n2 bits
`SIN`	f1 → f2	Sine
`SQRT`	f1 → f2	$f2 = \sqrt{f1}$
`STATE`	→ addr	System state variable
`STRCAT`	s1 s2 →	String concatenate
`STRCHAR`	s1 s2 →	String character search
`STRCMP`	s1 s2 → n	String compare
`STRCPY`	s1 s2 →	Store string
`STRFORM`	n format str →	Integer edit
`STRING` *x*	size →	Declare string
`STRINT`	s1 → s2 n	String to integer
`STRLEN`	s → n	String length
`(STRLIT)`	→ s	String literal
`STRREAL`	s1 → s2 f	String to real
`SUBSTR`	s1 start length s2 →	Extract substring
`SWAP`	n1 n2 → n2 n1	Swap top two items
`SYSTEM`	s → n	Execute system command
`TAN`	f1 → f2	Tangent
`THEN`	→	End if
`TRACE`	n →	Trace mode
`TYPE`	s →	Print string
`UNTIL`	flag →	End `BEGIN`—`UNTIL` loop
`VARIABLE` *x*	→	Declare variable
`WALKBACK`	n →	Walkback mode
`WHILE`	flag →	Decide `BEGIN`—`WHILE`—`REPEAT` loop
`WORDS`	→	List words defined
`WORDSUSED`	→	List words used
`WORDSUNUSED`	→	List words not used
`XOR`	n1 n2 → n3	Bitwise exclusive OR
`(XDO)`	limit n →	Execute loop
`(X?DO)`	limit n →	Execute conditional loop
`(XLOOP)`	→	Increment loop index
`(+XLOOP)`	incr →	Add to loop index

Benchmark Program Listings

SQRT.ATL

```
2variable x
2variable y

: csqrt
        2dup 0.0 f< if
           cr ." "SQRT: Negative argument!"
           exit
        then
        2dup 0.0 f<> if
           2dup 2dup x 2!
           1.893872 f* 0.154116 f+
           1.047988 f* 1.0 f+
           f/ y 2!                 \ y=(0.154116+1.893872*x)/(1.0+1.047988*x)

           y 2@                    \ y
           0.0                     \ y c
           begin
                 2swap             \ c y
                 2dup              \ c y y
                 x 2@              \ c y y x
                 2over             \ c y y x y
                 f/                \ c y y x/y
                 f-                \ c y y-x/y
                 -0.5              \ c y (y-x/y) -0.5
                 f*                \ c y (y-x/y)*-0.5
                 2dup              \ cl y c c
                 2rot              \ cl c c y
                 f+                \ cl c c+y
                 2rot              \ c c+y cl
                 2rot              \ c+y cl c
                 2swap             \ c+y c cl
                 2over             \ c+y c cl c
                 f=                \ c+y c =0?
           until
           2drop
        then
;

: cbenchmark 10000 0 do 2.0 csqrt 2drop loop ." "Done\n" ;
: sbenchmark 100000 0 do 2.0 sqrt 2drop loop ." "Done\n" ;

.( "Type \"cbenchmark\" to run the CSQRT benchmark (10000 iterations).\n"
.( "Type \"sbenchmark\" to run the SQRT benchmark (100000 iterations).\n"
```

Benchmark Program Listings

CSQRT.C

```
#include <stdio.h>

double
/*FCN*/asqrt(x)
  double x;
{
    double c, cl, y;
    int n;

    if (c == 0.0)
        return (0.0);

    if (x < 0.0)
        abort();

    y = (0.154116 + 1.893872 * x) / (1.0 + 1.047988 * x);
    c = 0.0;
    n = 20;
    do {
        cl = c;
        c = (y - x / y) * 0.5;
        y -=  c;
    } while (c != cl && --n);
    return y;
}

main()
{
    int i;
    char a[300];

    fputs("Ready to test: ", stdout);
    gets(a);

    for (i = 0; i < 100000; i++)
        asqrt(2.0);
    printf("Done.\n");
}
```

Benchmark Program Listings

SSQRT.C

```
#include <stdio.h>
#include <math.h>

main()
{
    int i;
    char a[300];

    fputs("Ready to test: ", stdout);
    gets(a);

    for (i = 0; i < 100000; i++)
        sqrt(2.0);
    printf("Done.\n");
}
```

Benchmark Program Listings

SQRT.LSP

```
(defun sqr (x / y c cl)
    (if (or (= 'REAL (type x)) (= 'INT (type x)))
        (progn
            (cond ((minusp x) 'Negative-argument)
                  ((zerop x) 0.0)
                  (t (setq y (/ (+ 0.154116 (* x 1.893872))
                                           (+ 1.0 (* x 1.047988))
                                )
                     )
                     (setq c (/ (- y (/ x y)) 2.0))
                     (setq cl 0.0)
                     (while (not (equal c cl))
                        (setq y (- y c))
                        (setq cl c)
                        (setq c (/ (- y (/ x y)) 2.0))
                     )
                     y
                  )
            )
        )
        (progn
            (princ "Invalid argument.")
            (princ)
        )
    )
)

(defun C:csqrt () (repeat 10000 (sqr 2.0)))
(defun C:ssqrt () (repeat 10000 (sqrt 2.0)))
```