Six RePTIL Verbs and the Macintosh

Israel Urieli
Ohio University
Athens OHIO 45701

Abstract:
The Apple Macintosh computer introduced some revolutionary concepts into the personal computer world, one of them being that pointing with a mouse is more intuitive than typing on a keyboard. Significant applications have been developed in which the user need not type on the keyboard at all. Applications are typically developed in terms of an 'event loop', in which the various basic events (such as pressing the mouse button) are continuously polled and resolved.

The token threaded infrastructure of RePTIL is uniquely suited to this environment. The operating system is a self contained kernel and is comprised of the Outer and Inner interpreters (a total of six verbs, 78 bytes of 68000 code) and the Return stack. Thus the input of a keyboard character, and hence the development of the entire RePTIL language can be considered as an application shell on top of this fundamental kernel.

This paper describes the implementation of the RePTIL operating system kernel on the Macintosh computer. An application example of its use is presented in the context of a typical mouse driven application - Conway's Game of Life.

Forth-like systems have a number of features in common. The most fundamental of them is the concept of the inner interpreter - the virtual 'heart' of the computer, and the outer interpreter with its related return stack tying the whole system together. Unfortunately these two elements are usually embedded within the entirety of the language and it is difficult to isolate them as an operating system entity which can be used in its own right. Since they are hidden in the language they loose their identity and the emphasis shifts to reading in, parsing and resolving a line of keyboard characters. The keyboard is the traditional input device for interactive computing and both machine and language have been adapted around that concept. We have bred a community which seems to have genetically accepted 'Alternate-Shift-Delete' as a natural standard.

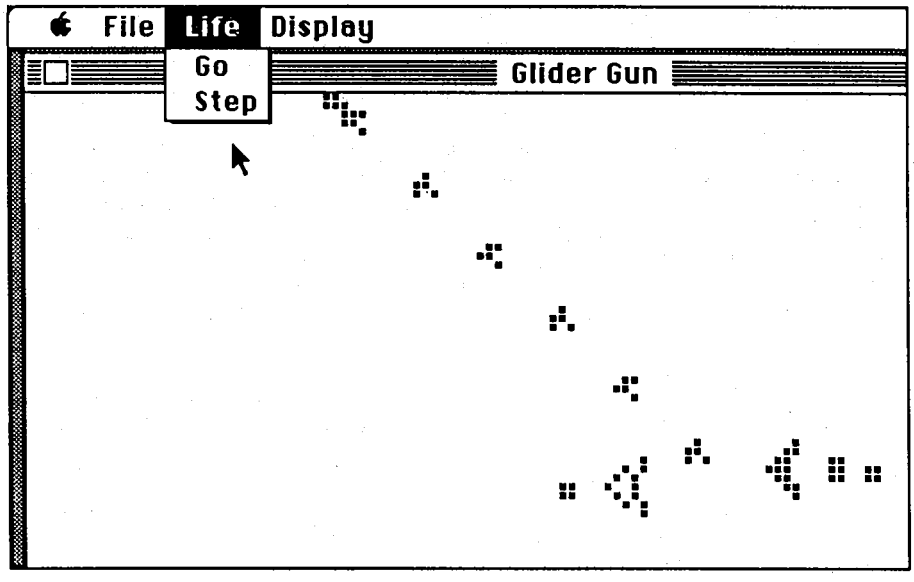
The Apple Macintosh introduced the mouse, shocking us into the realization that there are viable alternatives for interacting with the machine. Languages which could effectively develop applications on the Macintosh were slow in coming, and significantly MacFORTH was the first high level development language which could be adapted to this unique concept. The basic approach to application development is the 'event loop'. Typical events are the null event, depressing the mouse button, releasing the mouse button, depressing a keyboard key, releasing a keyboard key, inserting a disk, interaction with external devices (I/O driver), and so on. Programming an application is typically done by programming a mini operating system within the language, in which there is an infinite 'event loop' to successively poll and resolve the various events.

The token threaded infrastructure of RePTIL (1) is uniquely suited to this environment in which the event loop is mapped onto the outer interpreter of the system. The outer interpreter is at the highest level of the system and is always coded as a threaded secondary. Consider the outer interpreter of RePTIL together with the symbolic tokens of the compiled verb as follows:

```
'Run Do:                          Run: (Status Word)
   Loop[                             Loop[%
      DoProgram                         DoProgram
      DoEvent                           DoEvent
   ]EndLoop                          ]EndLoop%
                                        -8 \ branch offset
   :End                           Return
```

Thus we see that the outer interpreter is an infinite loop in which the two verbs DoProgram

and DoEvent are invoked successively. The verb DoProgram refers to the execution of one complete step of the internal background program, and the verb DoEvent refers to getting and resolving a single external event. To make the discussion more meaningful we have chosen a typical mouse driven application, being Conway's Game of Life (2). Effectively the game involves setting up colonies of cells in a rectangular Universe, and watching them evolve from generation to generation in accordance with defined genetic rules. A snapshot of a typical Glider Gun life form is shown in the following figure:



Thus in the context of this example, the verb DoProgram refers to executing the next generation of the universe and updating the display. The verb DoEvent refers to the user interaction in setting up, modifying or saving life forms, as well as single stepping to the next generation, or simply allowing life to evolve. All of the user interaction is done exclusively with the mouse, either through the menu bar or directly on the universe.

Ones initial reaction may be to reject this format of the outer interpreter as being too specific. User intensive applications such as word processing or spreadsheet applications, for example, do not require a background program. That is the beauty of an extensible language -- simply redefine the verb Run, and presto -- a new operating system made to order. What is being presented here is a methodology rather than a fixed immutable system -- a fundamental token threaded kernel which can be used for developing applications, one of which could be the entire RePTIL interpreter.

The RePTIL kernel consists of the outer interpreter as shown above, the associated return stack, and the inner interpreter. The inner interpreter Next is at the lowest level of the system and is always coded as a headerless primitive. As in many Forth implementations the verb Next is appended to the verb Return, which is the last verb compiled into every threaded secondary. Since Next is coded as a primitive, its implementation will always be host dependent. For a high level functional description of Next, refer Reference 1. The Macintosh implementation is governed by the rather elaborate memory management scheme which requires all references to be program-counter relative.

The verbs Loop[% and]EndLoop% are the runtime verbs compiled respectively by Loop[and]EndLoop (1). Loop[% is a noop location to branch to, and in our implementation is coded as a threaded secondary, just for fun. Every verb begins with a 16 bit status word which summarizes the various attributes of the verb (3). In our basic system the only information contained in the status word is whether the verb is a primitive (status word negative) or a

```
XDEF    Next            ;Make Next available to DoProgram and DoEvent
XDEF    RePTIL          ;So that system initialize routine can invoke it
XREF    DoProgram       ;The background program
XREF    DoEvent         ;Get and resolve the next event
;------RePTIL symbolic token values ------------------------------
T_Return        EQU   0       ; Return from a threaded call
T_Run           EQU   2       ; Run - the Outer Interpreter loop
T_DoEvent       EQU   4       ; DoEvent - the user interface
T_DoProgram     EQU   6       ; DoProgram - the background program
T_Loop_r        EQU   8       ; Loop[% - the runtime Loop[
T_EndLoop_r     EQU   10      ; ]EndLoop% - the runtime ]EndLoop
CodeTable ; all code is relative to CodeTable-----------------------
        DC      Return-CodeTable
        DC      Run-CodeTable
        DC      DoEvent-CodeTable
        DC      DoProgram-CodeTable
        DC      Loop_r-CodeTable
        DC      EndLoop_r-CodeTable
;-------------------------------------------------------------------
ReturnStack     DS.L    $100  ;reserve space in Globals area
RP0             DS.L    1     ;the start of the Return stack
RunToken        DC      T_Run ;the first RePTIL program (to invoke Run)
RePTIL ; Setup and Go-----------------------------------------------
        LEA     RP0(A5),A4    ; Initialize R-stack pointer (RP = A4)
        LEA     RunToken,A3   ; Run Token address to IP (IP = A3)
        BRA     Next          ; to RePTIL Inner interpreter
Return ; from a threaded call--------------------------------------
        DC      $8000
        MOVE.L  (A4)+,A3      ; Pop R-stack to IP
Next ;----the Inner Interpreter------------------------------------
        MOVE.W  (A3)+,D0              ; Fetch token from IP, Bump IP
        MOVE.W  CodeTable(D0),D0      ; Code-CodeTable to D0
        LEA     CodeTable(D0),A0      ; Code Pointer (CP = A0)
        TST.W   (A0)+                 ; Status word sign check, bump CP
        BPL     Threaded
Primitive
        JMP     (A0)                  ; Execute primitive code
Threaded
        MOVE.L  A3,-(A4)              ; Save IP (return addr) on R-stack
        LEA     (A0),A3               ; Next token addr to IP
        BRA     Next                  ; and back again
Run ; the Outer Interpreter----------------------------------------
        DC      $0000         ; 'Run Do:
        DC      T_Loop_r      ;     Loop[
        DC      T_DoProgram   ;        DoProgram
        DC      T_DoEvent     ;        DoEvent
        DC      T_EndLoop_r   ;     ]EndLoop
        DC      -8
        DC      T_Return      ;   :End
Loop_r ; Loop[% (a nop to branch to)-------------------------------
        DC      $0000         ; 'Loop[% Do:
        DC      T_Return      ;   :End
EndLoop_r ; ]EndLoop% (unconditional branch to Loop[%)-------------
        DC      $8000
        ADDA    (A3),A3       ; increment IP by following inline value
        BRA     Next
```

The RePTIL operating system kernel

threaded secondary.]EndLoop% is a primitive which executes an unconditional branch by the value in the following inline word, which is the relative offset to Loop[%.

The verbs DoProgram and DoEvent are coded externally as primitives. They are regular assembly language routines with the exception that they are headed with a status word and return with a 'BRA Next' rather than the regular 'RTS'. DoProgram does a single Life generation on the entire 124 by 75 rectangular universe. It accesses two global flags from the verb DoEvent, being the StepFlag, denoting whether or not we are in the single step mode, and the GoFlag, denoting whether or not we wish Life to 'Go Forth and Evolute'.

DoEvent is the largest verb of the system. It is embedded in the file which includes the system initialization routines including the initialization of all the relevant Macintosh tools and resources. Currently the only events that are processed are the null event, and the pressing of the mouse button.

The complete RePTIL outer and inner interpreter is shown in 68000 assembly language in the figure below, and assembles into 78 bytes of machine code. Notice the unique way of setting up the codetable (sequence of code addresses), and the program-counter relative method of resolving them in the inner interpreter. This approach is typical of Macintosh assembly language application programming. The address register allocation used is as follows:

A7 → SP (Parameter stack pointer)
A6 → reserved for future use as the stack frame for local variables
A5 → GP (Global variables pointer)
A4 → RP (Return stack pointer)
A3 → IP (Interpretive pointer)
A0 → CP (Code pointer)

Conclusions

The question that comes to mind is: "OK, the Game of Life is kinda cute, but what has been presented here that is different from the many other application development approaches?" Whereas most approaches involve creating a mini operating system in the form of an infinite 'event loop', it is usually done in terms of the host language, whether it be C, Pascal, or whatever. The proposal here is that the mini operating system *is* the operating system of the host language -- that language, computer and application be merged into one. What better project to demonstrate this principle on than developing the RePTIL language itself? At this stage we intend to first build the RePTIL interpreter into the system by expanding the verb DoEvent to include keyboard entry as well as convert it to a threaded secondary, adding verbs as needed. This will enable gradually building the language within the framework of a working outer and inner interpreter. It is hoped that this new approach will enable the painless development of the entire language.

References

1. I Urieli, 'REPTIL - Bridging the Gap between Education and Application', Jnl of Forth Application and Research, Vol 4, No 3, 1987.
2. M Gardner, 'Wheels, Life and Other Mathematical Amusements', W H Freeman & Co, 1983.
3. R Buege, 'Status Threaded Code', Proceedings of the 1984 Rochester Forth Conference, pp. 103-104.