APPENDIX A

Quick Reference to Dalek's Syntax

This section begins with an index to Dalek's major commands (many of which are inherited unchanged from gdb). The index also includes most predefined debugger functions (the leading '$' is considered significant for collation purposes). After the index come the commands and predefined functions, which are organized as follows:

INDEX

A.1. Inquiring about Status

1. **info breakpoints**
   Describes all breakpoints, or with one argument, just that breakpoint number. Shows whether enabled, and gives core address and file/line number.

2. **info events**
   Describes all events, or with one argument, of that particular event.

3. **info udc**
   Describes all User-Defined-Commands, or with one argument, of that particular User-Defined-Command.

4. **info udf**
   Describes all User-Defined-Functions, or with one argument, of that particular User-Defined-Function.

5. **info internal-functions**
   Describes all predefined functions, or with one argument, of that particular predefined function.

6. **info functions** REGEXP
   Lists all inferior procedure names, or those matching REGEXP.

7. **info variables** REGEXP
   Lists all global and static variable names, or those matching REGEXP.

8. **info address** VAR
   Describes the core address or register where variable VAR is stored.

9. **info line** NUM
   Describes the range of core addresses where the code for the specified source line is stored.

10. **info frame** ADDR
    All about selected stack frame, or frame at ADDR.

11. **info args**
    Describes argument variables of the currently selected stack frame.

12. **info locals**
    Describes local variables of the currently selected stack frame.

A.2. Event Commands

13. **event define** ‘name_of_event (‘optional_attrib_names) +‘lower_level_event
    Define an event. The ‘lower_level_events (usually more than one) are events whose activation will in turn cause this event to be activated, unless we replace the ‘+’ by a ‘-’ (minus) sign preceding them in the event definition. For each lower level event mentioned in the definition, a separate queue will be attached to the new event and will hold tokens representing instances of those lower level events as they are activated. The predefined function **$qlen**() counts the number of tokens in a queue, and the attributes of these event instances can be accessed by the **$value**() predefined function.

14. **event raise** ‘event_name (actual_parm1, *missing_parm_2* , actual_parm3)
    Cause Dalek to begin interpreting that primitive event’s command block. In practice, this would normally appear within a block of commands attached to a breakpoint. As shown in the above syntax, it is optional whether to instantiate *all* attributes immediately.

15. **event set-attribute** ‘attrib_name expression
    Set the value of an attribute passed from this event to the upper levels. For complex primitive events, the user might need to use this command within an **if** statement in the body of the event, rather than instantiating the attribute at the time the event is raised.

16. **event remove** ‘eventname, N
    Remove the Nth element from the "‘eventname" queue of the current event.

17. **event dont-propagate**
    Continue to process the current event’s commands, but do *not* activate any higher level event that depends on it.

18. **event exit**
    Quit processing this event. Typically preceded by **event dont_propagate**.

19. **event str-def** Y/N
    Specify default handling for attributes of type (char*). If ‘Y’ then we copy the *current contents* of the target string. Else if ‘N’, the attribute is merely the *address* in the pointer.

A.3. Predefined Functions Used Inside an Event

20. **$value()**
    Given three parameters, **$value**('lower_level_event_name, queue_position, 'low_level_attrib_name) returns the value of that constituent occurrence's attribute in the current high-level event's queues. Event instances in a queue are denoted in their temporal order, so that '1' is oldest, then '2', . . . then '0' as a convenient way to indicate the most recent instance.

21. **$qlen()**
    Given one parameter, **$qlen**('lower_level_event_name) returns the length of the specified queue attached to the current high-level event. This length corresponds to the number of occurrences of that particular lower-level constituent event in the high-level event's private queue.

22. **$time()**
    Given two parameters, **$time(**'lower_level_event_name, queue_position) returns the logical timestamp of that constituent occurrence in the current high-level event's queues. The logical timestamp of the first primitive event to occur on the inferior's timeline will be 1, of the second primitive event occurrence, 2, and so forth. All high-level events triggered by a particular primitive event instance and successfully recognized will carry that same timestamp.

23. **$seq_num()**
    Given two parameters, **$seq_num**('lower_level_event_name, queue_position) returns the sequence number of that constituent occurrence in the current high-level event's queues. **$seq_num**() can distinguish the more recent among several higher-level constituents that were all triggered by the same instance of a primitive event.

24. **$get_attribute()**
    Given one parameter, **$get_attribute**('attribute_name) returns the value of that attribute bound by the current instance of the currently executing event. This function may be called by primitive as well as high-level events.

A.4. Access to the Event History

25. **event history** H/P NUM
    Print the last N High-level (or Primitive) events activated. If NUM is not specified, print all High-level (or Primitive) events in history. If neither ‘H’ nor ‘P’ is specified then no selection is performed on the level of the event. In either case, the user will be given the opportunity to further specify his view of the history by choosing the names of only certain events.

26. **$hvalue()**
    Given one parameter, **$hvalue**(‘event_name, pos, ‘attrib_name) is analogous to **$value**(). However, because it accesses the global history list’s queues, it may be invoked at any time, not just inside a high-level event. Although invoking this function allows the user visually to review previous events, more often it is used programmatically, e.g., within a **while** loop. As usual, "pos" refers to instances of events in temporal order so that ‘1’ is oldest, then ‘2’, . . . then ‘0’ as the most recent occurrence.

27. **$hlen()**
    Given one parameter, **$hlen**(‘event_name) is analogous to **$qlen**(). However, because it accesses the global history list’s queues, it may be invoked at any time, not just inside a high-level event.

28. **$htime()**
    Given two parameters, **$htime**(‘event_name, queue_position) returns the logical timestamp of that event occurrence in the global history list’s queues.

A.5. Miscellaneous Items

29. **$time_stamp()**
    Taking no parameters, this function returns the timestamp of the most recent instance among all events.

30. **$savestr()**
    Given one parameter, **$savestr** (var_name_of_type_char*) returns a permanent copy of that string’s *current* contents. This copy will be stored in Dalek’s address space, not that of the inferior. Note that for *event attributes*, there is a flag that can cause this to happen automatically. To set it: "**event str-default** yes" .

31. **$func_length()**
    Given one parameter, **$func_length**("*func_name*") returns the code length (in bytes) of the inferior function named *func_name*. A call to this function can be useful as the second parameter to the **asm-break** command.

32. **source** filename
    Read commands from the named file.

33. **close**
    Close the current **source** file. Do *not* read any more lines from it.

34. **set query** Y/?/N
    Set default response for queries generated from a **source** file. Many commands can cause Dalek to interactively query the user. Yet when such commands are coming from a **source** file, an interactive query could be extremely annoying. This command can be put in a **source** file to specify the default response to any interactive queries that might be generated by commands in that file. If the argument to this command is ‘Y’ or ‘N’, Dalek will accept that as a response to all queries that otherwise might be generated by commands in **source** files. By specifying the ‘?’ parameter, the user indicates that he wants all queries to take their answers only from the keyboard.

35. **list**
    List specified inferior function or line. With no argument, lists ten more lines after or around previous listing.

36. **forward-search** REGEXP
    Search for regular expression from last inferior line listed.

37. **reverse-search** REGEXP
    Search backward for regular expression from last inferior line listed.

38. **udc** new_command_name
    Specify a User-Defined-Command. Definition appears on following lines, one command per line. End with a line of just "end". If you require arguments, instead use **udf** (User-Defined-Function).

A.6. Commands Related to Meta-Debugging

39. **debug** Y/N
    Prints non-interactive command lines before executing them. Will toggle **debug**-mode if no Y/N argument is given.

40. **dstep** NUM
    Sets meta-single-step (self-debugging) mode. In **dstep** mode, Dalek waits for

approval before executing embedded commands. If NUM is negative, Dalek will meta-single-step indefinitely. If NUM is zero, Dalek will cancel meta-single-stepping. If NUM is positive, Dalek will engage meta-single-stepping, but not until *after* it has interpreted NUM embedded commands. At each **dstep**, the user may elect to Edit the displayed command, sKip it, Step into it, or Pause before executing it.

41. **dtrace**
    Just as **backtrace** shows the inferior's stack, **dtrace** shows stacked Dalek command blocks. The main difference is that **dtrace** shows the *next* command to be executed in underlying blocks, *not* the command that invoked the newer blocks.

42. **dlist**
    Display a block of Dalek commands (will show any meta-breakpoints). Sometimes requires 1 argument — the name of the command-block to display. With no argument, friendly defaults similar to those for **edit** are provided.

43. **edit**
    Edit a block of Dalek commands, or view/set/clear meta-breakpoints. Sometimes requires 1 argument — the name of the command-block to edit. With no argument, **edit** will default so as to position the user on the current (or next) command of the currently executing block. These defaults arise naturally when invoking **edit** from the **dstep** (meta single-step) mode. They are also helpful once **pause**d (e.g., after hitting Control-C within a **while** loop, or after encountering a meta-breakpoint). Indeed, the defaults are essential for the user to edit nameless, 'lambda' blocks such as freestanding **if** or **while** statements that are not embedded in any other named block.

44. **pause**
    To be executed inside a block of commands, this stacks pending commands and temporarily returns control to interactive mode. After the user has finished poking around, typing **unpause** will return control to the originating block of commands.

45. **unpause**
    Cause Dalek to resume interpreting subsequent commands in the command block that contained the last **pause** command.

46. **set exit** NUM
    Set internal flag to exit the NUM most recently stacked Dalek command blocks. This command is typically invoked interactively, after a **pause** or Control-C, although it can also be used to avoid tail recursion.

A.7. Breakpoint and Signal Handling Commands

47. **break**

Set breakpoint at specified line or function. Argument may be line number, function name, or "*" and an address. In Dalek the convenience variable "$LAST_BP_SET" holds the number of the last breakpoint defined. Therefore breakpoints can be "named" immediately after defining them and referred to later, e.g.,
set $breakpoint_in_foo = $LAST_BP_SET;
disable $breakpoint_in_foo.

48. **commands** NUM

Set commands to be executed when breakpoint number NUM is hit. With no argument, defaults to the last breakpoint set. The commands themselves follow starting on the next line. Type **end** to indicate the end of the block. Give **silent** as the first command to make the breakpoint silent; then no output is printed when it is hit, except what the commands print.

49. **handle** SIGNUM nostop print nopass **commands**

Specify how to handle a signal. Args are signal number followed by flags. Flags allowed are "(no)stop", "(no)print", "(no)pass". Print means print a message if this signal happens. Stop means reenter debugger if this signal happens (implies print). Pass means let program see this signal; otherwise program doesn't know. Pass and Stop may be combined. Dalek permits **commands** as the *last* flagword. It allows users to bind a command block to the occurrence of this signal.

50. **clear**

Clear breakpoint at specified line or function entry. Argument may be line number, function name, or "*" and an address. See also the **delete** command, which clears breakpoints by number.

51. **delete**

Delete some breakpoints. Give breakpoint numbers as arguments.

52. **disable**

Disable some breakpoints. Give breakpoint numbers as arguments. A disabled breakpoint is not forgotten, but has no effect until reenabled.

53. **enable**

Enable some breakpoints. Give breakpoint numbers as arguments. This is used to cancel the effect of the **disable** command.

54. **condition** N COND
    Stop at breakpoint number N only if COND evaluates to nonzero. N is an integer; COND is a C expression to be evaluated whenever breakpoint N is reached.

55. **ignore** N COUNT
    Ignore the next COUNT crossings of breakpoint number N.

56. **proc-break** REGEXP
    Broadcast a breakpoint (*shared* commands) to all procedures whose names match REGEXP. Dalek will query the user regarding the style of the regular expression (whether ala *csh* or *vi*), and also will ask whether it should check to see if REGEXP matches the names of any *library* procedures. If a **proc-break** command is coming from a **source** file, you may want to precede it by a "**set query** Y/?/N" command to force a certain default response to the two queries.

    If the REGEXP parameter matches very many procedure names, the user may get tired of seeing those names printed to the screen. By typing Control-C, he will be given the opportunity to abort the **proc-break** command entirely, or the option of having it continue its job without writing to the screen.

    All breakpoints in this new set will initially share the same command block. Later, the user may give a **commands** command to re-define the command block of a particular member of this set without altering the command block shared by the set's remaining members. However, if the user **edit**s a shared block, the changes will apply to all remaining members of the broadcast set.

    After a **proc-break** command finishes, the convenience variable $LAST_BP_SET (which normally holds the NUMBER of the last breakpoint the user has set) will instead be of type STRING. It will contain a list of all breakpoint numbers set by this **proc-break** command. By recording its value in another convenience variable (e.g., by "**set** $bp_set_1 = $LAST_BP_SET"), the user may later **disable**, **enable**, or **delete** every breakpoint in this set, e.g., by "**disable** $bp_set_1".

57. **asm-break** &foo, $func_length ("foo"), jsr
    Given three parameters — START-ADDR, LENGTH, and ASM_MNEMONIC — Dalek will scan that region of the inferior's address space for all assembler instructions whose opcode matches ASM_MNEMONIC. By embedding this command in a **source** file and using the **set query** command, the user will have the option of broadcasting breakpoints to all matching instructions, merely displaying them, or deciding whether to set breakpoints on an individual basis as each matching instruction in the region is encountered.

A.8. Commands for Full Programmability

58. **‘ { ’**
    Create a new block for local variables.

59. **local** $new_local_var_name
    Create a local debugging variable.

60. **‘ } ’**
    Pop the topmost block for local variables.

61. **set** VAR = EXP.
    You must type the "=". VAR may be a debugger "convenience" variable (names starting with $), a register (a few standard names starting with $), or an inferior variable. EXP is any expression. If VAR is a convenience variable that does not yet exist, it will be created as a new global.

62. **udf** $User_Defined_Function ($formal_parm_names)
    Specify a $User_Defined_Function. Use the **func-return** command (*not* just **return**) to return a value. End with a line of just "end". These functions can take arguments, unlike User-Defined-Commands.

63. **call** FUNC (actual_parms)
    Call a function in the inferior program, or a $User_Defined_Function().

64. **func-return**
    Return a value from within a User-Defined-Function. Caution: Typing just **return** will make the current function in the *inferior* return!

65. **while** commands **endwhile**
    The **while** statement is supported in all 6 modes: live input, in a user-defined command or $user_defined_func(), when input from a **source** file, and as a breakpoint (or event) command. Nested **while**’s are permitted, as are interactions with the **if** command (e.g., conditional **wbreak** and **wnext** commands, and nesting of **if**’s and **while**’s).

66. **wbreak**
    Exit from the **while** loop. Caution: just typing **break** will set a breakpoint!

67. **wnext**
    Hop to the next iteration of the **while** loop. Caution: **wnext** is analogous to C’s ‘continue’ statement, whereas **cont** in Dalek means allow the inferior to proceed!

68. **if** commands **else** commands **endif**
The **if** statement is supported in all 6 modes: live input, in a user-defined command or $user_defined_func(), when input from a **source** file, and as a breakpoint (or event) command. Nested **if**'s are permitted, as are interactions with the **while** command (e.g., conditional **wbreak** and **wnext** commands, and nesting of **if**'s and **while**'s). Whenever an **if** appears as the *first* word of a new line, it requires its own **endif**.

A.9. Controlling the Inferior's Execution

69. **run**
Start program being debugged. You may specify arguments to give it. Input and output redirection with ">", "<", or ">>" are also allowed. With no arguments, uses arguments last specified.

70. **cont**
Continue program being debugged, after signal or breakpoint.

71. **jump**
Continue program being debugged at specified line or address.

72. **step**
Step program until it reaches a different source line. Argument N means do this N times (or until program stops for another reason).

73. **stepi**
Step one instruction exactly. Argument N means do this N times (or until program stops for another reason).

74. **next**
Step program, proceeding through subroutine calls. Like the **step** command as long as subroutine calls do not happen; when they do, the call is treated as one instruction. Argument N means do this N times (or until program stops for another reason).

75. **nexti**
Step one instruction, but proceed through subroutine calls. Argument N means do this N times (or until program stops for another reason).

76. **call** FUNC (actual_parm_list)
Call a function in the inferior program, or a $User_Defined_Function().

77. **signal** SIGNUM
Continue the program, giving it the specified signal number.

78. **finish**
    Execute until the selected inferior stack frame returns. Upon return, the value returned is printed and put in the value history.

79. **$quiet_finish()**
    Taking no parameters, **$quiet_finish**() allows the inferior function executing in the currently selected frame to return naturally. It then returns, as its own value, the value that was returned by that inferior function.

80. **return**
    Make selected stack frame return to its caller immediately. Control remains in the debugger, but when you continue, execution will resume in the frame above the one now selected. If an argument is given, it is an expression for the value to return.

A.10. Commands for Output

81. **echo** constant_string
    Print a constant string. Give string as argument.

82. **print** EXP
    Print the value of expression EXP. Variables accessible are those of the lexical environment of the selected stack frame, plus all those whose scope is global or an entire file, plus convenience variables and the values returned by Dalek or inferior functions.

83. **printf** "format_string", arg1, arg2, arg3, ..., argn
    This is useful for formatted output in non-interactive commands.

A.11. Accessing the Inferior's Stack

84. **backtrace** COUNT
    Print backtrace of all stack frames, or innermost COUNT frames.

85. **frame** NUM
    Select and print stack frame number NUM. With no argument, print the currently selected stack frame.

86. **down**
    Select and print stack frame called by this one. An argument says how many frames down to go.

87. **up**
    Select and print stack frame that called this one. An argument says how many

frames up to go.

88. **info frame** ADDR
    All about selected stack frame, or frame at ADDR.

89. **$frame()**
    Taking no parameters, **$frame**() returns the absolute frame # of the currently selected frame.

90. **$t_frame()**
    Taking no parameters, **$t_frame**() returns the # of frames above the currently selected frame where the top-level frame is executing. Thus at any breakpoint the stack depth can be learned by calling this function, and, e.g., the frame of *main*() can be selected by commanding "**up $t_frame**()".

91. **$g_frame()**
    Taking no parameters, **$g_frame**() returns the # of frames above the currently selected frame where the nearest function compiled with the "-g" option is located. Thus if, e.g., *malloc*() calls *malloc_at_addr*() which calls *free*() which calls itself, it is still easy programmatically to determine which user-level routine was originally responsible for this chain of library calls.

92. **$file()**
    Given one parameter, **$file**(NUM) returns a string: the filename that contains the procedure executing NUM frames above the currently selected frame. NUM may be a constant, an inferior variable, or a computed integer. It may be negative. If *no* parameter is given, NUM == 0 by default.

93. **$line()**
    Given one parameter, **$line**(NUM) returns the current line number in the function NUM frames above the currently selected frame. NUM may be a constant, an inferior variable, or a computed integer. It may be negative. If *no* parameter is given, NUM == 0 by default.

94. **$func()**
    Given one parameter, **$func**(NUM) returns a string: the name of the function executing NUM frames above the currently selected frame. NUM may be a constant, an inferior variable, or a computed integer. It may be negative. If *no* parameter is given, NUM == 0 by default.

A.12. Measuring the Inferior’s Virtual Time

95. **$ttime()**
Returns total time charged to the inferior process.

96. **$stime()**
Returns UNIX system time charged to the inferior process. Not entirely accurate because some of the cost of the debugger’s calls to ptrace is charged to the inferior as system time.

97. **$utime()**
Returns time inferior spent executing non-system code.

A.13. Proposed New Predefined Functions

98. **$eval()**
Hypothetical new predefined function initially proposed in Section 6.5 and later featured prominently in Section 8.4.1. Given a string, **$eval**() would try to interpret that string, assuming it to be a syntactically valid debugger command. The real benefit of **$eval**() would come not so much from its string-invocation capability, but rather from its ability to redirect all output from that command into a convenience variable of type string. **$eval**() would then return the newly constructed value of that convenience variable as its own return value.

99. **$sed()**
Hypothetical new predefined function proposed in Section 8.4.1. **$sed()** would behave much like the UNIX stream editor of the same name, but instead of operating on a file, it would operate on a string passed to it as a parameter.

100. **$scan_line()**
Hypothetical new predefined function proposed in Section 7.3.2. Given a filename and a pattern as arguments, **$scan_line**() would broadcast breakpoints to every line of source code in that file that matched the given pattern.

101. **$bind_commands()**
Hypothetical new predefined function proposed in Section 7.3.2. Given a filename as argument, **$bind_commands()** would search for all lines beginning with a particular *lint*-like directive denoting "break at this line". At such lines, it would try to read in customized breakpoint command blocks.

References

[Ayer36]
A. Ayer. *Language, Truth, and Logic*. Originally published in 1936, reprinted by Penguin Books. Harmondsworth, Middlesex, England. 1971.

[Baia86]
F. Baiardi, N. De Francesco, and G. Vaglini. ‘‘Development of a debugger for a concurrent language’’. *IEEE Trans. on Software Engineering*, **SE-12:4**, 547-553. April 1986.

[Bate82]
P. Bates and J. Wileden. ‘‘EDL: a basis for distributed system debugging tools’’. *Proc. Fifteenth Hawaii International Conference on System Sciences*, 86-93. 1982.

[Bate83]
P. Bates and J. Wileden. ‘‘High-level debugging of distributed systems: the behavioral abstraction approach’’. *The Journal of Systems and Software*, **3:4**, 255-264. 1983.

[Brue83]
B. Bruegge and P. Hibbard. ‘‘Generalized path expressions: a high-level debugging mechanism’’. *The Journal of Systems and Software*, **3:4**, 265-276. 1983.

[Carn47]
R. Carnap. *Meaning and Necessity, a Study in Semantics and Modal Logic*. University of Chicago Press. Chicago, IL. 1947.

[Chas87]
B. Chase and R. Hood. ‘‘Selective interpretation as a technique for debugging computationally intensive programs’’. *ACM SIGPLAN Notices*, **22:7**, 113-124. July 1987.

[DBX83]
‘‘DBX (1)’’. in *UNIX Programmer’s Manual*, 4.2 Berkeley System Distribution, Vol. 1, University of California, Berkeley. August 1983.

[Deli84]
N. M. Delisle, D. E. Menicosy, and M. D. Schwartz. ‘‘Viewing a programming environment as a single tool’’. *ACM SIGPLAN Notices*, **19:5**, 49-56. May 1984.

[Dona85]
J. Donahue. ‘‘Integration mechanisms in Cedar’’. *ACM SIGPLAN Notices* **20:7**, 245-251. July 1985.

[Fish90]
G. Fisher. Private communication. July 1990.

[Floy60]
R. W. Floyd, B. Kallick, and C. J. Moore. *Advanced Studies in Computer Programming — Quarterly Report No. 4, ARF Project E121*. Armour Research Foundation. Illinois Inst. of Tech., Chicago, IL. July 1960.

[Göde31]
K. Gödel. ‘‘Uber formal unentscheidbare Satze der Principia Mathematica und verwandter Systeme I’’. English translation in *The Undecidable*, 5-38. M. Davis. Raven Press, New York, NY. 1965.

[Gold47]
H. Goldstine and J. von Neumann. ‘‘Planning and coding problems for an electronic computing instrument’’. (report prepared for U. S. Army Ordnance Dept., 1947-1948). in *John von Neumann, Collected Works, Volume V*, 80-235. (A. Taub, Ed.) Pergamon Press, London, 1963.

[Hans78]
D. R. Hanson. ‘‘Event associations in SNOBOL4 for program debugging’’. *Software—Practice and Experience*, **8:2**, 115-129. March-April 1978.

[Hans90]
D. R. Hanson. Private communication. Sept. 1990.

[Hart85]
P. Harter, D. Heimbigner, and R. King. ‘‘IDD: an interactive distributed debugger’’. *International Conference on Distributed Computing Systems*, 498-506. March 1985.

[Heis27]
W. Heisenberg. ‘‘The physical content of quantum kinematics and mechanics’’. reprinted in *Quantum Theory and Measurement*, 62-84. (J. Wheeler and W. Zurek, Ed.) Princeton University Press, Princeton, NJ. 1983.

[Heis58]
W. Heisenberg. *Physics and Philosophy*. Harper & Row, New York, NY. 1958.

[John78]
M.S. Johnson. *The design and implementation of a run-time analysis and interactive debugging environment*. Ph.D. Dissertation, Department of Computer Science, University of British Columbia, TR-78-6. August 1978.

[John81]
M.S. Johnson. ‘‘DISPEL: a run-time debugging language’’. *Computer Languages*, **6**, 79-94. 1981.

[Kain72]
R. Kain. *Automata Theory: Machines and Languages*. McGraw-Hill. 1972.

[Kats79]
H. P. Katseff. ‘‘Sdb: a symbolic debugger’’. in *UNIX Programmer’s Manual, 7th Edition, Vol. 2C* Bell Laboratories, Holmdel, NJ. January 1979.

[Kauf88]
S. Kaufer, R. Lopez, and S. Pratap. ‘‘Saber-C, an interpreter-based programming environment for the C language’’. *Proc. Summer 1988 USENIX Conference*, 161-171. June 1988.

[Kill84]
T. J. Killian. ‘‘Processes as files’’. *Proc. Summer 1984 USENIX Conference*, 203-207. June 1984.

[Knut84]
D. Knuth. ‘‘Literate programming’’. *The Computer Journal.*, **27:2**, 97-111. 1984.

[Laka76]
Imre Lakatos. *Proofs and Refutations: the Logic of Mathematical Discovery*. Cambridge University Press. 1976.

[Lamp78]
L. Lamport. ‘‘Time, clocks, and the ordering of events in a distributed system’’. *Communications of the ACM*, **21:7**, 558-569. July 1978.

[Mara79]
J. F. Maranzano and S. R. Bourne. ‘‘A tutorial introduction to ADB’’. in *UNIX Programmer's Manual, 7th Edition, Vol. 2A* Bell Laboratories, Holmdel, NJ. January 1979.

[Olss90]
R. A. Olsson, R. H. Crawford, and W. W. Ho. ‘‘Dalek: a GNU, improved programmable debugger’’. *Proc. Summer 1990 USENIX Conference*, 221-231. June 1990.

[Olss91]
R. A. Olsson, R. H. Crawford, and W. W. Ho. ‘‘A dataflow approach to event-based debugging’’. *Software—Practice and Experience*. To appear.

[Ridd79]
W. E. Riddle. ‘‘An approach to software system behavior description’’. *Computer Languages*, **4**, 29-47. 1979.

[Salo66]
A. Salomaa. ‘‘Two complete axiom schemes for the algebra of regular events’’. *Journal of the ACM*, **13**, 158-169. 1966.

[Sand78]
E. Sandewall. ‘‘Programming in an interactive environment: the Lisp experience’’. *ACM Computing Surveys*, **10:1**, 35-71. March 1978.

[Satt75]
E. J. Satterthwaite Jr. *Source Language Debugging Tools*. Ph.D. Dissertation, Computer Science Department, Stanford University, 1975.

[Schw83]
R. Schwartz, P. Melliar-Smith, and F. Vogt. ‘‘An interval logic for higher-level temporal reasoning’’. *Proc. Second Annual Symposium on Principles of Distributed Computing*, 173-186. Aug. 1983.

[Solo87]
E. Soloway. ‘‘I can’t tell what in the code implements what in the specs’’. *Proc. Second International Conference on Human-Computer Interaction*, 317-328. August 1987.

[Stall89]
R. M. Stallman. *GDB Manual (The GNU Source-Level Debugger), Third Edition, GDB version 3.1*. Free Software Foundation, Cambridge, MA. January 1989.

[Swee85]
R. E. Sweet. ‘‘The Mesa programming environment’’. *ACM SIGPLAN Notices* **20:7**, 216-229. July 1985.

[Swin85]
D. E. Swinehart, P. T. Zellweger, and R. B. Hagmann. ‘‘The structure of Cedar’’. *ACM SIGPLAN Notices* **20:7**, 230-244. July 1985.

[Teit78]
W. Teitelman. *Interlisp Reference Manual*. Xerox PARC, Palo Alto, CA. October 1978.

[Teit85]
W. Teitelman. ‘‘A tour through Cedar’’. *IEEE Trans. on Software Engineering*, **SE-11:3**, 285-302. March 1985.

[Tars35]
A. Tarski. ‘‘The Concept of Truth in Formalized Languages’’. reprinted in *Logic, Semantics, Metamathematics*, 152-278. Hackett Publishing Company, Indianapolis, IN. 1983.

[VMS88]
VMS Debugger Manual. AA-LA59A-TE, VMS Version 5.0. Digital Equipment Company, Maynard, MA. April 1988.

[Wile84]
R. Wilensky. *LISPcraft*. W. W. Norton & Company, 1984.