NISTIR 4388

AMPLE CORE INTERPRETER: USER'S GUIDE

J. C. Boudreaux

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Center for Manufacturing Engineering
Automated Production Technology Division
Gaithersburg, MD 20899

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
John W. Lyons, Director

NIST

NISTIR 4388

AMPLE CORE INTERPRETER: USER'S GUIDE

J. C. Boudreaux

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Center for Manufacturing Engineering
Automated Production Technology
Gaithersburg, MD 20899

September 1990

Foreword

Origins. Since its inception in 1984, *AMPLE* has been conceived as a programming language environment which permits the construction of control interfaces to industrial manufacturing systems /3/, /5/, /6/. This report describes the *AMPLE Core Interpreter, Version 1.0.*

AMPLE has gone through two major editions. The first was Version 0.1 in which *AMPLE* Core was a dialect of LISP called Franzlisp /5/, /17/. This version was hosted on a Silicon Graphics[1] IRIS workstation under the Unix[2] operating system. This version of *AMPLE*, including such operational modules as the *AMPLE* Workstation Animation Package (AWAP) and the Real-Time Control Interface (ARTCI), which were coded in Fortran, has been discussed in /3/. The primary objective of *AMPLE* Version 0.1 was to provide off-line programming services within the framework of the Automated Manufacturing Research Facility (AMRF). For example, ARTCI allowed the user to select operations for the AMRF Horizontal Workstation from pre-defined menus. After all operations were selected, command data files for all levels of the control hierarchy were created and then transfered to the workstation controller by means of the *AMPLE* Communication (AComm) module.

By mid-1988 the main focus of our effort was shifted to the Quality in Automation (QIA) project. Even though QIA would make use of the off-line programming capabilities of *AMPLE* /16/, two decisions were made which significantly changed the character of the *AMPLE* Core Interpreter (amcore). First, it was decided that the *AMPLE* platform in the QIA environment would be an MS-DOS[3] AT-class personal computer. This decision had two immediate consequences: (i) such modules as the Animation Package would be very difficult or impossible to transport to the new platform, and (ii) familiar Unix multiprocessing commands, which had been used to good effect in Version 0.1, would no longer be available. Second, it was decided that amcore would have to provide hard real-time services. Specifically, amcore had to be able to produce values which were not only correct in an abstract mathematical sense, but which were also timely enough to be of practical use to an external system. The second decision forced us to abandon the view that amcore could be defined as a conservative extension of any existing Lisp dialect. The difficulty is that all Lisp dialects, and all interpretive systems, need on occasion to reclaim memory, a process called *garbage collection*. Depending upon the algorithm chosen for this purpose, garbage collection for MS-DOS AT-class computers can take 500 ms or more, which is of little or no importance to the off-line programmer, but which can have devastating effects in real-time processing. Clearly, to develop an *AMPLE* prototype for the QIA project, we would have build a Lisp interpreter with some exotic capabilities, and to do this we would need access to the source code in which the interpreter itself was implemented.

By late 1988, I was busily reviewing those Lisp interpreters for the AT class MS-DOS machines for which public domain source code was available. The short list of candidate system included the XLISP 1.7 system of David Betz, of which I had already obtained a copy[4]. After

[1]Commercial equipment, instruments, or materials are identified in this report in order to specify adequately certain experimental procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the material or equipment identified is necessarily the best available for the purpose.

[2]Unix is a trademark of Bell Laboratories.

[3]MS-DOS is a registered trademark of Microsoft Corporation.

[4]XLISP 1.7 was implemented in the C programming language. Betz distributed XLISP 1.7 and its source code very widely. The distributed material has been copyrighted by Betz, and all rights reserved. The copyright notice in /2/ also states that Betz grants permission "for unrestricted non-commercial use."

several weeks of discussion with Scott Staley (then of the University of Miami), we both concluded that this system would be an execellent starting point for our work. One limitation which demanded our immediate attention was the need to stay within the MS-DOS 640K upper bound on usable memory. Since a substantial run-time stack would be needed for recursion, and since several modules would normally be loaded into **amcore** at the same time, I thought that the **amcore** executable code would have to be no more 250 Kbytes, and preferrably much smaller. This meant that we would have to be absolutely ruthless in eliminating from XLISP all functions except those which were absolutely essential. We succeeded in doing this by reducing XLISP to those functions which are available in almost all Lisp dialects. In fact, the remaining set of functions was just about the same as that provided in the original dialect of Lisp, called LISP 1.5 /13/. The next phase was to develop those new capabilities which would be needed for the system to be a useful prototype of the *AMPLE* Core Interpreter, Version 1.0. By the Fall of 1988, Staley extended the arithmetic of XLISP 1.7 by adding complex numbers, described in Section 15. By mid-summer of 1989, I added the textport and graphic viewport functions, described in Section 18. The primitive **amcore** functions needed to support real-time processing are being developed and will be introduced in subsequent publications.

Getting Started. **Amcore** Version 1.0 has been implemented in Microsoft C 5.0 and will run on MS-DOS AT-class personal computers. An EGA graphics adaptor is also recommended. The distribution disk contains two executable programs `AMCORE.EXE` and `AMCOREM.EXE`. If your computer has an 80x87 floating point coprocessor, then use `AMCORE.EXE`, otherwise, use `AMCOREM.EXE`. To start the interpreter, key in `amcore` (or `amcorem`) to the MS-DOS prompt. Once you see the **amcore** prompt (>), the command

```
> (load "tutor.lsp")
```

will begin a tutorial session. Follow the navigation rules on the bottom line of the screen. This command will not only give you access to the tutorial material, but will also allow you to access `HELP` files. Please remember that this system is a working prototype and that *it is subject to change and revision without prior notification.*

How to Stop. To exit the Interpreter and to return to the operating system, you may either execute the `exit` command:

```
>(exit)
```

or enter the CTRL-Z key chord.

Preparing Programs. The Core Interpreter may be thought of as the boiler room of the *AMPLE* system: it has been specifically designed to be as compact and efficient as possible. Such programming tools as a syntax-directed editor and a workspace manager are now in final development and will be released as *AMPLE* modules. Until these modules are generally available, the most direct way to program this system is to use a text editor to prepare an ASCII file, that is, a file *without* embedded control characters, and then to `load` the file into the interpreter. Program files are handled in the same way as was the file `TUTOR.LSP` which was itself prepared by using a text editor. The effect of `load`ing a program file is identical to keying the same programs into the interpreter directly. Comments begin with a semicolon character and continue to the end of the line. Since the interpreter treats commented characters as whitespace, they may be placed wherever a blank character would be legal

How to Use This Guide. This document is being released as a companion to the *AMPLE* Core Interpreter. I assume throughout that the reader is simultaneously interacting with the Core Interpreter itself. The primary purpose of the *User's Guide* is to allow the reader to get comfortable with the interpreter and ultimately to develop that confidence which comes from being able to correctly predict the behavior of a complicated system. The first five sections present a general overview of *AMPLE.* The remaining sections introduce the reader to all of the functions which are available in the Core Interpreter. Each of these sections has the same basic plan: a brief explanation of the section's main topic; a list of the section's functions, grouped according to similarity; and then a discussion session with transcripts of actual interactions with the Core Interpreter. Because only a few functions can be discussed in detail, the final section is an Appendix which provides a brief description of all of the functions mentioned in this report, listed in alphabetic order.

More Information. For further information or for a copy of the *AMPLE* Core Interpreter, please write to:

J.C. Boudreaux
Center for Manufacturing Engineering
National Institute of Standards and Technology
Bldg. 233, Room A-107
Gaithersburg MD 20899
tel (301)975-3560
FAX (301)417-0514

This page intentionally blank.

Table of Contents

This page intentionally blank.

1 Introduction

- The *Automated Manufacturing Programming Language Environment* (*AMPLE*) system is a global programming language which was developed with the Center for Manufacturing Engineering of the U.S. National Institute of Standards and Technology (formerly the National Bureau of Standards). *AMPLE* was designed to provide
  - a uniform programming language environment for the construction of control interfaces to industrial manufacturing processes, and
  - an integrated system of software tools for translating product design and process planning specifications into workstation and equipment-level control programs.
- The initial design of *AMPLE* was based upon an analysis of the requirements of *flexible manufacturing systems* (FMS). This category of systems includes manufacturing systems whose resident capital equipment is
  - programmable,
  - rapidly reconfigurible to produce a wide spectrum of industrial products, and
  - designed to require infrequent operator intervention under normal conditions.

This page intentionally blank.

2 Defining FMS Workstations

- FMS workstations are configurations of devices, and devices are distributed networks of controllers which can be commanded to perform specific manufacturing operations. Devices include such equipment as
  - numerically controlled (NC) machines,
  - programmable fixtures and other specialized workholding devices,
  - machine tools and tool-changing devices,
  - robots,
  - automated vehicles, and
  - material-handling devices.
- FMS workstations are multi-language environments. There are many controller-level languages: APT (ANSI X3.37-1977), NC machine code (EIA RS-274-D), recently proposed standard languages for programmable controllers (IEC/SC65A/67), and many vendor-specific languages, especially for robotic control. There are also many language-like formalisms for encoding inspection plans (DMIS), part design and specification data (IGES, PDES, STEP, EDIF), and other data elements needed for automated manufacturing.
- FMS workstations are equipped to monitor their own operating states and have the ability to adapt to changes in their operating environment. The need for self-monitoring, especially in the context of adaptive error-recovery, presupposes that FMS workstations are configured with sensors.

This page intentionally blank.

3 Requirements for a programming language enviromnment

- **Reliability.** Every programming language environment (ple) must be a system which can be used as a reliable assistant. Every ple must be able to carry out elementary operations with little or no operator assitance, and must be able to undertake more complicated operations if given appropriate step-by-step instructions.
- **Portability.** Programs must be portable between similarly configured FMS workstations.
- **Service modalities.** Every reasonable ple must provide both off-line and real-time services.
- **Device abstraction.** To insulate the programmer from device-dependent properties of the controllers, every ple must be based on an abstract or generic model of manufacturing devices and processes. Specific devices should be represented as instances of *abstract typedefs*, which group devices into classes based on their *components*, the *operations* or motions that can be performed, and their operating *states*.
- **Code generation.** Since FMS workstations are multilingual and each controller needs to be addressed in its own native language, every ple must supply a very-high-level *code generation* facility which translates high-level commands into the controller's native language. Code generation has two stages:
  1. build a library of program templates by inserting *parameters* within fragments of control programs,
  2. generate application programs by gluing templates together and then uniformly replacing the embedded parameters with appropriate values.

This page intentionally blank.

4 *AMPLE* Architecture

- *AMPLE* consists of a central kernel, or *core*, around which is a loosely confederated bundle of software *modules*.

- Each *module* is a collection of *symbols* whose meaning is established by a legal *AMPLE* program. This program may reference symbols which are supplied in *AMPLE* core or which have been defined within other modules. The set of modules in terms of which the meaning of a symbol is defined is called the *environment*, or *symbolic frame*, of the symbol.

- Some modules are very general and supply functions which are very commonly used. These modules are called *core support modules.* Other modules are tailored to specific application areas and have a narrower scope.

- This architecture has two consequences: first, *AMPLE* must contain explicit mechanisms for the definition of interfaces between each module and the core, and also between several modules; and second, *AMPLE* must be based on a clear and concise model of computation.

This page intentionally blank.

5 The *AMPLE* Core Interpreter

- *AMPLE* core (amcore) is a Lisp dialect which is a small subset of Common Lisp, adapted to the specific requirements of automated manufacturing. Amcore is a set of functions, whose meaning is defined in terms of an explicit model of computation. This model is approximated by an interactive program, called the amcore interpreter. The Version 1.0 prototype of the amcore interpreter was derived from Betz's XLISP 1.7 /1/, /2/.

- Though several paradigms could have guided the design of amcore, Lisp was chosen for the following reasons:

  - **structural simplicity**, that is, every Lisp expression consist of an atomic value or is the result of a function being applied to its arguments.
  - **extensiblity**, that is, primitive functions introduced in amcore and functions which are constructed have precisely the same standing.

- In addition to primitive functions which constitute a small subset of Common Lisp, amcore includes *textport* functions for interactive text windows and graphic *viewport* functions as well as *object-oriented programming* capabilites inherited from XLISP.

- Because of the real-time requirements, amcore also includes a collection of functions which define a *real-time processor.* These functions are not documented in this report.

This page intentionally blank.

6 The Model of Computation

- The *AMPLE* Core Interpreter is a **read-eval-print** loop.
- The **reader** returns the next *AMPLE* expression from the designated input stream. This expression is assembled character at a time under the control of a data structure called the *readtable*.
- Once an expression has been **read**, it is then **evaluated**:

  1. If the expression is a *list*, then **eval** applies the first member of the expression, which is the *function*, to the remaining members of the expression, which are the *arguments*.
  2. In most cases, functions are applied to the values returned by **evaling** the argument expressions. If the function is a *special form*, then the argument expressions are passed without prior **evaluation**.
  3. If the expression is a non-list, or *atom*, then it **evaluates** to itself.

- The returned value is then **printed** to the designated output stream.

```
A:  eval   apply   quote   backquote   function
```

Discussion

6(A): The normal operation of the reader may be modified by inserting **read** *macros* in the input stream:

`'<expr>`	`is`	`(quote <expr>)`
`` `<expr> ``	`is`	`(backquote <expr>)`
`,<expr>`	`is`	`(comma <expr>)`
`,@<expr>`	`is`	`(comma-at <expr>)`

The quote and backquote functions are used to prevent the evaluation of their arguments, that is, the value of a quoted or backquoted expression is the expression itself. Within the context of a backquoted expression it is possible to invoke eval by using the comma or the comma-at construction.

```
> (quote (1 2 3))
(1 2 3)
> '(1 2 3)                   ;using the QUOTE read macro
(1 2 3)
> (backquote (1 2 3))
(1 2 3)
> `(1 2 3)                   ;using the BACKQUOTE read macro
(1 2 3)
> (setq ace '(56 78))        ;QUOTE here prevents evaluation of (56 78)
(56 78)
> (backquote (1 2 (comma ace)))
(1 2 (56 78))
> `(1 2 ,ace)                ;using the COMMA read macro
(1 2 (56 78))
> (backquote (1 2 (comma-at ace)))
(1 2 56 78)
> `(1 2 ,@ace)               ;using the COMMA-AT read macro
(1 2 56 78)
```

Another family of read macros is signaled by the occurrence of the ***hash mark*** in the input stream. The hash mark signals the reader that special handling will be required to build the expression and that the precise nature of the special handling is specified by the next character.

`#'<expr>`	`is`	`(function <expr>)`
`#(<expr>...)`	`is`	`an array of the expressions`
`#x<hdigits>`	`is`	`a hexadecimal number`
`#o<odigits>`	`is`	`an octal number`
`#C(<real><imag>)`	`is`	`a complex number`
`#\<char>`	`is`	`the ASCII code of <char>`

The macro #' abbreviates the function **function**, which is used in place of quote to apply to functional expressions. The value returned is a *function closure.* An error is returned if the argument expression is not a function.

```
> (function 2)
error: not a function - 2
> (function (+ 1 2))
error: not a function - (+ 1 2)
> (function +)
#<Subr: #4e47af18>   ;address of compiled code for + in hex
> #'+
#<Subr: #4e47af18>
```

The **eval** function which defines the operation of **amcore** may be used in the same manner as all other functions. When presented with a single expression, **eval** returns its value.

```
> (eval 2)
2
> (eval '(+ 2 3))
5
> (eval +)
#<Subr: #4e47af18>
```

The **apply** function is quite similar to **eval**, except that **apply** requires two expressions: an expression which designates a function and then the *list* of arguments to which the function is to be applied.

```
> (apply sin '(1.0))
0.841471
> (apply + '(1 2 3))
6
> (apply eval '((+ 1 2 3)))
6
```

This page intentionally blank.

7 Expressions

- *Everything* is an expression.
- Legal *AMPLE* expressions include:

```
characters   ==   ASCII character set
strings      ==   "  a string  "
symbols      ==   ace   a_long_symbol   **a!!?
numbers      ==   1   23.45   7.89e-07   #C(1.2 3.98)
lists        ==   (1 2 3 4)
arrays       ==   #(0 2 3 89 "this is ok")
streams      ==   files
```

- The following expressions are lists with special significance to the eval function:

```
function expressions
applicative expressions
prog expressions
cond expressions
iteration expressions
object expressions
```

```
A:  type-of   atom   symbolp   numberp   complexp   listp   consp

B:  eq   eql   equal

C:  lambda

D:  length
```

Discussion

7(A): The function **type-of** returns the type to which the value of the argument expression belongs. The remaining functions in this group are type predicates, that is, **atom** returns **T** (true) if applied to an atomic entity and **nil** (false) if applied to a list, and so on.

```
> (type-of "  a string  ")
:STRING
> (type-of 'ace)
:SYMBOL
> (type-of 1)
:FIXNUM
> (type-of 23.45)
:FLONUM
> (type-of 7.89e-07)
:FLONUM
> (type-of #C(1.2 3.98))
:COMPLEX
> (type-of '(1 2 3 4))
:CONS
> (type-of #(0 2 3 89 "this is ok"))
:ARRAY
```

7(B): Each of the functions in this group introduces a different equality predicate. The function **equal** returns **T** if and only if its arguments evaluate to equivalent values. The function **eq** returns **T** if and only if its arguments are identical, that is, if and only if both argument expressions point to one and the same stored entity. The function **eql** returns **T** if and only if its arguments are **eq**, or they are numbers of the same type, characters, or strings that designate the same value.

```
> (equal 4 4)
T
> (eq 4 4)
NIL
> (eql 4 4)
T
> (setq a 4)
4
> (setq b a)
4
> (eq a b)
T
```

7(C): The **lambda** function, when applied to a list of expressions called *formal parameters*, and a sequence of expressions called the **lambda** *body*, returns a *function closure*. When presented with a list whose head is such a function closure, **eval** evaluates all of the remaining members of the tail of the list, binds these values to the formal parameters and then evaluates each expression in the **lambda** body within this new environment. After the body has been completely evaluated, the formal parameters are unbound and removed from the environment. The value of the last evaluated expression in the **lambda** body is returned as the value of the entire expression.

```
> (lambda (x) (* x x))
((LAMBDA (X) (* X X)))
> ((lambda (x) (* x x)) 5)
25
> ((lambda (x) (* x x x)) 5)
125
> ((lambda (x y) (+ x y)) 6 7)
13
```

The formal parameter list may be resolved into three distinct blocks of parameters, any one of which may be empty. The first block consists of the list of all *required* parameters. When a lambda expression is applied to an argument list, there must be at least as many arguments as there are required parameters. The parameters are bound to the arguments in left-to-right order. The second block consists of the list of *optional* parameters, beginning with the keyword &optional. If this block is present, then after all of the required parameters have been processed, the optional parameters are bound to the arguments in left-to-right order. If no arguments remain, then the unprocessed optional parameters are bound to **nil**. The third block begins with the keyword &rest which must be followed by a single parameter. If this block is present, then after all required and optional parameters have been processed, the single *rest* parameter is bound to the list of all remaining arguments. If there are no remaining arguments, the *rest* parameter is bound to **nil**.

```
> ((lambda (x &optional y)
2> (cond (y (+ x y))              ;if y is not nil, return x + y;
3>       (t x)))                  ;otherwise, return x.
1> 1 2)                           ;the argument list
3                                 ;the value returned

> ((lambda (x &optional y)
2> (cond (y (+ x y)
4>     )
3>       (t x)))
1> 1)
1

> ((lambda (x &rest end)
2> (cond (end (cons x end))      ;if end is not nil, return the argument list
3>       (t x)))                 ;otherwise,
1> 1 2 3 4 5 6 7)                ;the argument list
(1 2 3 4 5 6 7)                  ;the value returned

> ((lambda (x &rest end)
2> (cond (end (cons x end))
3>       (t x)))
1> 1)
1
```

7(D): Some amcore entities have **length** and others do not. The entities with **length** are lists, arrays, and strings.

```
> (length '(1 2 3 4))
4
> (length '#(0 2 3 89 "this is ok"))
5
> (length "  a string  ")
12
> (length 5)
error: bad argument type - 5
```

8 Symbols

- Symbols are unempty sequences of characters. Symbol names can consist of any sequence of non-blank printable characters except:

  ```
  (   )   '   `   ,   "   ;
  ```

  Uppercase and lowercase characters are not distinguished within symbol names.

- Every symbol may have a *binding*, a *property list*, or both.
- An amcore *environment* is the list of every available symbol together with its associated binding and/or property list.

```
A:   set   setq  setf

B:   defun   defmacro

C:   getprop   putprop   remprop   assoc

D:   symbol-name   symbol-value   symbol-plist

E:   boundp  hash   gensym   intern   make-symbol
```

Discussion

8(A): Three functions are used to assign values to symbols. The function **set** evaluates its arguments, which means that its first argument, a symbol, must be quoted. The functions **setq** and **setf** are both special forms, which means that their arguments are passed without being evaluated. The first argument of **setq** must be a symbol. The first argument of **setf** may be any legal *place*, that is, not only a symbol but also other more complicated expressions. All three assignment functions obtain the value to be assigned by evaluating the second argument.

```
> (set 'ace 45.67)
45.67
> ace
45.67
> (setq beta '(1 2))
(1 2)
> beta
(1 2)
> (setf (car beta) '(4 5))
(4 5)
> beta
((4 5) 2)
> (setf (cdr beta) '(6 7))
(6 7)
> beta
((4 5) 6 7)
```

8(B): The construction of user-defined functions is accomplished by the special form **defun**. The formal parameter list of **defun** has exactly the same structure as that which has already been described for the **lambda** function. Note that the effect of evaluating **defun** is to bind the function symbol to a function closure.

```
> (defun square (x) (* x x))
SQUARE
> square
((LAMBDA (X) (* X X)))
> (square 4)
16
```

The next session shows one method for interpreting list objects as *sets*. In order to be a valid interpretation of sets, the functions must ignore both the relative order of the elements and the presence of any number of duplicate elements. This example introduces the **recursive** style of programming in which the definition of the function contains an explicit or implicit reference to the function being defined.

```
> (defun memberset (x y)
1> (cond
2>    ((null y) nil)
2>    ((equalset x (car y)) t)
2>    (t (memberset x (cdr y)))))
MEMBERSET
>
> (defun equalset (x y)
1>    (cond
2>     ((and (atom x)(atom y)) (equal x y))
2>     ((and (listp x)(listp y))
3>        (and (subset x y)(subset y x)))
2>     (t nil)))
EQUALSET
>
> (defun subset (x y)
1>   (cond
2>     ((and (null x)(null y)) t)
2>     ((null x) t)
2>     ((null y) nil)
2>     ((memberset (car x) y) (subset (cdr x) y))
2>     (t nil)))
SUBSET
>
> (memberset 1 '(2 3 1 2))
T
> (memberset 5 '(2 3 1 2))
NIL
> (equalset '((1 2)(3 4)) '((4 3 3) (1 2 2)(2 1 1 1)))
T
> (equalset nil nil)
T
```

Functions in amcore are treated in much the same way as other types of entities: they can be stored in data structures, passed as arguments, and returned as results. Functions whose arguments explicitly admit functions as arguments may be called *higher-order functions.*

```
> (defun twice (foo)            ;TWICE is higher order
1> (lambda (x) (foo (foo x)))  ;lambda returns a function closure
1> )
TWICE

> twice
((LAMBDA (X) (FOO (FOO X))))
```

```
> (twice 1+)
((LAMBDA (X) (FOO (FOO X))) ((FOO . #<Subr: #43fcadb0>)))
                    ;the function closure in which FOO is
                    ;bound to the compiled function 1+
> 1+
#<Subr: #43fcadb0>

> ((twice 1+) 4)           ;the application of (twice 1+) to 4
6                          ; 6 = (1+ (1+ 4))
```

Symbols defined by means of the defmacro special form are called *macros*. When eval is given a list whose head is a macro, the interpretation takes place in two stages. Using the defmacro body as a *template*, the macro is expanded to an expression, which is then evaluated to produce the value to be returned. For example, the macro while may be defined in terms of the do function, discussed in Section 12:

```
> (defmacro while (test &rest body)
1> '(do ()
2>      ((not ,test))
2>      ,@body))
WHILE
```

That is, the expression

```
          (while (< i 10) (setq i (1+ i)))
```

would be expanded to

```
         (do ()
             ((not (< i 10))
             (setq i (1+ i))
         )
```

which given some assigned value to i would continue looping so long as i is less than 10. Macros constitute a useful but limited tool. For example, macros cannot occur as the first argument of the apply function, nor can macros be used to express inherently *recursive* algorithms. But the fact that macro expansion occurs before evaluation, and specifically before the argument expressions are evaluated, also suggests that macros can on occasion be used to accomplish results which functions are simply not able to do:

```
> (defmacro nil! (x)
1> '(setq ,x nil))
NIL!
> (nil! a_new_symbol)
NIL
> a_new_symbol
```

```
NIL
> (defun nil! (x)
1> (setq x nil))
NIL!
> (nil! another_new_symbol)
error: unbound variable - ANOTHER_NEW_SYMBOL
```

8(C): The property list, or *plist*, of a symbol is a list which contains zero or more entries. Each entry consists of a symbol, called the *property indicator* or *key*, followed by an expression, called the *property value*.

```
> (putprop 'pencil_001 'mechanical 'type)
MECHANICAL
> (putprop 'pencil_001 'pentel 'manufacturer)
PENTEL
> (putprop 'pencil_001 8.99 'cost)
8.99
> (symbol-plist 'pencil_001)
(COST 8.99 MANUFACTURER PENTEL TYPE MECHANICAL)
> (getprop 'pencil_001 'type)
MECHANICAL
> (getprop 'pencil_001 'manufacturer)
PENTEL
> (getprop 'pencil_001 'cost)
8.99
> (remprop 'pencil_001 'cost)
NIL
> (symbol-plist 'pencil_001)
(MANUFACTURER PENTEL TYPE MECHANICAL)
> (putprop 'pencil_001 18.99 'cost)
18.99
> (getprop 'pencil_001 'cost)
18.99
```

An association list, or *alist*, is a list of pairs. The first member of the pair is called the *key* and the second member is called the *value*.

```
> (setq example '((name john)(age 32)(job programmer)))
((NAME JOHN)(AGE 32)(JOB PROGRAMMER))
> (assoc 'name example)
(NAME JOHN)
> (assoc 'age example)
(AGE 32)
> (assoc 'job example)
(JOB PROGRAMMER)
```

```
> (cadr (assoc 'name example))
JOHN
```

9 Lists

- Lists are entities with *heads* and *tails*. The function cons constructs a list entity whose head is the first argument and whose tail is the second argument.
- The function **car** returns the head of the list, **cdr** returns the tail of the list.
- There is one headless list, called **nil**. Every other list has one and only one head.

```
A:   car   cdr   cons

B:   nth   nthcdr   list    append   reverse   last   null
     not   member   subst   sublis   remove

C:   rplaca   rplacd   nconc   delete
```

Discussion

9(A): The functions **car**, **cdr**, and **cons** are the primitive list-processing functions.

```
> (setq a (cons 1 '(2 3 4)))
(1 2 3 4)
> (car a)
1
> (cdr a)
(2 3 4)
> (setq b (cons '(17) '((-4 (0)))))
((17) (-4 (0)))
> (car b)
(17)
> (cdr b)
((-4 (0)))
```

Compositions of up to three **car** and **cdr** operations are represented by single functions whose names begin with **c**, end with **r**, and in between have a sequence of **a** and **d** letters.

```
> (cadr b)
(-4 (0))
> (caadr b)
-4
> (cdadr b)
((0))
```

When **cons** is applied to two atomic arguments, the returned value is called a *dotted pair*.

```
> (cons 1 2)
(1 . 2)
> (setq dotted-pair (cons 1 2))
(1 . 2)
> (car dotted-pair)
1
> (cdr dotted-pair)
2
```

A list may be defined as a dotted pair whose tail is either **nil** or another list.

```
> (cons 1 nil)
(1)
> '(1 . nil)
(1)
> '(1 . (2 . nil))
(1 2)
> '(1 . (2 . (3 . nil)))
(1 2 3)
> '((1 . nil) . (2 . nil))
((1) 2)
```

9(C): The functions in this group are used to do surgery on list structures. The structure is not copied but is destructively altered.

```
> (setq c a)          ;C points to same value that A points to
(1 2 3 4)
> (rplaca a 101)
(101 2 3 4)
> a                   ;A is altered
(101 2 3 4)
> c                   ;alteration is visible from C
(101 2 3 4)
> (rplacd c '(102 103 104))
(101 102 103 104)
> c
(101 102 103 104)
> a
(101 102 103 104)
> (rplaca (cdr a) -202)
(-202 103 104)
> a
(101 -202 103 104)
> c
(101 -202 103 104)
> (rplaca (cddr a) -203)
(-203 104)
> a
(101 -202 -203 104)
> (rplacd (cdr a) nil)
(-202)
> a
(101 -202)
```

The functions **nconc** and **delete** are similar to **append** and **remove**, except that the former alter their arguments and the latter do not.

```
> (append a '(303 304))
(101 -202 303 304)
> a
(101 -202)
> (nconc a '(303 304))
(101 -202 303 304)
> a
(101 -202 303 304)
> c
(101 -202 303 304)
> (remove 303 a :test equal)
```

```
(101 -202 304)
> a
(101 -202 303 304)
> (delete 303 a :test equal)
(101 -202 304)
> a
(101 -202 304)
> c
(101 -202 304)
```

The functions in this class have long been considered problematic and troublesome. Though these functions should be used carefully, what appears at first glance to be misbehavior is nothing more than a clear message that to understand **amcore** one needs to be alert to the inherent memory structure of lists.

```
> (setq case-one (cons (cons T T)(cons T T)))
((T . T) T . T)
> (setq case-two ((lambda (x) (cons x x))(cons T T)))
((T . T) T . T)
> case-one
((T . T) T . T)
> case-two
((T . T) T . T)
> (rplaca (car case-one) NIL)
(NIL . T)
> case-one
((NIL . T) T . T)
> (rplaca (car case-two) NIL)
(NIL . T)
> case-two
((NIL . T) NIL . T)
```

10 Cond expressions

- The primary branching construct is the **cond** expression, which consists of
- a list of expressions called **cond** clauses
- each of which consists of a test expression, followed by a list of zero or more expressions.
- The evaluator processes each clause in sequence, selecting for further evaluation the first clause whose test expression evaluates to a non-nil value.

```
A:   cond   and   or
```

Discussion

10(A): The special form **cond** is almost always used within function definitions. The function **new-abs** is a simple version of the absolute value function **abs**, which returns the result of multiplying its argument by -1, if the argument is less than 0; otherwise, it returns the original number. Notice that the *otherwise* clause is represented by a clause whose head is the conventional non-**nil** value **t**, and that this clause is the last argument of **cond**.

```
> (defun new-abs (x)
1>  (cond ((< x 0) (* -1 x))
2>        (t x))
1> )
NEW-ABS
> (new-abs 1)
1
> (new-abs -1)
1
> (new-abs 0)
0
```

The following examples show that **and** and **or** are special forms whose behavior during evaluation is derived from **cond**. The form **and** processes its arguments until it encounters a **nil** value, at which point it returns **nil** without any further evaluation; otherwise, it returns the value of its last argument. The form **or** processes its arguments until it encounters a non-**nil** value, at which point it returns that value; otherwise, it returns **nil**.

```
> (and (setq a 100)(setq b 101)(setq c 102))
102
> (and (setq d 34)(setq e nil)(setq f 67))
NIL
> d
34
> e
NIL
> f
error: unbound variable - F
> (or (setq h nil)(setq i 230)(setq j 765))
230
> h
NIL
> i
230
> j
error: unbound variable - J
```

11 Prog expressions

- The **prog** expression is used to support the *procedural*, as opposed to strictly functional, style of programming.
- This expression creates an environment within which local variables can be declared and referenced, and the primitive **goto** control method can be used.

```
A:   prog   prog*   go   return

B:   catch   throw
```

Discussion

11(A): The special form **prog** allows one to program in the procedural style. The **prog** form creates a small environment of limited duration and scope within which three distinct operations may be performed: the declaration and binding of *local variables*, that is, symbols whose binding is limited to the **prog** body; the use of the **return** function, which terminates evaluation of the **prog** body and returns a value; and the use of the **go** function, which causes a transfer of control to a *tag* within the **prog** body.

```
> (defun new-length (L)
1> (prog (sum)                        ;SUM is a local variable
2>    (setq sum 0)                    ;initialize SUM to 0
2> here                               ;the HERE tag
2>    (cond ((atom L)(return sum))    ;if L is atomic, return SUM
3>          (t (setq sum (1+ sum))    ;otherwise, increment SUM
4>             (setq L (cdr L))       ;           update L to (cdr L)
4>             (go here)))            ;           go to HERE tag
2> ))
NEW-LENGTH
> (new-length nil)
0
> (new-length '(1 2 3))
3
> (new-length '((((1)))))
1
```

12 Iteration expressions

- Iteration expressions in **amcore** provide general mechanisms for performing repetitive calculations.
- The **do** special form allows an arbitrary number of loop variables, each of which is declared by specifying an initial value and a step function. When the declared *end condition* is met, the iteration terminates and a specified value is returned.
- The **dolist** and **dotimes** special forms evaluate a body of code once for each value of a single variable. The form **dotimes** iterates over a sequence of integers, and the form **dolist** iterates over the elements of a list.
- Mapping is a type of iteration in which a function is successively applied to pieces of one or more sequences.

```
A:   do   do*   dotimes   dolist

B:   mapcar   mapc   maplist   mapl
```

Discussion

12(A): Though the iteration expressions are technically unnecessary in the sense that their behavior can be easily simulated by prog expressions, explicit iteration makes an important contribution to good programming style:

```
> (defun new-reverse (L)
1> (do
2>  ((x L (cdr x))                ;x starts at L, updates to (cdr x)
3>   (y nil (cons (car x) y)))    ;y starts at nil, updates to (cons (car x) y)
2>  ((null x) y))                 ;WHEN x is nil, return y
1>                       ;loop body is NIL
1> )
NEW-REVERSE
> (new-reverse nil)
NIL
> (new-reverse '(1 2 3))
(3 2 1)
> (new-reverse '(((1))))
(((1)))
```

The next function will loop indefinitely, that is, infinite-loop will iterate until it is terminated by the CTRL-C key chord.

```
> (defun infinite-loop ()
1> (do ()                        ;there are no loop variables
2>     ()                        ;there is no loop termination test
2>                           ;the loop body is NIL
2> )
1> )
INFINITE-LOOP
```

A more useful variant of this function loops until a key is pressed.

```
> (defun loop-until-key-pressed ()
1> (do ()                        ;there are no loop variables
2>     ((kbhit))                 ;when KBHIT, terminate loop and return NIL
2>                          ;the loop body is NIL
2> )
1> )
LOOP-UNTIL-KEY-PRESSED
```

12(B): The use of explicit iteration expressions have tended to make the mapping functions less useful than they once were. It should be noted that these functions can still be helpful by suggesting places at which amcore code can be parallelized.

```
> (mapcar '1+ '(1 2 3 4 5))
(2 3 4 5 6)
> (mapcar 'sin '(.10 .20 .30 .40 .50 .60 .70))
```

```
(0.0998334 0.198669 0.29552 0.389418 0.479426 0.564642 0.644218)
> (mapcar '+ '(1 2 3 4 5) '(10 20 30 40 50))
(11 22 33 44 55)
> (mapcar '(lambda (x)(* x x)) '(2 4 6 8))
(4 16 36 64)
> (maplist 'cons '(a b) '(x y))
(((A B) X Y) ((B) Y))
```

This page intentionally blank.

13 Arrays

- An *array* is a compound data structure whose component values are accessed by an *index* mechanism.
- The time it takes to access an **array** component must not be dependent upon the size of its index.
- The components of **amcore** arrays may belong to different types.

```
A:   make-array    aref
```

Discussion

13(A): Note that amcore only allows one-dimensional arrays. But given the covert implementation of arrays as arrays of pointers, this restriction can be liberalized in an entirely straightforward manner.

```
> (defun make-table (m n)
1>   (do ((tmp (make-array m))
3>        (index 0 (1+ index)))
2>       ((= index m) tmp)
2>       (setf (aref tmp index) (make-array n))
2> ))
MAKE-TABLE
> (setq tab (make-table 2 2))
#(#(NIL NIL) #(NIL NIL))
>
> (defun tabref (tab m n)
1>   (aref (aref tab m) n))
TABREF
>
> (tabref tab 1 1)
NIL
> (defun tabupdate (tab m n val)
1>  (setf (aref (aref tab m) n) val)
1> )
TABUPDATE
>
> (tabupdate tab 1 1 -56)
-56
> (tabupdate tab 0 1 "a test")
"a test"
> tab
#(#(NIL "a test") #(NIL -56))
> (tabref tab 1 1)
-56
> (tabref tab 0 1)
"a test"
```

14 Strings

- The concatenation of any finite number of characters is a *string.*
- String literals are sequences of characters surrounded by double quotes. Non-printable characters may be included by using the following codes:

`\\`	is	the escape character '\'
`\n`	is	newline
`\t`	is	tab
`\r`	is	carriage return
`\f`	is	form feed

```
A:   char   string   strcat   substr
```

Discussion

14(A): There are very few string functions in **amcore** which is in keeping with the overall requirement that the Core Interpreter be kept as small as possible.

```
> (char "abcde" 0)
97
> (string 97)
"a"
> (defun strchar (str index)
1> (string (char str index)))
STRCHAR
> (strchar "abcde" 0)
"a"
> (strchar "abcde" 1)
"b"
> (strchar "abcde" 4)
"e"
> (strcat " first" "  second")
" first  second"
> (strcat " first" " second" " third")
" first second third"
> (substr "abcde" 2 4)
"bcde"
> (substr "abcde" 3 7)
"cde"
```

15 Numbers

- **amcore** supports *integers* (fixnums) and *floating-point numbers* (flonums), which are scalar numeric types.

```
12   -456   34567   #o677    #xfde56

1.456    -8.9083456    -1.89098E-2
```

- **amcore** also supports *complex numbers*, which are pairs of integers or floats.

```
#C(34 -67)    #C(1.23 -4.56)  #C(-123 4.567e11)
```

```
A:   fix   float   complex   realpart   imagpart

B:   +  -  *  /  1+  1-

C:   min   max   abs   minusp   zerop   plusp
     <   <=   =   /=   >=   >

D:   rem   random   logand   logior   logxor   lognot

E:   sin   cos   tan   asin   acos   atan   sinh   cosh   tanh

F:   expt   exp   ln   log   sqrt

G:   cis   phase
```

Discussion

15(B): These primitive arithmetic operations may be applied to one or more numeric arguments. If all of the arguments are integers, then the value returned is an integer. If any of the arguments is a floating point number or a complex number, then the result is coerced to that type, just as if the appropriate type coercion function had been called. The default format for printing floating point numbers is the g format, which uses the shorter of the f or e floating-point format, and which suppresses the printing of nonsignificant zeros.

```
> (+ 1)
1
> (+ 1 2)
3
> (+ 1 2 3)
6
> (- 1)
-1
> (setq a 9)
9
> (type-of a)
:FIXNUM
> (setq b 10.0)
10
> (type-of b)
:FLONUM
> (setq c (* a b))
90
> (type-of c)
:FLONUM
> (setf d #C(1.0 0.0))
#C(1 0)
> (+ a b d)
#C(20 0)
> (+ #c(4.0 5.0) #c(7.0 8.0)
1> )
#C(11 13)
> (* #c(4.0 5.0) #c(7.0 8.0))
#C(-12 67)
> (- #c(4.0 5.0) #c(7.0 8.0))
#C(-3 -3)
> (* 2.0 #c(4.0 5.0))
#C(8 10)
> (/ #c(4.0 5.0) #c(7.0 8.0))
#C(0.60177 0.0265487)
> (defun conj (cmplx)
1> (complex (realpart cmplx) (- (imagpart cmplx)))
1> )
CONJ
```

```
> (conj #c(4.0 5.0))
#C(4 -5)
> (conj #c(4.0 -5.0))
#C(4 5)
> (defun norm (cmplx)
1> (sqrt (realpart (* cmplx (conj cmplx))))
1> )
NORM
> (norm #c(1.0 0.0))
1
> (norm #c(4.0 5.0))
6.40312
> (norm #c(7.0 8.0))
10.6301
```

This page intentionally blank.

16 Objects

- An object consists of a data structure containing a pointer to the *class* and an array containing the values of the *instance* variables.
- The only way to communicate with an object is by sending it a *message*. When **eval** is presented a list whose first element is an object, it interprets the value of the second element, which must be a symbol, as the *message selector*,
- then **eval** gets the class of the receiving object and attempts to find a method corresponding to the message selector in the set of messages defined for that class.
- If the message is not found in the object's class and the class has a superclass, the search continues by looking at the messages defined for the superclass.

```
A:   object   class
```

Discussion

16(A): The following example, taken from Betz's tutorial on Xlisp /1/, defines a class of objects that represent simple dictionaries. Each dictionary instance will be an association list whose first member is a symbol and whose second member is a pair. In this example, the class dictionary supports two methods **add** and **find**.

```
> (setq dictionary (class :new '(entries)))
#<Object: #391b7072>

> (dictionary :show)
Object is #<Object: #391b7072>, Class is #<Object: #391bd900>
  MESSAGES = NIL
  IVARS = (ENTRIES)
  CVARS = NIL
  CVALS = #()
  SUPERCLASS = #<Object: #391bd882>
  IVARCNT = 1
  IVARTOTAL = 1
#<Object: #391b7072>
> (dictionary :answer 'add '(name value)
1>   '((setq entries
3>       (cons (cons name value) entries))
2>        value))
#<Object: #391b7072>


> (dictionary :answer 'find '(name &aux entry)
1>   '((cond ((setq entry (assoc name entries))
4>            (cdr entry))
3>           (t
4>            nil))))
#<Object: #391b7072>


> (dictionary :show)
Object is #<Object: #391b7072>, Class is #<Object: #391bd900>
  MESSAGES = ((FIND (NAME &AUX ENTRY)
                (COND ((SETQ ENTRY (ASSOC NAME ENTRIES))
                        (CDR ENTRY)) (T NIL)))
              (ADD (NAME VALUE)
                (SETQ ENTRIES
                  (CONS (CONS NAME VALUE) ENTRIES))
                        VALUE)
             )
  IVARS = (ENTRIES)
  CVARS = NIL
  CVALS = #()
  SUPERCLASS = #<Object: #391bd882>
  IVARCNT = 1
  IVARTOTAL = 1
#<Object: #391b7072>
```

Dictionary instances may now be created and some entries added:

```
> (setq d (dictionary :new))
#<Object: #391b549a>
> (d 'add 'mozart 'composer)
COMPOSER
> (d 'add 'winston 'computer-scientist)
COMPUTER-SCIENTIST
> (d 'find 'mozart)
COMPOSER
> (d 'find 'winston)
COMPUTER-SCIENTIST
> (d :show)
Object is #<Object: #391b549a>, Class is #<Object: #391b7072>
  ENTRIES = ((WINSTON . COMPUTER-SCIENTIST) (MOZART . COMPOSER))
#<Object: #391b549a>
```

```
> (setq e (dictionary :new))
#<Object: #481c794a>
> (e 'add 'one 1)
1
> (e 'add 'two 2.0)
2
> (e 'add 'three 3.0)
3
> (e 'find 'one)
1
> (e 'find 'two)
2
> (e 'find 'three)
3
> (e 'find 'winston)
NIL
> (e :show)
Object is #<Object: #481c794a>, Class is #<Object: #391b7072>
  ENTRIES = ((THREE . 3) (TWO . 2) (ONE . 1))
#<Object: #481c794a>
```

This page intentionally blank.

17 Streams

- *Streams* are expressions that serve as sources or sinks of expressions.
- Streams are usually connected to *files* or to external peripheral devices.
- Streams may be input-only, output-only, or input-output. The streams *standard-input* and *standard-output* have been predefined in the amcore interpreter.

```
A:   read   print   princ   read-char   peek-char   write-char
     read-line   cfprintf1   terpri

B:   flatsize   flatc

C:   openi   openo   openio   opena   close

D:   fseek   ftell

E:   kbhit   getkey
```

Discussion

17(A): All input and output operations are performed by the functions of this group. If the file pointer argument is omitted, then the default stream is understood to be either standard input or standard output.

```
> (setf test "this is a line")
"this is a line"
> (princ test)
this is a line"this is a line"  ;TEST is PRINCed then value is returned
> (print test)
"this is a line"                ;TEST is PRINTed, including newline...
"this is a line"                ;... then value returned

> (setq sample "\tone\n\t\ttwo\n\t\t\tthree") ;more complicated case
"\tone\n\t\ttwo\n\t\t\tthree"
> (princ sample)
         one
               two
                      three"\tone\n\t\ttwo\n\t\t\tthree"
> (print sample)
"\tone\n\t\ttwo\n\t\t\tthree"
"\tone\n\t\ttwo\n\t\t\tthree"


> (defun little-reader ()
1> (do ()
2>     ()
2>     (princ "\t::")
2>     (read)))
LITTLE-READER
> (little-reader)
        ::> 12
        ::> 34
        ::> 56
        ::> this
        ::>                   ;Ctrl-C entered here
[ back to the top level ]
> (defun little-reader-writer ()
1> (do ()
2>     ()
2>     (princ "\t::")
2>     (princ (read))
2>     (terpri)))                ;TERPRI causes newline
LITTLE-READER-WRITER
> (little-reader-writer)
        ::> 12
12
```

```
        ::> 34
34
        ::> this
THIS
        ::> (a list (is here))
(A LIST (IS HERE))
        ::>                ;Ctrl-C entered here
[ back to the top level ]
```

17(C): In order to perform external IO operations, streams must be associated with *files*. This association is established by applying an **open** function to a legal path and filename. If no path is given, then the pathname of the default directory is assumed. All **open** functions return a file pointer which provides access to the external file. The **open** functions differ in *direction*: **openi** is an *input* stream, **openo** is an **output** stream, **openio** is an *in-out* stream, and **opena** is a stream opened in *append* mode. The **close** function breaks the association of a stream with an external file.

```
> (setf fp (openo "test.dat"))     ;TEST.DAT opened for output
#<File: #481c2922>
> (print "this is line 1" fp)
"this is line 1"
> (print "this is line 2" fp)
"this is line 2"
> (print "this is line 3" fp)
"this is line 3"
> (close fp)                      ;TEST.DAT closed
NIL
> (setq gp (openi "test.dat"))    ;TEST.DAT opened for input
#<File: #481c2436>
> (defun dump (fpnt)
1>   (do ((tmp (read-line fpnt)(read-line fpnt))
3>        )
2>       ((null tmp))
2>       (princ tmp)
2>       (terpri)
2> ))
DUMP
> (dump gp)
"this is line 1"
"this is line 2"
"this is line 3"
NIL
> (close gp)
NIL
```

17(D): The **ftell** function returns the position in the stream from which the next IO operation will be performed. The **fseek** function changes the current position.

```
> (setf fp (openi "test.dat"))
#<File: #391b6ec2>
> (ftell fp)             ;get current position
0
> (read fp)
"this is line 1"
> (ftell fp)            ;get position after first READ
16
> (read fp)
"this is line 2"
> (ftell fp)            ;get position after second READ
34
> (read fp)
"this is line 3"
> (ftell fp)            ;get position after third READ
52
> (fseek fp 16 0)       ;set position to 16'th byte
16
> (read fp)             ;read file again ...
"this is line 2"
> (fseek fp 16 0)       ;... and repeat
16
> (read fp)
"this is line 2"
```

18 Textport and Viewport Functions

- A *textport* is any region of a display space which is a two-dimensional array of *character positions*, indexed by **row** in the vertical axis and by **column** in the horizontal.
  - Every character position is itself a two-dimensional array of pixels: those that make up the character are called *foreground* pixels, the rest are called *background* pixels.
  - The appearance of each character position is determined by an *attribute* byte, which determines the color and intensity of the foreground pixels, the color of the background pixels, and whether the foreground blinks or not. The value of the attribute byte is set by **settextcolor**.
  - The current position is indicated by the *cursor*.
- A *viewport* is any region of display space which is a two-dimensional array of *pixels*, indexed by **row** and **column**.
  - Each pixel is a discrete dot of color.
  - The function **setbkcolor** sets the background color for all of the viewport's pixels.
  - The function **setcolor** sets the color to be used by **setpixel** and the other drawing functions.

```
A:    make-window   box   get-margins   savescr    restorescr
      settextcolor  gettextcolor

B:    clear   clearregion   cleareol   cleareos
      scrollup   scrolldown

C:    setcursor   getcurrow   getcurcol   curshift
      rowhome   colhome

D:    setvideomode   setpixel   getpixel
      setcolor   getcolor   setbkcolor   getbkcolor

E:    lineto   moveto   arc   pie   ellipse   rectangle

F:    setcliprgn   setviewport   floodfill
```

Discussion

18(A): The argument of settextcolor must be an integer in the range from 0 to 255. There are eight basic colors:

black	is	0	red	is	4
blue	is	1	magenta	is	5
green	is	2	brown	is	6
cyan	is	3	white	is	7

To determine the attribute value, it is first necessary to assign values to two bit-valued variables: blink (0 is no-blink, 1 is blink), and intensity (0 is low, 1 is high). The value of the attribute byte may be calculated as follows:

```
(128 * blink) + (16 * bkgrnd) + (8 * intensity) + fgrnd
```

where bkgrnd and fgrnd are any basic color values. The next session creates a function which shows all of the possible values of settextcolor and what they actually look like on the screen.

```
> (defun show-sample-text-colors (ch)
1> (clear)
1> (color-block 0 ch)
1> (color-block 64 ch)
1> (color-block 128 ch)
1> (color-block 192 ch)
1> (settextcolor 7)
1> )
SHOW-SAMPLE-TEXT-COLORS
>
> (defun color-block (n ch)
1> (do ((i 0 (1+ i)))
2>     ((= i 64)(terpri))
2>     (settextcolor (+ i n))
2>     (princ ch)
2> ))
COLOR-BLOCK
```

The function basic-screen defines a textport whose upper-left position is in row 1 column 1, and whose lower-right position is in row 25 column 80. The attribute byte is set to 7, which means that the foreground color is low-intensity white, the background color is black, and there will be no blinking. When this function is executed, the window will be cleared and the cursor will be placed in the home position row 1 column 1.

```
> (defun basic-screen ()
1> (settextcolor 7)
1> (make-window 1 1 25 80)
1> (clear)
1> )
BASIC-SCREEN
```

The next function creates a smaller window in the lower part of the screen, beginning in row 8 column 1. The attribute byte indicates that the foreground color is low-intensity `blue`, the background is `white`, and there will be no blinking. Using special printing characters whose ASCII code is greater than 127, the function `box` prints a double-lined box around the margins of the defined window. Having drawn the box, a new window is created inside of it and that new window is `cleared`.

```
> (defun lower-window ()
1> (settextcolor 113)
1> (make-window 8 1 25 80)
1> (clear)
1> (box 8 1 25 80 2)
1> (make-window 9 2 24 79)
1> (clear)
1> )
LOWER-WINDOW
```

18(C): The next function is more complicated than those we have been considering so far in this write up. After storing the current status information concerning the current textport, such as its dimensions, text color, and the position of the cursor, a 10x10 window is created and then the character string passed as an argument is **princ**ed in the new window. Before each invocation of **princ**, **setcursor** is used to set the cursor at a randomly selected row and column. The color of the **princ**ed character is also randomly selected. The inner **do** loop may be terminated by pressing any key, at which time the previous textport is restored.

```
> (defun random-princ (ch)
1> (prog (i j
3>        (*margins (get-margins))     ;store description of current
3>        (*row (getcurrow))           ;textport in local variables
3>        (*col (getcurcol))
3>        (*tcolor (gettextcolor)))
2>    (make-window 10 50 19 59)        ;make new window and ...
2>    (settextcolor 113)
2>    (clear)                          ;CLEAR it
2>    (do ()
3>        ((kbhit)                             ;when KBHIT,
4>         (apply make-window *margins)        ;terminate loop and
4>         (apply setcursor (list *row *col)) ;restore previous texport
4>         (settextcolor *tcolor))
3>       (setq i (rem (random 255) 8))         ;get random row
3>       (setq j (rem (random 255) 8))         ;get random column
3>       (setq c (rem (random 255) 127))       ;get random color
3>       (setcursor (+ 11 i)(+ 51 j))          ;move cursor
3>       (settextcolor c)                      ;change color
3>       (princ ch)                            ;princ
3>    )
2> ))
RANDOM-PRINC
```

18(D): At any given time, a screen may be in one and only one video mode. The current mode may be changed by invoking setvideomode with a proper integer value. Which values are proper depend upon the graphics adaptor and upon the specific functional capabilities of the display monitor. If an improper value is used, then the execution of setvideomode has no effect. The following video modes are commonly used:

mode	type	color	dimensions	adaptor
0	text	b/w	40x25	CGA
1	text	16	40x25	CGA
2	text	b/w	80x25	CGA
3	text	16	80x25	CGA
4	graphics	4	320x200	CGA
5	graphics	4 grey	320x200	CGA
6	graphics	b/w	640x200	CGA
13	graphics	16	320x200	EGA
14	graphics	16	640x200	EGA
15	graphics	4	640x350	EGA

Every EGA adaptor supports all of the video modes defined for CGA, but not conversely.

19 System Functions

- These functions make operating system services available within amcore.

A: dos date tstamp load transcript

B: gc address-of expand alloc mem

C: peek poke

D: exit

Discussion

19(A): The argument of dos is a string which is a legal DOS command string.

```
> (dos "dir /p")
```

Following the normal conventions for strings, occurrences of the backslash character in command strings is represented by a double backslash with no intervening whitespace.

```
> (dos "cd c:\\msc\\lib")
```

When the screen has been patiently put together using the textport functions of the preceeding section, sometimes dos can do terrible damage, seeming almost to tear the screen apart. The following function, using savescr and restorescr, solve the problem rather elegantly:

```
> (defun neat-dos (dos-cmd-str)
1> (savescr)
1> (dos dos-cmd-str)
1> (getkey)
1> (restorescr)
1> )
NEAT-DOS
```

The function **transcript**, when applied to a legal DOS filename, causes the file to be opened in output-only mode and then writes to it every character from ***standard-input*** and ***standard-output***. This file is a complete record of an **amcore** session. If the function **transcript** is called with no argument, then the recording process is terminated and the file closed.

```
> (transcript "sample.trs")
T
> (* 4 5)
20
> (transcript)
NIL
```

20 Debugging Support Functions

- The functions in this group are used to control the behavior of the evaluator when it encounters an error. This behavior is determined by the values assigned to several global variables:

```
*breakenable* controls entrance to break loop on errors
*tracenable*  enable baktrace on errors
*tracelimit*  number of levels of baktrace
```

- If ***breakenable*** is **T**, the error message is printed. If the error is correctable, the correction message is printed.

- If ***tracenable*** is **T**, then the evaluation stack is baktraced. The number of entires printed depends on the value assigned to ***tracelimit***. If it has been assigned as non-numeric value, then the stack is printed.

- At this point, a special **read-eval-print** loop, called the *break loop*, is entered. Unlike the normal top-level loop, break loops allow programmers to invoke the `continue` function and then continue processing after a correctable error. While in a break loop, a break level number is prefixed to the **amcore** prompt. If another error causes a new break loop to be activated, the break level is incremented.

- If ***breakenale*** is **nil**, then the processor looks for an `errset` function. If the flag of `errset` is **T**, then the error message is printed. If there is no surrounding **errset**, then the error message is printed and the system returns to the top-level

```
A:   error   cerror   errset

B:   break   continue   clean-up   top-level   baktrace

C:   evalhook
```

Discussion

20(A): The error function is used to cause a non-recoverable break in the processing of a program.

```
> (defun integer-only (n)
1> (cond ((equal (type-of n) :FIXNUM) n)
2>       (t (error "Not an integer")))
1> )
INTEGER-ONLY
> (integer-only 6)
6
> (integer-only 6.0)
error: Not an integer
> (integer-only nil)
error: Not an integer
```

The function cerror causes an error to be signaled from which it is possible to continue. This function requires two strings: the first is the *continue* message which may be used to suggest corrective actions, and the second is the *error* message. This operational mode requires that the global variable ***breakenable*** be set to **T**.

```
> (setq *breakenable* T)
T
> (defun continuable-integer-only (n)
1> (cond ((equal (type-of n) :FIXNUM) n)
2>       (t (cerror "SETQ n TO INTEGER" "Not an integer")
3>          (* n n)))
1> )
CONTINUABLE-INTEGER-ONLY
> (continuable-integer-only 10)
10
> (continuable-integer-only 5.0)
error: Not an integer
if continued: SETQ n TO INTEGER
1:> (setq n 5)
5
1:> (continue)
[ continue from break loop ]
25
> (setq this-example (continuable-integer-only 9.0))
error: Not an integer
if continued: SETQ n TO INTEGER
1:> (setq n 9)
9
1:> (continue)
[ continue from break loop ]
81
> this-example
81
```

20(B): The break and continue functions are used together to create an interpretive environment which is especially useful when debugging programs.

```
> (defun using-break ()
1> (prog (x)
2>   (setq x 10)
2>   (break)          ;break A
2>   (prog (x)
3>    (setq x -20)
3>    (break)         ;break B
3>   )
2>   (break)          ;break C
2> ))
USING-BREAK

> (using-break)
break: **BREAK**          ;break A processed (break level 1)
1:> x                     ;get current value of x
10
1:> (setq x 78.9)         ;change value of x
78.9
1:> (continue)            ;continue from break A (break level 0)
[ continue from break loop ]
break: **BREAK**          ;break B processed (break level 1)
1:> x                     ;get current value of x
-20
1:> (continue)            ;continue from break B (break level 0)
[ continue from break loop ]
break: **BREAK**          ;break C processed (break level 1)
1:> x                     ;check value of x
78.9
1:> (continue)            ;continue from break C (break level 0)
[ continue from break loop ]
NIL                       ;value returned by USING-BREAK
```

The backtrace function shows the current state of the evaluation stack. That is, it returns a printed list of all of the expressions which are currently being evaluated, beginning with the most recent and working backwards:

```
> (continuable-integer-only 9.5)
error: Not an integer
if continued: SETQ n TO INTEGER
1:> (baktrace)
(BAKTRACE)
(CERROR "SETQ n TO INTEGER" "Not an integer")
(COND ((EQUAL (TYPE-OF N) :FIXNUM) N)
      (T (CERROR "SETQ n TO INTEGER" "Not an integer")
         (* N N)))
(CONTINUABLE-INTEGER-ONLY 9.5)
```

```
NIL
1:> (setq n 9)
9
1:> (continue)
[ continue from break loop ]
81
```

References

1. Betz, D. "An Xlisp Tutorial," *BYTE*, March 1985, 221 *et sq.*

2. Betz, D.M. "XLISP: An Object-oriented Lisp, Version 1.7," unpublished manuscript, June 2, 1986.

3. Bandy, H.T., Carew, V.E. Jr. and Boudreaux, J.C. "An *AMPLE* Version 0.1 Prototype: The HWS Implementation," *National Bureau of Standards, NBSIR 88-3770.*

4. Boudreaux, J.C. "Problem Solving and the Evolution of Programming Languages," *The Role of Language in Porblem Solving - 1*; edited by R. Jernigan, B. Hamil, and D. Weintraub, North-Holland, Amsterdam, 1985; 103-126.

5. Boudreaux, J.C. "The AMPLE Project," *National Bureau of Standards, NBSIR 86-3496.*

6. Boudreaux, J.C. "AMPLE: A Programming Language Environment for Automated Manufacturing," *The Role of Language in Problem Solving - 2*; edited by J.C. Boudreaux, B. Hamill, and R. Jernigan, North Holland, Amsterdam, 1987; 359-376.

7. Boudreaux, J.C. "Requirements for Global Programming Languages," *Proceedings of the Symposium on Manufacturing Application Languages, MAPL 88*, 107-114.

8. Boudreaux, J.C. and Staley, S.M. "Representing and Querying Objects in *AMPLE*," *Engineering Database Management: Leadership Key for the 90's,* The American Society of Mechanical Engineers, 1989; 73-79.

9. Graham, P. "Common Lisp Macros," *AI Expert,* March 1988, 42 *et sq.*

10. Hudak, P. "Conception, Evolution, and Application of Functional Programming Languages," *ACM Computing Surveys,* vol 21(1989), 359-411.

11. Kaisler, S.H. *INTERLISP: The Language and its Usage,* Wiley-Interscience, 1986.

12. Mason, I.A. *The Semantics of Destructive Lisp,* CSLI Stanford, 1986.

13. McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., and Levin, M.I. *LISP 1.5 Programmer's Manual*, Second Edition, MIT Press, 1965.

14. Staley, S.M. and Boudreaux, J.C. "Programming Language Environments for CIM," *Proceedings of PROCIM '88*, 70-72.

15. Steele, G.L. *Common Lisp: The Language,* Digital Press, 1984.

16. Vorberger, T.V. and Boudreaux, J.C. "An Architecture for Quality Control in the QIA Project," in C. Denver Lovett (ed.), *Progress Report of the Quality in Automation Project for FY88, NISTIR 89-4045;* 5-47.

17. Wilensky, R. *LISPcraft,* Norton, 1984.

This page intentionally blank.

Alphabetic List of *amcore* Functions

```
(* {<number>})
  <number>    the numbers
  returns     the result of the multiplication


(+ {<number>})
  <number>    the numbers
  returns     the result of the addition


(- {<number>})
  <number>    the numbers
  returns     the result of the subtraction


(/ {<number>})
  <number>    the numbers
  returns     the result of the division


(/= <scalar_1> <scalar_2>)
  <scalar_1>  the left operand of the comparison
  <scalar_2>  the right operand of the comparison
  returns     the result of comparing <scalar_1> with <scalar_2>


(1+ <number>)
  <number>    the number
  returns     the number plus one


(1- <number>)
  <number>    the number
  returns     the number minus one


(< <scalar_1> <scalar_2>)
  <scalar_1>  the left operand of the comparison
  <scalar_2>  the right operand of the comparison
  returns     the result of comparing <scalar_1> with <scalar_2>


(<= <scalar_1> <scalar_2>)
  <scalar_1>  the left operand of the comparison
  <scalar_2>  the right operand of the comparison
  returns     the result of comparing <scalar_1> with <scalar_2>


(= <scalar_1> <scalar_2>)
  <scalar_1>  the left operand of the comparison
  <scalar_2>  the right operand of the comparison
  returns     the result of comparing <scalar_1> with <scalar_2>


(> <scalar_1> <scalar_2>)
  <scalar_1>  the left operand of the comparison
  <scalar_2>  the right operand of the comparison
  returns     the result of comparing <scalar_1> with <scalar_2>


(>= <scalar_1> <scalar_2>)
  <scalar_1>  the left operand of the comparison
  <scalar_2>  the right operand of the comparison
  returns     the result of comparing <scalar_1> with <scalar_2>


(abs <scalar_number>)
  <scalar_number>      the number
  returns              the absolute value of the number


(acos <scalar_number>)
  <scalar_number>     the number
  returns     the arccosine of the number


(address-of <expression>)
  <expression>      the node
  returns     the address of the node


(alloc <integer>)

  <integer>       the number of nodes to allocate
  returns     the old number of nodes to allocate


(and {<expression>})
  <expression>  the expressions to be ANDed
  returns       nil if any expression evaluates to nil;
                otherwise the value of the last expression
                (evaluation of expressions stops after the first
                expression that evaluates to nil)


(append {<list>})
  <list>      lists whose elements are to be appended
  returns     the new list


(apply <function_expression> <list>)
  <function_expression>   a symbol with function binding, a lambda
                            expression, else an error
  <list>      the list of arguments to which <function_expression>
               is applied
  returns     the result of applying the function to the arguments


(arc <integer_1> <integer_2> <integer_3> <integer_4>
     <integer_5> <integer_6> <integer_7> <integer_8> )

  <integer_1><integer_2>     x- and y-axis coordinates  of upper-left
                             corner of bounding rectangle
  <integer_2><integer_3>     x- and y-axis coordinates of lower-right
                             corner of bounding rectangle
  <integer_5><integer_6>     x- and y-axis coordinates of start vector
  <integer_7><integer_8>     x- and y-axis coordinates of end vector

  returns       true, if the arc is successfully drawn;
                otherwise, nil

  side effect   draws an elliptical arc, the center of which is the
                center of the bounding rectangle, starting from the
                intersection point with the start vector and ending
                at the intersection point with the end vector
```

```
(aref <array> <integer>)
  <array>     the array
  <integer>   the array index
  returns     the value of the array element
```

```
(asin <scalar_number>)
  <scalar_number>      the number
  returns     the arcsine of the number
```

```
(assoc <symbol> <list> [<key> <test>])
  <symbol>  the symbol to find
  <list>    the assoc list of the form ([{(<symbol> . <expression>)}])
  <key>     the keyword :test or :test-not
  <test>    the test function (defaults to eql)
  returns   the assoc list entry testing true on <symbol>
```

```
(atan <scalar_number>)
  <scalar_number>      the number
  returns     the arctangent of the number
```

```
(atom <expression>)
  <expression>        the expression to check
  returns             t if the value is an atom, nil otherwise
```

```
(backquote <expression>)
  <expression>        the template
  returns             a copy of the template with comma and comma-at
                      expressions expanded:
                      (i) if (comma <expression>) in scope of backquote,
                      then replace with (eval <expression>).
                      (ii) if (comma-at <expression>) in scope of
                      backquote, then replace with (eval <expression>)
                      spliced in.
```

```
(baktrace [<integer>])
  <integer>           the number of levels (defaults to all levels)

  returns     nil
```

```
(boundp <symbol>)
  <symbol>        the symbol
  returns     t if a value is bound to the symbol, nil otherwise
```

```
(box <integer_1> <integer_2>
     <integer_3> <integer_4>
     <integer_5>)
  <integer_1>     upper-left row (top margin)
  <integer_2>     upper-left column (left margin)
  <integer_3>     lower-right row (bottom margin)
  <integer_4>     lower-right column (right margin)
  <integer_5>     border lines to draw
                    2 = draw double-line
                    otherwise draw single-lines
  returns         true
  side-effects    draws a double- or single-line bordered box of
                  the designated size.  Usually the interior of
                  this region is marked with the make-window command.
```

```
(break [<string> [<expression>]])
  <string>       the break message, defaults to "**BREAK**"
  <expression>   (eval <expression>) is printed after the message
  returns        nil when continued from the break loop
```

```
(car <list>)
  <list>       the list node
  returns      the head of the list node
```

```
(catch <tag> [{<expression>}])
  <tag>            the catch tag
  <expression>     expressions to evaluate
  returns          the value (eval (last {<expression>})) or the
                   value of a throw expression
```

```
(cdr <list>)
  <list>       the list node
  returns      the tail of the list node
```

```
(cerror <string_1> <string_2> [<expression>])
  <string_1>     the continue message
  <string_2>     the error message
  <expression>   (eval <expression>) is printed after the message
  returns        nil when continued from the break loop
```

```
(cfprintf1 <file_pointer> <string> <expression>)
  <file_pointer>     the output sink
  <string>           any format conversion strings in C
  <expression>       the expression to be printed
  returns            true
  side-effect        the value (eval <expression>) is printed
                     in the format specified by <string>
```

```
(char <string> <index>)
  <string>     the string
  <index>      the string index (zero relative)
  returns      the ascii code of the character
```

```
(cis <scalar_number>)
  <scalar_number>      the number
  returns      the result of evaluating cis <scalar_number>
```

```
(class <selector> [<expression>])
  <selector>       the method selector
  <expression>     the (optional) argument expression
  returns          the value obtained by applying the method to
                   list of (eval <expression>):

    :new
    returns     the new class object

    :isnew <list_1> [<list_2>[<object>]]
    <list_1>    the list of instance variable symbols
  <list_2>     the list of class variable symbols
  <object>     the superclass (default is Object)
  returns     the new class object

  :answer <selector> ([{<symbol>}]) [{<expression>}]
  <selector>       the message symbol
  ([{<symbol>}])   the list of formal arguments:
                   (
                    [{<farg>}]
                    [&optional [{<oarg>}]]
                    [&rest <rarg>]
                    [&aux [{<aux>}]]
                   )
                  where
                 <farg>      is a formal argument
                 <oarg>      is an optional argument
                 <rarg>      bound to the rest of the arguments
                 <aux>       is an auxiliary variable
  [{<expression>}] the body of the method
  returns     the object
```

```
(clean-up)
 returns     never returns
```

```
(clear)
  returns       true
  side-effect   clears screen in active window
```

```
(cleareol)
  returns       true
  side-effect  clears to end of current line
```

```
(cleareos)
 returns      true
 side-effect  clears to end of current window
```

```
(clearregion <integer_1><integer_2><integer_3><integer_4><integer_5>)
  <integer_1>   the row in active window
  <integer_2>   the column in active window
  <integer_3>   the width of window region to be cleared
  <integer_4>   the number of lines to be cleared
  <integer_5>   the color attribute
  returns       true
  side-effect   starting at (<integer_1> <integer_2>), clears
                <integer_3> positions on each of the next
                <integer_4> rows, using the color attribute
                <integer_5>
```

```
(close <file_pointer>)
 <file_pointer>  the file pointer
 returns          nil
```

```
(colhome)
 returns      column position of cursor after side-effect
 side-effect  positions cursor at left-margin of active
              window
```

```
(complex <scalar_number_1> <scalar_number_2>)
  <scalar_number_1>  realpart of complex number
  <scalar_number_2>  imagpart of complex number
  returns  the complex number #C(<scalar_number_1> <scalar_number_2>)
```

```
(cond {<cond_clause>})
  <cond_clause> a list of cond-clauses
  returns       (eval (last <rest_expression>)) of the first
                cond-clause such that (eval <test_expression> is
                non-nil; otherwise nil
```

```
(cons <expression_1> <expression_2>)
  <expression_1>     the head of the new list node
  <expression_2>     the tail of the new list node
  returns            the new list node
```

```
(consp <expression>)

  <expression>      the expression to check
  returns           t if the value is a list node, nil otherwise
```

```
(continue)
  returns     never returns
```

```
(cos <scalar_number>)
  <scalar_number>      the number
  returns     the cosine of the number
```

```
(cosh <scalar_number>)
  <scalar_number>      the number
  returns     the hyperbolic cosine of the number
```

```
(curshift  <integer_1> <integer_2>)
  <integer_1>   row increment
  <integer_2>   column increment
  returns        row of cursor in active window after side-effect
  side-effect   shifts cursor to position:
                   row = (+ (get-row-pos) <integer_1>)
                   col = (+ (get-col-pos) <integer_2>)
```

```
(date)
  returns    system time and date (string)
```

```
(defun <symbol_1> ([{<symbol_2>}]) [{<expression>}])
(defmacro <symbol_1> ([{<symbol_2>}]) [{<expression>}])
  <symbol_1>       symbol being defined
  <symbol_2>       list of formal arguments:
                    (
                     [{<farg>}]
                     [&optional [{<oarg>}]]
                     [&rest <rarg>]
                     [&aux [{<aux>}]]
                    )
                    where
```

```
              <farg>     is a formal argument
              <oarg>     is an optional argument
              <rarg>     bound to the rest of the arguments
              <aux>      is an auxiliary variable
  <expression>      the body of the function
  returns           <symbol_1>
```

```
(delete <expression> <list> [<key> <test>])
  <expression>    the expression to delete
  <list>          the list
  <key>           the keyword :test or :test-not
  <test>          the test function (defaults to eql)
  returns         the node <list> with the matching expressions deleted
```

```
(do <loop_binding_list> <loop_test> [<loop_body>])
(do* <loop_binding_list> <loop_test> [<loop_body>])
  <loop_binding_list>  the variable bindings which are either:
                        1)  a symbol, initialized to nil
                        2)  a <loop_binding> of the form:
                             (<symbol> <initial> [<next>])
                     where:
                       <symbol>  is the symbol to bind
                       <initial> during loop initialization, <symbol>
                                 is set to (eval <initial>)
                       <next>    before each iteration, <symbol> is
                                 set to (eval <next>)

  <loop_test>     the loop termination test
                  if (eval <test_expression>) is non-nil, then
                  all of the expressions in <rest_expression> are
                  evalnated, and (eval (last <rest_expression>)) is
                  returned

  <loop_body>     the body of the loop, which is an instance of
                  <prog_body>

  returns       if <loop_test> activated, then the value of this
                clause is returned; otherwise, the valne returned is
                the valne of a return function in <loop_body>
```

```
(dolist (<symbol> <list> [<expression>]) [<loop_body>])
  <symbol>      the symbol to bind to each list element
  <list>        the list to iterate through
  <expression>  the result expression (the default is nil)
  <loop_body>   the body of the loop
  returns       when all elements of <list> have been processed,
                then (eval <expression>) is returned; otherwise,
                the valne returned is the valne of a return
                function in <loop_body>
```

```
(dos <string>)
  <string>    the DOS command to execnte
  returns     true or error code
```

```
(dotimes (<symbol> <expression_1> [<expression_2>]) [<loop_body>])
  <symbol>        the loop variable, which is initialized to 0
                  and which is set to (eval (1+ <symbol>)) before
                  each iteration
  <expression_1>  the number of times to loop
  <expression_2>  when <symbol> is (1- <expression_2>), then return
                  (eval <expression_2>)
  <loop_body>     the body of the loop
  returns         when loop count satisfied, then return
                  (eval <expression_2>); otherwise,
                  the value returned is the value of a return
                  function in <loop_body>
```

```
(ellipse <integer_1>
         <integer_2> <integer_3>
         <integer_4> <integer_5>)

 <integer_1>                 fill control parameter:
                               2 = draw only outline (no fill)
                               3 = fill nsing current color and
                                   fill mask
 <integer_2><integer_3>      x- and y-axis coordinates  of upper-left
                             corner of bounding rectangle
 <integer_4><integer_5>      x- and y-axis coordinates of lower-right
                             corner of bounding rectangle

 returns       true, if the ellipse is successfully drawn;
               otherwise, nil

 side effect   draws an ellipse whose center is the center of the
               bounding rectangle
```

```
(eq <expression_1> <expression_2>)
  <expression_1>     the first expression
  <expression_2>     the second expression
  returns            t if they are equal, nil otherwise
```

```
(eql <expression_1> <expression_2>)
  <expression_1>     the first expression
  <expression_2>     the second expression
  returns            t if they are equal, nil otherwise
```

```
(equal <expression_1> <expression_2>)
  <expression_1>     the first expression
  <expression_2>     the second expression
  returns            t if they are equal, nil otherwise
```

```
(error <string> [<expression>])
  <string>      the error message
  <expression>  (eval <expression>) is printed after the message
  returns        never returns
```

```
(errset <expression> [<pflag>])
  <expression> the expression to execnte
  <pflag>      flag to control printing of the error message
  returns      the valne of the last expression consed with nil
               or nil on error
```

```
(eval <expression>)
  <expression>      the expression to be evaluated
  returns           the result of evaluating the expression
```

```
(evalhook <expression_1> <expresion_2> <expression_3> [<env>])
  <expression_1>   the expression to evaluate
  <expression_2>   *evalhook* is set to (eval <expression_1>)
  <expression_3>   *applyhook* is set to (eval <expression_3>)
  <env>       the environment (default is nil)
  returns     the result of evaluating the expression
```

```
(exit)
  returns     never returns
```

```
(exp <scalar_number>)
  <scalar_number>    the number
  returns     e to the <scalar_number> power
```

```
(expand <integer>)
  <integer>       the number of segments to add
  returns     the number of segments added
```

```
(expt <scalar_number_1> <scalar_number_2>)
  <scalar_number_1>  the number
  <scalar_number_2>  the power
  returns     <scalar_number_1> to the <scalar_number_2> power
```

```
(fix <scalar_number>)
  <scalar_number>  the number
  returns          if <scalar_number> is a <float>, then the truncation
                   to <integer>; if <scalar_number> is a <integer>, the
                   <scalar_number>
```

```
(flatc <expression>)
  <expression>  the expression
  returns       the number of characters to print <expression>
                using princ
```

```
(flatsize <expression>)
  <expression>  the expression
  returns       the number of characters to print <expression>
                using print
```

```
(float <scalar_number>)
  <scalar_number>      the number
  returns          if <scalar_number> is an <integer>, then the
                   coresponding <float>; <scalar_number> is a <float>,
                   then <scalar_number>
```

```
(floodfill <integer_1> <integer_2> <integer_3>)
  <integer_1><integer_2>     x- and y-axis coordinates of
                             start point
  <integer_3>   fill boundry color
  returns       true, if the fill is successfully drawn;
                otherwise, nil
  side effect   fills an area of the display usinf current color
                and fill mask, starting at (<integer_1>, <integer_2>).
                If this point lies inside the figure, the interior is
                filled.  If this point lies outside, the background
                is filled.  Filling stops at the fill boundry color.
```

```
(fseek <fp> <integer_1> <integer_2>)
  <fp>          the file pointer
  <integer_1>   offset for position in file
  <integer_2>   origin from which offset is calculated:
                   0 = origin is beginning of file
                   1 = origin is current position in file
                   2 = origin is end of file
  returns       offset for position in file
  side-effect   changes file position as required for next IO
                operation
```

```
(ftell <fp>)
  <fp>          the file pointer
  returns       current position of pointer in file
```

```
(function <expression>)
  <expression>      the function to be quoted
  returns           a function closure
```

```
(gc)
  returns     nil
```

```
(gensym [<tag>])
  <tag>       string or number
  returns     the new symbol
```

```
(get-margins)
returns      a list containing the margins of the current window
             in the following order: upper_left row, upper_left col
             lower_right row, and lower_right col
```

```
(getbkcolor)
  returns     the pixel value of the current background color
```

```
(getcolor)
  returns     the pixel value of the current color
```

```
(getcurcol)
  returns   current column of cursor in active window
```

```
(getcurrow)
  returns    current row of cursor in active window
```

```
(getkey)
  returns    the next key stroke (integer)
```

```
(getpixel <integer_1> <integer_2>)
  <integer_1><integer_2>    x- and y-axis coordinates
  returns       if successful, the pixel value at position
                (<integer_1>, <integer_2>); otherwise, -1


(getprop <symbol_1> <symbol_2>)
  <symbol_1>          the symbol
  <symbol_2>          the property symbol
  returns             the property value or nil


(gettextcolor)
  returns    the color attribute of active window


(go <tag>)
  <tag>       the tag (quoted)
  returns     never returns


(hash <symbol> <integer>)
  <symbol>    the symbol or string
  <integer>   the table size
  returns     the hash index


(imagpart <complex>)
  <complex>      complex number
  returns     the imaginary part of <complex>


(intern <string>)
  <string>     the symbol's print name string
  returns      the new symbol


(kbhit)
  returns    a system-dependent nonzero value if a key has been
             hit; otherwise, it returns 0.


(lambda ([{<symbol>}]) [{<expression>}])
  <symbol>            the formal parameter symbol
  <expression>        expressions of the function body
  returns             the function closure


(last <list>)
  <list>      the list
  returns     the last list node in the list


(length <expression>)
  <expression>  the expression whose length is to be determined
  returns       if <expression> is a list, string or array, then
                the length of <expression>; otherwise error


(lineto <integer_1> <integer_2>)
  <integer_1><integer_2>    x- and y-axis coordinates
  returns       true, if the line is successfully drawn;
                otherwise, nil
  side effect   draws a line from the current position to
                coordinate (<integer_1>, <integer_2>) using
                current color and line style.  If no error
                occurs, lineto sets current position to
                (<integer_1>, <integer_2>).


(list {<expression>})
  <expression>      expressions to be combined into a list
  returns           the new list


(listp <expression>)
  <expression>      the expression to check
  returns           t if the value is a list node or nil, nil otherwi


(ln <scalar_number>)
  <scalar_number>    the number
  returns     the natural logarithm of <scalar_number>


(load <fname> [<vflag> [<pflag>]])
  <fname>     the filename string or symbol
  <vflag>     the verbose flag (default is t)
  <pflag>     the print flag (default is nil)
  returns     the filename


(log <scalar_number_1> <scalar_number_2>)
  <scalar_number_1>  the number
  <scalar_number_2>  the power
  returns     logarithm of <scalar_number_1> to the base
               <scalar_number_2>


(logand {<integer>})
  <integer>      the numbers
  returns     the result of the and operation


(logior {<integer>})
  <integer>      the numbers
  returns     the result of the inclusive or operation


(lognot <integer>)
  <integer>      the number
  returns     the bitwise inversion of number


(logxor {<integer>})
  <integer>      the numbers
  returns     the result of the exclusive or operation


(make-array <integer>)
  <integer>   the size of the new array, assumimng zero origin
  returns     the new array


(make-symbol <string>)
```

```
  <string>      the symbol's print name string
  returns       the new symbol
```

```
(make-window <integer_1> <integer_2> <integer_3> <integer_4>)
  <integer_1>     upper-left row (top margin)
  <integer_2>     upper-left column (left margin)
  <integer_3>     lower-right row (bottom margin)
  <integer_4>     lower-right column (right margin)
  returns         true, if error returns nil
  side-effects    sets margins for active window
```

```
(mapc <function_expression> {<list>]})
  <function_expression>   the function expression to be mapped
  <list>       a list for each argument of the function
  returns      the first list of arguments
```

```
(mapcar <function_expression> {<list>})
  <function_expression>   the function expression to be mapped
  <list>      a list for each argument of the function
  returns     a list of the values returned
```

```
(mapl <function_expression> {<list>})
  <function_expression>   the function expression to be mapped
  <list>      a list for each argument of the function
  returns      the first list of arguments
```

```
(maplist <function_expression> {<list>})
  <function_expression>   the function expression to be mapped
  <list>      a list for each argument of the function
  returns      a list of the values returned
```

```
(max {<scalar_number>})
  <scalar_number> the expressions to be checked
  returns       the largest number in the list
```

```
(mem)
  returns     nil
```

```
(member <expression> <list> [<key> <test>])
  <expression>       the expression to find
  <list>             the list to search
  <key>              the keyword :test or :test-not
  <test>             the test function (defaults to eql)
  returns            the remainder of the list from <expression>
```

```
(min {<scalar_number>})
  <scalar_number>  the numbers to be compared
  returns        the smallest number in the list
```

```
(minusp <scalar_number>)
  <scalar_number>  the number to test
  returns          t if the number is negative, nil otherwise
```

```
(moveto <integer_1> <integer_2>)
  <integer_1><integer_2>    x- and y-axis coordinates
  returns       true
  side effect   moves current position to the point
                (<integer_1>, <integer_2>).
```

```
(nconc {<list>]})
  <list>       lists to concatenate
  returns      the result of destructively concatenating the lists
```

```
(not <expression>)
  <expression>      the expression to check
  return            t if the expression is nil, nil otherwise
```

```
(nth <integer> <list>)
   <integer>   the position index of the element to return,
                assuming zero origin
   <list>      the list
   returns     if (length <list>) >= <integer>, the n'th car;
                otherwise nil
```

```
(nthcdr <integer> <list>)
  <integer>   the position index of the sub-list to return,
               assuming zero origin
  <list>      the list
  returns     if (length <list>) >= <integer>, the <integer>'th;
               otherwise nil
```

```
(null <expression>)
  <expression>      the list to check
  returns           t if the list is empty, nil otherwise
```

```
(numberp <expression>)
  <expression>      the expression to check
  returns           t if the expression is a number, nil otherwise
```

```
(object <selector> [<expression>])
  <selector>     the method selector
  <expression>   the (optional) argument expression
  returns        the value obtained by applying the method to
                 list of (eval <expression>):

    :show
    returns     the object

    :class
    returns     the class of the object

    :isnew
    returns     the object

    :sendsuper <selector> [<expression>]
    <selector>    the message selector
    <expression> the message arguments
    returns      the result of sending the message
```

```
(opena <file_name>)
  <file_name>   the file name string or symbol
  returns       the <file_pointer>
  side-effect   <file_direction> is set to append and file position
                is set to beginning of file if no existent file,
                otherwise file position is set just before end of
                existent file
```

```
(openi <file_name>)
  <file_name>     the file name string or symbol
  returns         the <file_pointer>
  side-effect     <file_direction> is set to input and file position
                  is set to beginning of file
```

```
(openio <fname>)
  <fname>       the file name string or symbol
  returns       the <file_pointer>
  side-effect   <file_direction> is set to input-output and file
                position is set to beginning of file
```

```
(openo <file_name>)
  <file_name>     the file name string or symbol
  returns         the <file_pointer>
  side-effect     <file_direction> is set to output and file position
                  is set to beginning of file
```

```
(or {<expression>})
  <expression>  the expressions to be ORed
  returns       nil if all expressions evaluate to nil;
                otherwise the value of the first non-nil expression
                (evaluation of expressions stops after the first
                expression that does not evaluate to nil)
```

```
(peek <integer>)
  <integer>     the address to peek at
  returns      the value at the specified address (integer)
```

```
(peek-char [<flag> [<file_pointer>]])
  <flag>          flag for skipping white space (default is nil)
  <file_pointer> the input source (default is standard input)
  returns         the character (integer)
```

```
(phase <complex>)
  <complex>      complex number
  returns     the phase
```

```
(pie <integer_1> <integer_2> <integer_3> <integer_4>
     <integer_5> <integer_6> <integer_7> <integer_8> <integer_9> )

  <integer_1>                fill control parameter:
                               2 = draw only outline (no fill)
                               3 = fill using current color and
                                   fill mask
  <integer_2><integer_3>    x- and y-axis coordinates  of upper-left
                            corner of bounding rectangle
  <integer_4><integer_5>    x- and y-axis coordinates of lower-right
                            corner of bounding rectangle
  <integer_6><integer_7>    x- and y-axis coordinates of start vector
  <integer_8><integer_9>    x- and y-axis coordinates of end vector

  returns       true, if the pie is successfully drawn;
                otherwise, nil

  side effect   draws a pie-shaped wedge by drawing an elliptical arc
                whose center and end points are joined by lines.
```

```
(plusp <scalar_number>)
  <scalar_number>  the number to test
  returns          t if the number is positive, nil otherwise
```

```
(poke <integer> <expression>)
  <integer>     the address to poke
  <expression>  (eval <expression>) is the value to poke
  returns        the value
```

```
(princ <expression> [<file_pointer>])
  <expression>    the expressions to be printed
  <file_pointer>  the output sink (default is standard output)
  returns         the expression
  side-effect     file position is set to point to next byte after
                  the last <char> in <expression>
```

```
(print <expression> [<file_pointer>])
  <expression>    the expressions to be printed
  <file_pointer>  the output sink (default is standard output)
  returns         the expression
  side-effect     after the last byte of <expression>, a CR or LF
                  is printed, and file position is set to next byte
```

```
(prog <binding_list> <prog_body>)
(prog* <binding_list> <prog_body>)
  <binding_list>  the variable bindings each of which is either:
                  1)  a symbol (initialized to nil)
                  2)  a list whose car is a symbol and whose cadr
                      is an initialization expression
  <prog_body>   expressions to evaluate or tags (symbols)
  returns        nil or the argument passed to the return function
```

```
(putprop <symbol_1> <expression> <symbol_2>)
  <symbol_1>      the symbol
  <expression>    the attribute pair (<symbol_2 (eval <expression>)
                   is addded to property list of <symbol_1>
  <symbol_2>      the property symbol
  returns         (eval <expression>)
```

```
(quote <expression>)
  <expression>      the expression to be quoted
  returns           <expression> unevaluated
```

```
(random <integer>)
  <integer>   the upper bound (integer)
  returns     a random number


(read [<file_pointer> [<expression> [<rflag>]]])
  <file_pointer>  the input source (default is standard input)
  <expression>    the value to return on end of file (default is nil)
  <rflag>         recursive read flag (default is nil)
  returns         <stream_head_expression>, else (eval <expression>)
  side-effect     file position is set to next byte after last byte
                  the expression read


(read-char [<file_pointer>])
  <file_pointer>  the input source (default is standard input)
  returns         the character (integer)
  side-effect     file-position is set to next byte


(read-line [<file_pointer>])
  <file_pointer>   the input source (default is standard input)
  returns          all <char> in <stream> up to LF or LF/CR
  side-effect      file position is set to next byte after LF
                   or LF/CR


(realpart <complex>)
  <complex>      complex number
  returns     the realpart of <complex>


(rectangle <integer_1>
           <integer_2> <integer_3>
           <integer_4> <integer_5>)

  <integer_1>                 fill control parameter:
                                2 = draw only outline (no fill)
                                3 = fill using current color and
                                    fill mask
  <integer_2><integer_3>      x- and y-axis coordinates  of upper-left
                              corner of bounding rectangle
  <integer_4><integer_5>      x- and y-axis coordinates of lower-right
                              corner of bounding rectangle

  returns        true, if the rectangle is successfully drawn;
                 otherwise, nil

  side effect    draws a rectangle using current color and line
                 style


(rem {<integer>})
  <integer>   the numbers
  returns     the result of the remainder operation


(remove <expression> <list> [<key> <test>])
  <expression>      the expression to delete
  <list>       the list
  <key>        the keyword :test or :test-not
  <test>       the test function (defaults to eql)
  returns      the list with the matching expressions deleted


(remprop <symbol_1> <symbol_2>)
  <symbol_1>       the symbol
  <symbol_2>       the property symbol
  returns          nil
  side-effect      the attribute pair (<symbol_2 (eval <expression>)
                   is deleted from the property list of <symbol_1>


(restorescr)
  returns      true
  side-effect  restores textport screen image previously saved by
               (savescreen)


(return [<expression>])
  <expression>    in the scope of prog or prog*, this establishes
                  (eval <expression>) as the value returned;
                  otherwise undefined
  returns         never returns


(reverse <list>)
  <list>      the list to reverse
  returns     a new list in the reverse order


(rowhome)
  returns      row position of cursor after side-effect
  side-effect  positions cursor at top-margin of active
               window


(rplaca <list> <expression>)
  <list>         the list node
  <expression>   the new value for the car of the list node
  returns        the node <list> whose car is (eval <expression>)


(rplacd <list> <expression>)
  <list>        the list node
  <expression>  the new value for the cdr of the list node
  returns       the node <list> whose cdr is (eval <expression>)


(savescr)
  returns       true
  side-effect   moves textport screen image into unused video memory


(scrolldown <integer>)
  <integer>     the number of lines to scroll
  returns       system-dependent integer
  side-effect   scrolls the active window by <integer> lines


(scrollup <integer>)
  <integer>     the number of lines to scroll
  returns       system-dependent integer
  side-effect   scrolls the active window by <integer> lines
```

```
(set <expression_1> <expression_2>)
  <expression_1>   (eval <expression_1> is a symbol, else error
  <expression_2>     value of (eval <expression_1>) is set to
                     (eval <expression>)
  returns          (eval <expression_2>)


(setbkcolor <integer_1>)
  <integer_1>   the desired background color
  returns       true
  side effect   in graphics mode all background pixels are
                immediately changed


(setcliprgn  <integer_1> <integer_2> <integer_3> <integer_4>)
  <integer_1><integer_2>    x- and y-axis coordinates  of upper-left
                            corner of clip region
  <integer_3><integer_4>    x- and y-axis coordinates of lower-right
                            corner of clip region
  returns       true
  side effect   limits the display of graphic objects to those
                components within the clip region


(setcolor <integer_1>)
  <integer_1>   the color number (masked to be in range)
  returns       true


(setcursor <integer_1> <integer_2>)
  <integer_1>   the row in active window
  <integer_2>   the column in active window
  return        true
  side-effect   positions the cursor in the active window
                at (<integer_1> <integer_2>)


(setf {<placeform> <expression>})
 <placeform>  the field specifier (quoted):
            <symbol>                  set value of a symbol
            (car <expression>)          set car of a list node
            (cdr <expression>)          set cdr of a list node
            (nth <integer> <expression>)   set nth car of a list
            (aref <array> <integer>)    set nth element of an array
            (getprop <symbol> <expression>)   set value of a property
            (symbol-value <symbol>)     set value of a symbol
            (symbol-plist <symbol>)     set property list of a symbol
  <expression> the value of <placeform> is (eval <expression>)
  returns     (eval <expression>)


(setpixel <integer_1> <integer_2>)
  <integer_1><integer_2>    x- and y-axis coordinates
  returns       true
  side effect   sets the pixel at (<integer_1>, <integer_2>)
                to current color.


(setq {<symbol> <expression>})
  <symbol>          the symbol being set
  <expression>      value of <symbol> is set to (eval <expression>)
  returns           (eval <expression>)
```

```
(settextcolor  <integer>)
  <integer>    the color attribute for active window
  returns      true


(setvideomode <integer>)
  <integer>      the selected graphics mode
  returns        nil
  side-effects   system-dependent modifications of screen


(setviewport  <integer_1> <integer_2> <integer_3> <integer_4>)
  <integer_1><integer_2>     x- and y-axis coordinates  of upper-left
                             corner of viewport
  <integer_3><integer_4>     x- and y-axis coordinates of lower-right
                             corner of viewport
  returns       true
  side effect   limits the display of graphic objects in precisely
                the same manner as setcliprgn, and then sets the
                logical origin to point (<integer_1>, <integer_2>)


(sin <scalar_number>)
  <scalar_number>  the number
  returns     the sine of the number


(sinh <scalar_number>)
  <scalar_number>      the number
  returns     the hyperbolic sine of the number


(sqrt <scalar_number>)
  <scalar_number>   the number
  returns     the square root of the number


(strcat {<string>})
  <string>      the strings to concatenate
  returns       the result of concatenating the strings


(string <integer>)
  <integer>    a small integer
  returns      the one character string whose ASCII code is
                <integer>


(sublis <alist> <expression> [<key> <test>])
  <alist>        the association list
  <expression>   the expression in which to do the substitutions
  <key>          the keyword :test or :test-not
  <test>         the test function (defaults to eql)
  returns        the expression with substitutions


(subst <to> <from> <expression> [<key> <test>])
  <to>           the new expression
  <from>         the old expression
  <expression>   the expression in which to do the substitutions
  <key>          the keyword :test or :test-not
  <test>         the test function (defaults to eql)
  returns        the expression with substitutions
```

```
(substr <string> <integer_1> [<integer_2>])
  <string>      the string
  <integer_1>   the starting position
  <integer_2>   the length (default is rest of string)
  returns       substring starting at <integer_1> for
                 <integer_2>
```

```
(symbol-name <symbol>)
  <symbol>       the symbol
  returns        the symbol's print name
```

```
(symbol-plist <symbol>)
  <symbol>       the symbol
  returns        the symbol's property list
```

```
(symbol-value <symbol>)
  <symbol>       the symbol
  returns        the symbol's value
```

```
(symbolp <expression>)
  <expression>      the expression to check
  returns           t if the expression is a symbol, nil otherwise
```

```
(tan <scalar_number>)
  <scalar_number>      the number
  returns     the tangent of the number
```

```
(tanh <scalar_number>)
  <scalar_number>      the number
  returns     the hyperbolic tangent of the number
```

```
(terpri [<file_pointer>])
  <file_pointer>   the output sink (default is standard output)
  returns           nil
```

```
(throw <tag> [<expression>])
  <tag>          the catch tag
  <expression>   (eval <expression>) is the value for the associated
                  catch to return;
                  otherwise nil
  returns        never returns
```

```
(top-level)
  returns     never returns
```

```
(transcript [<fname>])
  <fname>     file name string or symbol
              (if missing, close current transcript)
  returns     t if the transcript is opened, nil if it is closed
```

```
(tstamp)
  returns   time stamp (string)
```

```
(type-of <expression>)
  <expression>  the expression to return the type of
  returns        nil if the value is nil otherwise one of the symbols:
              :SYMBOL  for symbols
              :OBJECT  for objects
              :CONS    for conses
              :SUBR    for built-ins with evaluated arguments
              :FSUBR   for built-ins with unevaluated arguments
              :STRING  for strings
              :FIXNUM  for integers
              :FLONUM  for floating point numbers
              :COMPLEX for complex numbers
              :FILE    for file pointers
              :ARRAY   for arrays
```

```
(write-char <char> [<file_pointer>])
  <char>          the character to put (integer)
  <file_pointer>  the output sink (default is standard output)
  returns         the character (integer)
  side-effect     file poistion is set to next byte
```

```
(zerop <scalar_number>)
  <scalar_number>  the number to test
  returns          t if the number is zero, nil otherwise
```

NIST-114A (REV. 3-90)	U.S. DEPARTMENT OF COMMERCE NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY	1. PUBLICATION OR REPORT NUMBER NISTIR 4388
		2. PERFORMING ORGANIZATION REPORT NUMBER
BIBLIOGRAPHIC DATA SHEET		3. PUBLICATION DATE SEPTEMBER 1990

4. TITLE AND SUBTITLE

AMPLE Core Interpreter: User's Guide

5. AUTHOR(S)

J. C. Boudreaux

6. PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS)	7. CONTRACT/GRANT NUMBER
U.S. DEPARTMENT OF COMMERCE NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY GAITHERSBURG, MD 20899	8. TYPE OF REPORT AND PERIOD COVERED

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

10. SUPPLEMENTARY NOTES

11. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

The Automated Manufacturing Programming Language Environment (AMPLE) system was developed in the Center for Manufacturing Engineering to provide a uniform environment for the construction of control interfaces to industrial processes. The AMPLE Core Interpreter, Version 1.0 is a working prototype, implemented in MicroSoft C 5.0 for PC/AT-class personal computers under MS-DOS. The User's Guide is an introduction to the prototype which is being circulated at this time to provide the manufacturing community and other potential users with an operational specification of AMPLE. For further information or to obtain a copy of the prototype on a 5.25" double sided/double density floppy diskette, please write to:

J. C. Boudreaux
Center for Manufacturing Engineering
National Institute of Standards and Technology
Bldg 233, Room A107
Gaithersburg, MD 20899

12. KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)

AMPLE; programming language environment; very-high level language.

13. AVAILABILITY	14. NUMBER OF PRINTED PAGES
[X] UNLIMITED	84
[ ] FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).	
[ ] ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.	15. PRICE A05
[X] ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.	

ELECTRONIC FORM