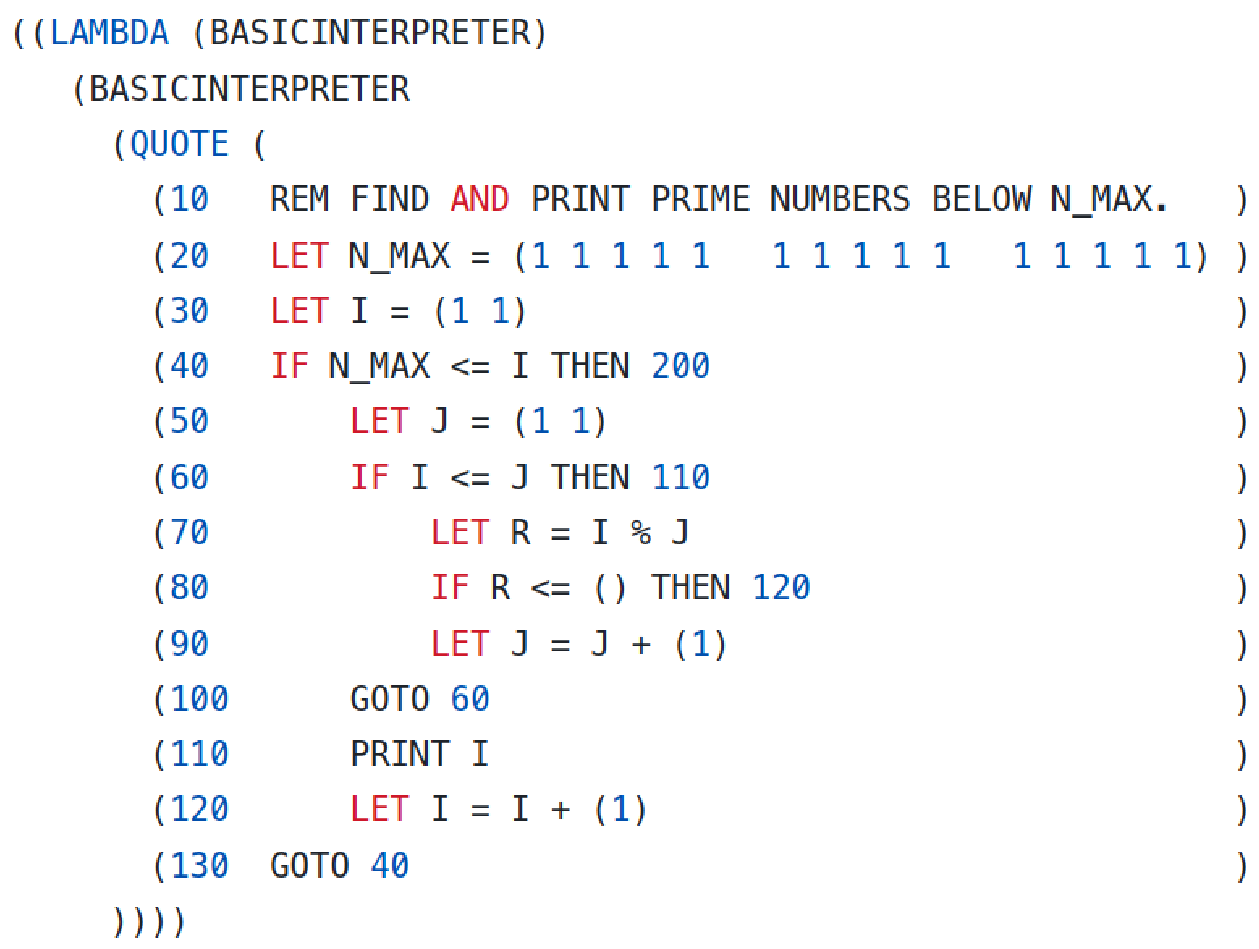
Woodrush's Blog Home

Extending SectorLISP to Implement BASIC REPLs and Games

Jan 12, 2022



```
((LAMBDA (BASICINTERPRETER)
   (BASICINTERPRETER
     (QUOTE (
       (10   REM FIND AND PRINT PRIME NUMBERS BELOW N_MAX.   )
       (20   LET N_MAX = (1 1 1 1 1   1 1 1 1 1   1 1 1 1 1) )
       (30   LET I = (1 1)                                   )
       (40   IF N_MAX <= I THEN 200                          )
       (50       LET J = (1 1)                               )
       (60       IF I <= J THEN 110                          )
       (70           LET R = I % J                           )
       (80           IF R <= () THEN 120                     )
       (90           LET J = J + (1)                         )
       (100      GOTO 60                                     )
       (110      PRINT I                                     )
       (120      LET I = I + (1)                             )
       (130  GOTO 40                                         )
     ))))
```

SectorLISP is an amazing project where a fully functional Lisp interpreter is fit into the 512 bytes of the boot sector of a floppy disk. Since it works as a boot sector program, the binary can be written to a disk to be used as a boot drive, where the computer presents an interface for writing and evaluating Lisp programs, all running in the booting phase of bare metal on the 436-byte program. As it hosts the Turing-Complete language of Lisp, I was in fact able to write a BASIC interpreter in 120 lines of SectorLISP code, which evaluates BASIC programs embedded as an expression within the Lisp code, shown in the screenshot above.

When I first saw SectorLISP and got it to actually run on my machine, I was struck with awe by how such a minimal amount of machine code could be used to open up the vast ability to host an entire programming language. You can write clearly readable programs which the interpreter will accurately evaluate to the correct result. I find it beautiful how such a small program is capable of interpreting a form of human thought and generating a sensible response that contains the meaning encapsulated in the inquired statement.

The Issue - Designing Interactions

After writing various programs for SectorLISP, there was a particular thought that came into my mind. Even after writing the BASIC interpreter, I felt that there was one very important feature that could significantly enhance the capabilities of SectorLISP - that is, the ability to *accept feedback* from the user depending on the program's output, by *designing the interaction* between the user and the computer.

The prime example of this is games. Games are possible to be played on a computer since the player can react *depending on* the output of the computer. Of course, even with pure functions as in SectorLISP, it's still possible to create a game if we make the user of the program run the same program again every time the program demands a new input. The entire history of user inputs can be expressed as a certain list in the program, and the input and output states can be passed through the course of the entire program, and the program can stop whenever a required input is not apparent, also showing its accumulated outputs. However, such an interface that requires repeated inputs is rather inconvenient for the user, inconvenient in the same sense that `IF` is inconvenient than `COND`, and how lambdas that can take only one argument are inconvenient than lambdas that can take any number of arguments, both being used to make the experience of the humans interacting with SectorLISP as simple and natural as possible.

When you think about it, the reason why computers are such a powerful device used almost everywhere in our lives today, is because they can be *redesigned* into an entirely different tool for an arbitrary purpose. The computer is then no longer a tool that is used only by the programmer, but can be used by *anybody* to run its applications. The transition from ENIAC to the dawn of the personal computing era was possible since computers became capable of general tasks other than computing equations, such as writing and saving documents for a business. Today, computers are being used for creating artwork, for playing games, for communicating with others, to only give a few examples. The entire history of computers is shaped by what new tasks computers became capable of, which is inseparable from the means of interaction between the human and the computer.

At the heart of the diverse applications for computers is the language used to program them. This is why programming languages capable of designing interactions are special - once a computer is programmed, it can leave the hands of the programmer and lie in the hands of the user, who interacts with it in a newly designed way.

As a matter of fact, all of the other languages mentioned in the SectorLISP blog post support an I/O functionality. SectorFORTH has the `key` and `emit` instructions which reads a keystroke from the user and prints a character to the console. BootBasic has the instructions `input` and `print` where `input` stores a user input to a variable. Even BF has the instructions `,` and `.` capable of designing arbitrary user text input and output. @rdebath has in fact made a text adventure game written entirely in BF.

Although the goal of SectorLISP is set in the realm of pure functions, I thought that it would be a massive gain if it were able to handle I/O and still have a smaller program size than the other languages mentioned in the SectorLISP blog post. In the context of comparing the binary footprint of programs, it would be a better comparison if all of the programs under discussion had even more functionalities in common. All of this could be achieved if we could construct a version of SectorLISP that is capable of handling user input and outputs that still has a small program size.

The Solution

What could we do to empower SectorLISP with the puzzle piece of interaction? What is a natural way of implementing I/O? To answer this, I created a fork of SectorLISP that supports two new special forms, `READ` and `PRINT`. These two special forms are the counterparts for the `,` and `.` instructions in BF. `READ` accepts an arbitrary S-Expression from the user, and `PRINT` prints the value of the evaluated argument to the console. `PRINT` also prints a newline when called with no arguments as `(PRINT)`.

The fork is available here: https://github.com/woodrush/sectorlisp/tree/io

Update (2022/4/6): The fork was merged into the original SectorLISP repository. Thanks for reviewing and merging it!

Adding all of these features only amounted to an extra 35 bytes of the binary, with a total of 469 bytes, or 471 bytes including the boot signature. This is still 22 bytes or more smaller than the two former champions of minimal languages that fit in a boot sector mentioned in the SectorLISP blog post, which are SectorFORTH (491 bytes) and BootBasic (510 bytes). The rather minimal increase was achievable since most of the code for handling input and output were already available from the REPL's functionality. This fork successfully shows that adding an I/O feature to SectorLISP will still allow it to have a smaller binary footprint than the two former champions.

Update: Thanks to a pull request by @jart, the author of the original SectorLISP, we're down to 465 bytes or 467 bytes including the boot signature. Thank you @jart for your contribution! The details of the assembly optimizations including the one used in this pull request are discussed in the Assembly Optimizations section.

Usage

To run the SectorLISP fork, first `git clone` and `make` SectorLISP's binary, `sectorlisp.bin`:

```
git clone https://github.com/woodrush/sectorlisp
cd sectorlisp
git checkout io
make
```

Update (2022/4/6): Since the fork was merged into the original SectorLISP repository, you could now checkout https://github.com/jart/sectorlisp instead for using these features.

This will generate `sectorlisp.bin` under `./sectorlisp`.

To run SectorLISP on the i8086 emulator Blinkenlights, first follow the instructions on its download page and get the latest version:

```
curl https://justine.lol/blinkenlights/blinkenlights-latest.com >blinkenlights.com
chmod +x blinkenlights.com
```

You can then run SectorLISP by running:

```
./blinkenlights.com -rt sectorlisp.bin
```

In some cases in Ubuntu, there might be a graphics-related error showing and the emulator may not start. In that case, run the following command first available on the download page:

```
sudo sh -c "echo ':APE:M::MZqFpD::/bin/sh:' >/proc/sys/fs/binfmt_misc/register"
```

Running this command should allow you to run Blinkenlights on your terminal. Instructions for running Blinkenlights on other operating systems is described in the Blinkenlights download page.

After starting Blinkenlights, expand the size of your terminal large enough so that the `TELETYPEWRITER` region shows up at the center of the screen. This region is the console used for input and output. Then, press `c` to run the emulator in continuous mode. The cursor in the `TELETYPEWRITER` region should move one line down. You can then start typing in text or paste a long code from your terminal into Blinkenlight's console to run your Lisp program.

Running on Physical Hardware

You can also run SectorLISP on an actual physical machine if you have a PC with an Intel CPU that boots with a BIOS, and a drive such as a USB drive or a floppy disk that can be used as a boot drive. First, mount your drive to the PC you've built sectorlisp.bin on, and check:

```
lsblk -o KNAME,TYPE,SIZE,MODEL
```

Among the list of the hardware, check for the device name for your drive you want to write SectorLISP onto. After making sure of the device name, run the following command, replacing `[devicename]` with your device name. `[devicename]` should be values such as `sda` or `sdb`, depending on your setup.

Caution: The following command used for writing to the drive will overwrite anything that exists in the target drive's boot sector, so it's important to make sure which drive you're writing into. If the command or the device name is wrong, it may overwrite the entire content of your drive or other drives mounted in your PC, probably causing your computer to be unbootable. Please perform these steps with extra care, and at your own risk.

```
sudo dd if=sectorlisp.bin of=/dev/[devicename] bs=512 count=1
```

After you have written your boot drive, insert the drive to the PC you want to boot it from. You may have to change the boot priority settings from the BIOS to make sure the PC boots from the target drive. When the drive boots successfully, you should see a cursor blinking in a blank screen, which indicates that you're ready to type your Lisp code into bare metal.

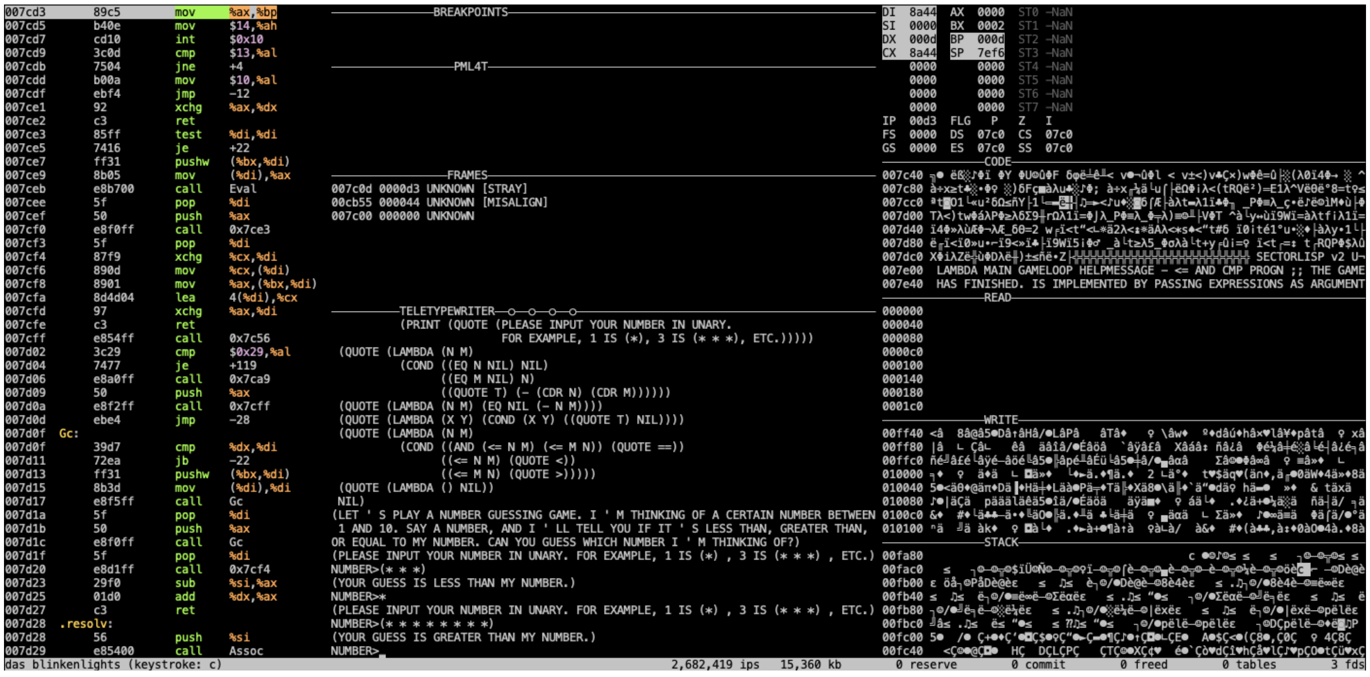
Applications

Here we present examples to showcase the capabilities of `READ` and `PRINT`.

Games

A major example of interactive programs is games. I created a simple number guessing game that works on the fork of SectorLISP.

Here is a screenshot of the game in action, run in Blinkenlights:



```
007cd3      89c5      mov     %ax,%bp
007cd5      b40e      mov     $14,%ah
007cd7      cd10      int     $0x10
007cd9      3c0d      cmp     $13,%al
007cdb      7504      jne     +4
007cdd      b00a      mov     $10,%al
007cdf      ebf4      jmp     -12
007ce1      92        xchg    %ax,%dx
007ce2      c3        ret
007ce3      85ff      test    %di,%di
007ce5      7416      je      +22
007ce7      ff31      pushw   (%bx,%di)
007ce9      8b05      mov     (%di),%ax
007ceb      e8b700    call    Eval
007cee      5f        pop     %di
007cef      50        push    %ax
007cf0      e8f0ff    call    0x7ce3
007cf3      5f        pop     %di
007cf4      87f9      xchg    %cx,%di
007cf6      890d      mov     %cx,(%di)
007cf8      8901      mov     %ax,(%bx,%di)
007cfa      8d4d04    lea     4(%di),%cx
007cfd      97        xchg    %ax,%di
007cfe      c3        ret
007cff      e854ff    call    0x7c56
007d02      3c29      cmp     $0x29,%al
007d04      7477      je      +119
007d06      e8a0ff    call    0x7ca9
007d09      50        push    %ax
007d0a      e8f2ff    call    0x7cff
007d0d      ebe4      jmp     -28
007d0f   Gc:
007d0f      39d7      cmp     %dx,%di
007d11      72ea      jb      -22
007d13      ff31      pushw   (%bx,%di)
007d15      8b3d      mov     (%di),%di
007d17      e8f5ff    call    Gc
007d1a      5f        pop     %di
007d1b      50        push    %ax
007d1c      e8f0ff    call    Gc
007d1f      5f        pop     %di
007d20      e8d1ff    call    0x7cf4
007d23      29f0      sub     %si,%ax
007d25      01d0      add     %dx,%ax
007d27      c3        ret
007d28   .resolv:
007d28      56        push    %si
007d29      e85400    call    Assoc
das blinkenlights (keystroke: c)
```

```
────────────BREAKPOINTS────────────

────────────PML4T────────────

────────────FRAMES────────────
007c0d 0000d3 UNKNOWN [STRAY]
00cb55 000044 UNKNOWN [MISALIGN]
007c00 000000 UNKNOWN

────────TELETYPEWRITER─○─○─○─○────────
          (PRINT (QUOTE (PLEASE INPUT YOUR NUMBER IN UNARY.
                         FOR EXAMPLE, 1 IS (*), 3 IS (* * *), ETC.)))))
 (QUOTE (LAMBDA (N M)
          (COND ((EQ N NIL) NIL)
                ((EQ M NIL) N)
                ((QUOTE T) (- (CDR N) (CDR M))))))
 (QUOTE (LAMBDA (N M) (EQ NIL (- N M))))
 (QUOTE (LAMBDA (X Y) (COND (X Y) ((QUOTE T) NIL))))
 (QUOTE (LAMBDA (N M)
          (COND ((AND (<= N M) (<= M N)) (QUOTE ==))
                ((<= N M) (QUOTE <))
                ((<= M N) (QUOTE >)))))
 (QUOTE (LAMBDA () NIL))
 NIL)
(LET ' S PLAY A NUMBER GUESSING GAME. I ' M THINKING OF A CERTAIN NUMBER BETWEEN
 1 AND 10. SAY A NUMBER, AND I ' LL TELL YOU IF IT ' S LESS THAN, GREATER THAN,
OR EQUAL TO MY NUMBER. CAN YOU GUESS WHICH NUMBER I ' M THINKING OF?)
(PLEASE INPUT YOUR NUMBER IN UNARY. FOR EXAMPLE, 1 IS (*) , 3 IS (* * *) , ETC.)
NUMBER>(* * *)
(YOUR GUESS IS LESS THAN MY NUMBER.)
NUMBER>*
(PLEASE INPUT YOUR NUMBER IN UNARY. FOR EXAMPLE, 1 IS (*) , 3 IS (* * *) , ETC.)
NUMBER>(* * * * * * * *)
(YOUR GUESS IS GREATER THAN MY NUMBER.)
NUMBER>_
                                        2,682,419 ips   15,360 kb
```

```
DI  8a44  AX  0000  ST0 -NaN
SI  0000  BX  0002  ST1 -NaN
DX  000d  BP  000d  ST2 -NaN
CX  8a44  SP  7ef6  ST3 -NaN
    0000      0000  ST4 -NaN
    0000      0000  ST5 -NaN
    0000      0000  ST6 -NaN
    0000      0000  ST7 -NaN
IP  00d3  FLG  P    Z   I
FS  0000  DS  07c0  CS  07c0
GS  0000  ES  07c0  SS  07c0
────────────CODE────────────
007c40 [illegible]
007c80 [illegible]
007cc0 [illegible]
007d00 [illegible]
007d40 [illegible]
007d80 [illegible]
007dc0 [illegible] SECTORLISP v2 U
007e00  LAMBDA MAIN GAMELOOP HELPMESSAGE - <= AND CMP PROGN ;; THE GAME
007e40  HAS FINISHED. IS IMPLEMENTED BY PASSING EXPRESSIONS AS ARGUMENT
────────────READ────────────
000000
000040
000080
0000c0
000100
000140
000180
0001c0
────────────WRITE────────────
00ff40 [illegible]
00ff80 [illegible]
00ffc0 [illegible]
010000 [illegible]
010040 [illegible]
010080 [illegible]
0100c0 [illegible]
010100 [illegible]
────────────STACK────────────
00fa80 [illegible]
00fac0 [illegible]
00fb00 [illegible]
00fb40 [illegible]
00fb80 [illegible]
00fbc0 [illegible]
00fc00 [illegible]
00fc40 [illegible]
 0 reserve    0 commit    0 freed    0 tables    3 fds
```

Here is the text shown in the console:

```
(LET ' S PLAY A NUMBER GUESSING GAME. I ' M THINKING OF A CERTAIN NUMBER BETWEEN
 1 AND 10. SAY A NUMBER, AND I ' LL TELL YOU IF IT ' S LESS THAN, GREATER THAN,
OR EQUAL TO MY NUMBER. CAN YOU GUESS WHICH NUMBER I ' M THINKING OF?)
(PLEASE INPUT YOUR NUMBER IN UNARY. FOR EXAMPLE, 1 IS (*) , 3 IS (* * *) , ETC.)
NUMBER>(* * *)
(YOUR GUESS IS LESS THAN MY NUMBER.)
NUMBER>*
(PLEASE INPUT YOUR NUMBER IN UNARY. FOR EXAMPLE, 1 IS (*) , 3 IS (* * *) , ETC.)
NUMBER>(* * * * * * * *)
(YOUR GUESS IS GREATER THAN MY NUMBER.)
NUMBER>
```

We can see that the game is able to produce interactive outputs based on the feedback from the user, which is an essential feature for creating games. Note that there is also robust input handling in action, where in the second input `NUMBER>*`, the user writes an invalid input `*`, which is not a list. The game can handle such inputs without crashing.

The code is available at https://github.com/woodrush/sectorlisp-examples/blob/main/lisp/number-guessing-game.lisp.

Extended Lisp REPL - Transforming the Language Itself

The I/O feature can be used to transform the SectorLISP language itself as well. As an example, I made an extended Lisp REPL where `macro`, `define`, `progn`, as well as `print` and `read` are all implemented as new special forms.

Here is an example session of the program:

```
REPL>(define defmacro (quote (macro (name vars body)
       (` (define (~ name) (quote (macro (~ vars) (~ body))))))))
=>(macro (name vars body) (` (define (~ name) (quote (macro (~ vars) (~ body))))
))

REPL>(defmacro repquote (x)
       (` (quote ((~ x) (~ x)))))
=>(macro (x) (` (quote ((~ x) (~ x)))))

REPL>(repquote (1 2 3))
=>((1 2 3) (1 2 3))

REPL>
```

The code is available at https://github.com/woodrush/sectorlisp-examples/blob/main/lisp/repl-macro-define.lisp.

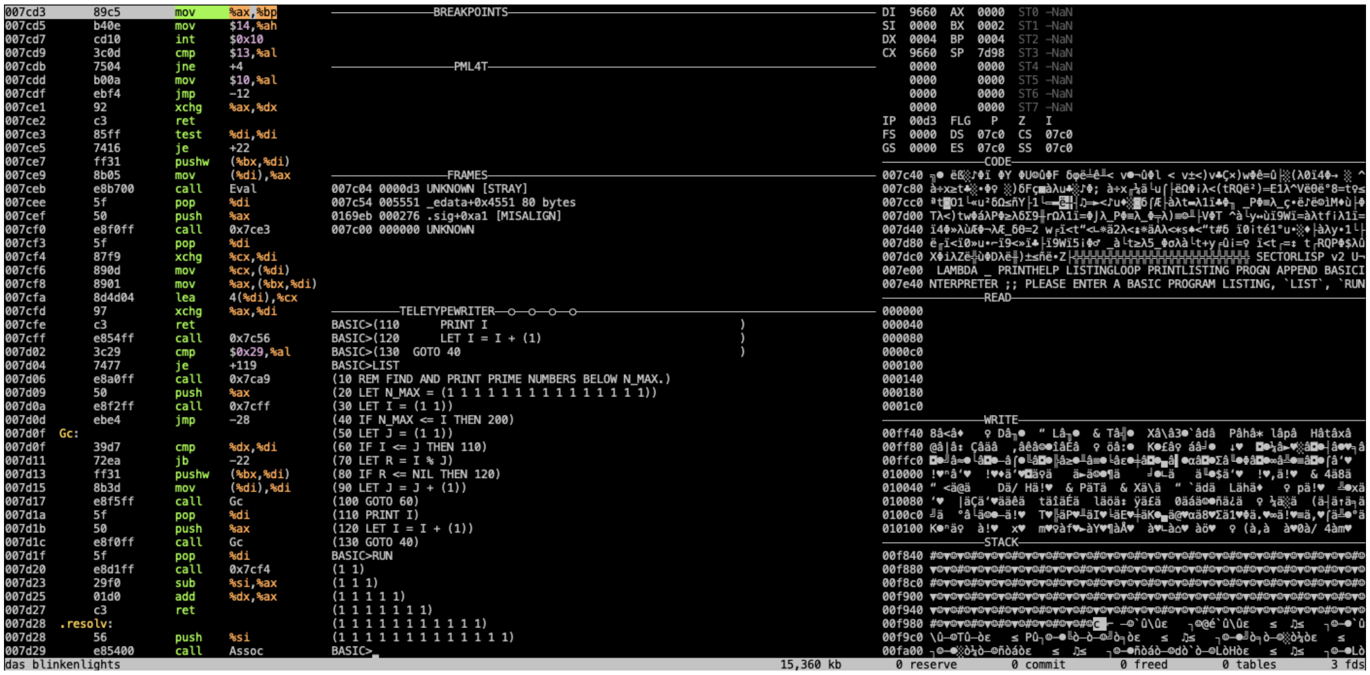
In the example above, the user first uses the backquote macro `` ` `` to define `defmacro` as a new macro, then uses `defmacro` to define a new macro `repquote`. These newly added features allow an interaction that is much more closer to those in modern Lisp dialects.

In the code, these additional user inputs are included at the end of the code which could be directly pasted in the console. However, we could look at this in another way - by writing the REPL code as the header, we have effectively transformed the *syntax of the language itself*, by introducing new special forms which were not present in the original interface. The `DEFINE` special form is also introduced in SectorLISP's friendly branch, which adds some extra bytes. With `READ` and `PRINT`, we can instead build these new features on top of the interface as software, allowing us to save a lot of the program size.

Interactive BASIC REPL

As a final example for drastically modifying the means of user interactions, I made an interactive BASIC interpreter written in the I/O SectorLISP. It runs a subset of BASIC with the instructions `LET`, `IF`, `GOTO`, `PRINT`, `REM`, and the infix operators `+`, `-`, `%`, and `<=`. Integers are expressed in unary as a list of atoms, such as `(1 1 1)`.

Here is a screenshot of the final results, run in Blinkenlights:



```
007cd3       89c5      mov     %ax,%bp
007cd5       b40e      mov     $14,%ah
007cd7       cd10      int     $0x10
007cd9       3c0d      cmp     $13,%al
007cdb       7504      jne     +4
007cdd       b00a      mov     $10,%al
007cdf       ebf4      jmp     -12
007ce1       92        xchg    %ax,%dx
007ce2       c3        ret
007ce3       85ff      test    %di,%di
007ce5       7416      je      +22
007ce7       ff31      pushw   (%bx,%di)
007ce9       8b05      mov     (%di),%ax
007ceb       e8b700    call    Eval
007cee       5f        pop     %di
007cef       50        push    %ax
007cf0       e8f0ff    call    0x7ce3
007cf3       5f        pop     %di
007cf4       87f9      xchg    %cx,%di
007cf6       890d      mov     %cx,(%di)
007cf8       8901      mov     %ax,(%bx,%di)
007cfa       8d4d04    lea     4(%di),%cx
007cfd       97        xchg    %ax,%di
007cfe       c3        ret
007cff       e854ff    call    0x7c56
007d02       3c29      cmp     $0x29,%al
007d04       7477      je      +119
007d06       e8a0ff    call    0x7ca9
007d09       50        push    %ax
007d0a       e8f2ff    call    0x7cff
007d0d       ebe4      jmp     -28
007d0f  Gc:
007d0f       39d7      cmp     %dx,%di
007d11       72ea      jb      -22
007d13       ff31      pushw   (%bx,%di)
007d15       8b3d      mov     (%di),%di
007d17       e8f5ff    call    Gc
007d1a       5f        pop     %di
007d1b       50        push    %ax
007d1c       e8f0ff    call    Gc
007d1f       5f        pop     %di
007d20       e8d1ff    call    0x7cf4
007d23       29f0      sub     %si,%ax
007d25       01d0      add     %dx,%ax
007d27       c3        ret
007d28  .resolv:
007d28       56        push    %si
007d29       e85400    call    Assoc
das blinkenlights
```

```
─────────────BREAKPOINTS──────────────────────────────

─────────────PML4T────────────────────────────────────

─────────────FRAMES───────────────────────────────────
007c04 0000d3 UNKNOWN [STRAY]
007c54 005551 _edata+0x4551 80 bytes
0169eb 000276 .sig+0xa1 [MISALIGN]
007c00 000000 UNKNOWN

─────────TELETYPEWRITER─○─○─○─○──────────────────────
BASIC>(110      PRINT I                              )
BASIC>(120      LET I = I + (1)                      )
BASIC>(130  GOTO 40                                  )
BASIC>LIST
(10 REM FIND AND PRINT PRIME NUMBERS BELOW N_MAX.)
(20 LET N_MAX = (1 1 1 1 1 1 1 1 1 1 1 1 1 1 1))
(30 LET I = (1 1))
(40 IF N_MAX <= I THEN 200)
(50 LET J = (1 1))
(60 IF I <= J THEN 110)
(70 LET R = I % J)
(80 IF R <= NIL THEN 120)
(90 LET J = J + (1))
(100 GOTO 60)
(110 PRINT I)
(120 LET I = I + (1))
(130 GOTO 40)
BASIC>RUN
(1 1)
(1 1 1)
(1 1 1 1 1)
(1 1 1 1 1 1 1)
(1 1 1 1 1 1 1 1 1 1 1)
(1 1 1 1 1 1 1 1 1 1 1 1 1)
BASIC>_
                                           15,360 kb
```

```
DI  9660  AX  0000  ST0 -NaN
SI  0000  BX  0002  ST1 -NaN
DX  0004  BP  0004  ST2 -NaN
CX  9660  SP  7d98  ST3 -NaN
    0000      0000  ST4 -NaN
    0000      0000  ST5 -NaN
    0000      0000  ST6 -NaN
    0000      0000  ST7 -NaN
IP  00d3  FLG   P   Z   I
FS  0000  DS  07c0  CS  07c0
GS  0000  ES  07c0  SS  07c0
───────────CODE──────────────────────
007c40 [illegible]
007c80 [illegible]
007cc0 [illegible]
007d00 [illegible]
007d40 [illegible]
007d80 [illegible]
007dc0 [illegible] SECTORLISP v2 U¬
007e00  LAMBDA _ PRINTHELP LISTINGLOOP PRINTLISTING PROGN APPEND BASICI
007e40 NTERPRETER ;; PLEASE ENTER A BASIC PROGRAM LISTING, `LIST`, `RUN
───────────READ──────────────────────
000000
000040
000080
0000c0
000100
000140
000180
0001c0
───────────WRITE─────────────────────
00ff40 [illegible]
00ff80 [illegible]
00ffc0 [illegible]
010000 [illegible]
010040 [illegible]
010080 [illegible]
0100c0 [illegible]
010100 [illegible]
───────────STACK─────────────────────
00f840 [illegible]
00f880 [illegible]
00f8c0 [illegible]
00f900 [illegible]
00f940 [illegible]
00f980 [illegible]
00f9c0 [illegible]
00fa00 [illegible]
  0 reserve     0 commit     0 freed     0 tables     3 fds
```

@jart has created a video of it running on Blinkenlights (Thank you @jart!):

0:00

The code is available at https://github.com/woodrush/sectorlisp-examples/blob/main/lisp/basic-repl.lisp.

In this example, SectorLISP no longer presents an interface for evaluating Lisp expressions, but provides a new interface for recording and evaluating BASIC programs, transforming SectorLISP into an entirely different application. This highlights how programming languages can be used to redesign computers into tools for arbitrary purposes - using this SectorLISP program, users can now interact with the computer in a new way using the BASIC language.

Although it is indeed possible to run this evaluator as a static program as in the code shown at the beginning, the new program is able to hide and encapsulate the details of the underlying Lisp program by presenting a new interface. For the static version, the evaluator must also be entirely retyped again to evaluate a new BASIC program, which is a major difference in terms of interaction. This shows how features as simple as `READ` and `PRINT` can be used to create a powerful application with the language. In a way, we can think that SectorLISP now works as a minimal operating system, and the programs within it such as this REPL works as an application that extends the capabilities of the underlying OS.

Implementation Details

Let’s look at some details for dealing with I/O.

Sequential Execution - Defining `PROGN` using Pre-existing Features

First of all, side effects are inseparable from the notion of sequential execution. Although lambda bodies in SectorLISP can only have one expression, there is in fact an already built-in way to naturally manage sequential execution - you can pass expressions as the *arguments* of lambdas to make them executed sequentially!

For example, the following program allows the execution of three consecutive `PRINT`s:

```
((LAMBDA () NIL)
 (PRINT (QUOTE A))
 (PRINT (QUOTE B))
 (PRINT (QUOTE C)))
```

Here, each `PRINT` statement is taken as the arguments of an empty lambda expression `(LAMBDA () NIL)`, which are all executed in the order of appearance. This is possible since `EVLIS` evaluates all of the arguments *before* calling `PAIRLIS` to bind the values to the variables, so all of the expressions get evaluated in order regardless to the number of arguments that the lambda expects.

Since this empty lambda can be used anywhere with an arbitrary number of expressions, you can name it `PROGN` and use it as follows:

```
((LAMBDA (PROGN)
   (PROGN (PRINT (QUOTE A))
          (PRINT (QUOTE B))
          (PRINT (QUOTE C))))
 (QUOTE (LAMBDA () NIL)))
```

Note that `PROGN` always returns `NIL` instead of the last expression inside the sequence, which is different from the behavior in most Lisp dialects. To extract the values from a PROGN sequence, you can create repeated lambda arguments as follows:

```
((LAMBDA (PROGN)
   (PROGN (PRINT (QUOTE A))
          (PRINT (QUOTE B))
          (PRINT (QUOTE C))
          (QUOTE D)))
  (QUOTE (LAMBDA (X X X X) X)))
  ;; Returns (QUOTE D)
```

Note that the return value of `PRINT` is designed to be undefined to save the program space. This does not become a problem as will be discussed later.

You can use `CONS` instead of `PROGN` as well for the same purpose:

```
(CDR (CDR (CDR
  (CONS (PRINT A)
  (CONS (PRINT B)
  (CONS (PRINT C)
        (QUOTE D)))))))
```

These tools are enough to deal with sequential execution and the extraction of the executed expressions.

When I first came up with the `PROGN` solution, I thought it was as if SectorLISP had been awaiting for sequential execution to be used. Although pure expressions as in the original SectorLISP implementation do not require this feature, it was a nice realization that this feature had already been built in so naturally in SectorLISP. It is also pleasing that the syntax it provides is the same as modern Lisp dialects, only with the difference that it always returns `NIL` instead of the final value, which still can be worked around using the methods discussed earlier.

Comments inside `PROGN`

Since all of the values inside `PROGN` are discarded after its execution, you can write comments inside a `PROGN` block, with the expense of some RAM space in the string interning region and some extra evaluations:

```
((LAMBDA (
          PROGN ;;
         )
   (PROGN ;; (QUOTE - THIS PRINTS 3 CONSECTUTIVE LETTERS.)
          (PRINT (QUOTE A))
          (PRINT (QUOTE B))
          (PRINT (QUOTE C))))
  (QUOTE (LAMBDA () NIL))
  NIL)
```

Here, the variable `;;` is bound to `NIL` and is placed inside `PROGN`. Since `;;` immediately evaluates to `NIL` and is discarded, this does nothing to the relevant states of the interpreter and the program. Because `;;` actually does not comment out its following statement in SectorLISP, the comment body that follows after is enclosed inside a `QUOTE` form to prevent from it being executed, which allows for its result to also be discarded after execution.

Also, note that the parentheses for the outer lambda have some extra newlines to prevent text editors from commenting out the parentheses `)`. This format is used in the number guessing program as well.

Loops using Recursion

Although this is not a newly added feature, it is worth noting that loops can be implemented as recursion, by calling a function within itself. In the number guessing game example, the functions `MAIN` and `GAMELOOP` are called within themselves to be executed an arbitrary number of times. This combined with `PROGN` provides a natural way for writing sequential programs.

Print Debugging

The `PRINT` feature is not only convenient for the user of the program, but in fact provides a helpful interface for the programmer as well. That is, it allows for print debugging, to check the values occurring at runtime. Even with Lisp having a comfortable syntax, even the most experienced programmer would have a difficult time debugging a large program if the internal states and variables could not be observed at runtime.

This can be done by simply wrapping the expression with a predefined `DEBUG` function:

```
((LAMBDA (DEBUG)
   ...
   (DEBUG EXPR)
   ...
 )
 (QUOTE (LAMBDA (X) (CDR (CONS (PRINT X) X)))))
```

The need for the extra wrapper function `DEBUG` occurs since the return value of `PRINT` is designed to be undefined to save the program size.

The art of writing a program always comes with the act of deleting and revising a program, by observing its behavior and the internal states. The print debugging feature is a simple yet powerful interface that is a de facto requirement if one wishes to write large programs. Such an interface is comparable to the reason why `COND` is implemented in SectorLISP instead of `IF` which usually induces a more obfuscated program structure. I myself heavily used this print debugging feature to write the BASIC interpreter, as well as the version that runs in the original SectorLISP which I wrote and debugged in the I/O SectorLISP fork.

Return Values of `PRINT`

As it was mentioned earlier, `PRINT` is designed to return an undefined value to save the program size. Since values passed to `PRINT` can be extracted using `DEBUG`, and `PRINT` can be used in `PROGN` where values are discarded, having `PRINT` to return undefined values was not a problem for at least in all of the examples discussed before. Running a bare `PRINT` expression in the REPL also didn't print any unwanted strings in the console, so I consider that this property can be safely managed in most use cases. Running various `PRINT` expressions in the REPL turns out like this:

```
(PRINT (QUOTE A))
A
(PRINT (PRINT (QUOTE A)))
ANIL
(PRINT (READ))AAA
AAA
```

Notice that the results are slightly odd in the first expression, since the REPL is supposed to show the return value of `PRINT` as well as its effect of printing `A` in the console, but nothing is printed. In the second expression, a nested `PRINT` expression turns out to return `NIL`, which is printed after `A` as a return value by the REPL. This phenomenon should not occur in well-written large programs, if the program is written so that the return values of `PRINT` is not referenced by anything, which should be a natural result if they are all executed inside `PROGN`.

`READ` is much safer since it is by definition designed to have a valid return value regardless of its context. At first, there was a bug where the first character was ignored by `READ`, but it was fixed by caching the lookahead character from the user input inside `GetChar`, as fixed in `162969d` (the latest version uses the `%bp` register instead of `%fs`, as fixed in `1af3db7`).

Assembly Optimizations

Here we'll cover the details of optimizing the assembly size. More details for the methods used in the original SectorLISP assembly code are available at the original SectorLISP blog post, https://justine.lol/sectorlisp2/.

Smaller Jump Instruction Encodings

This is a method used in the pull request by @jart, the author of the original SectorLISP. Conditional jumps in x86 are encoded in different instruction sizes depending on the size of the jump's displacement. When the displacement fits in one byte, i.e. it is between -128 and 127, the instruction fits in two bytes, instead of four bytes when the displacement is larger than that size.

The pull request by @jart uses this feature by first reordering the functions within the assembly code, allowing to shrink the displacements for the conditional jump instructions related to `READ` and `PRINT`.

Reducing Return Instructions using the Control Flow Structure

This is a method used in the original SectorLISP implementation, which is used in the fork as well. Consider the following example where a function `A` calls another function `B` and then immediately returns afterward:

```
A:      mov %ax,%si
        call B
        ret

B:      mov %si,%bp
        ret
```

the code can then be reduced by two instructions without changing the behavior, by concatenating `A` before `B` as the following:

```
A:      mov %ax,%si
;       slide
B:      mov %si,%bp
        ret
```

This way, even if there is no `ret` instruction in the `A` block, the control flow can immediately move inside `B` where it has a `ret` instruction. The same `ret` instruction is therefore shared by two functions `A` and `B`. This allows function calls such as `A` and `B` to both behave the same as in the previous code with a fewer amount of instructions.

This method is used to implement `READ` and `PRINT` as an extension of `.PutObject` and `GetToken` where some additional instructions are run before the original functions. This way of reusing existing code allowed the increase of the program size to be a rather small size.

Conclusion

I made a fork of SectorLISP that supports two new special forms `READ` and `PRINT`, which provides a natural I/O interface useful for both the programmer and the user of the program. This allowed for the following findings:

- The fork allows for writing interactive programs for SectorLISP, such as games and REPLs of other programming languages such as a subset of BASIC.
- With the new special forms `READ` and `PRINT`, you can now design the interactions between the user and the computer, which is a feature supported in all of the other languages mentioned in the original SectorLISP blog post, including SectorFORTH, BootBasic, and also BF.
- Adding all of these features only amounted to an extra 35 bytes of the binary, with a total of 469 bytes, or 471 bytes including the boot signature. When speaking of the binary footprint of a program, it is important for each program to share as many common features as possible. The fork of SectorLISP achieves this by supporting the I/O feature, and also accomplishes in showing that the program size can still be limited to an amount less by 22 bytes or more compared with the other programming languages mentioned in the SectorLISP blog post.
    - **Update:** As mentioned earlier, a pull request from @jart has allowed us to bring down the total program size to 465 bytes or 467 bytes including the boot signature. Thank you @jart for your contribution!
    - **Update (2022/4/6):** The fork was merged into the original SectorLISP repo. Thanks for reviewing and merging it!

Credits

The video for the interactive BASIC REPL was created by Justine Tunney. The new I/O fork of SectorLISP discussed in this post was first created by Hikaru Ikuta, and have received contributions from Justine Tunney. The SectorLISP project was first started by Justine Tunney and was created by the authors who have contributed to the project, and the authors credited in the original SectorLISP blog post. I'd also like to thank Justine and Hannah Miller from the Rochester Institute of Technology for the fruitful discussion on improving this blog post.

Woodrush's Blog

- woodrush
- woodrush924
- Hikaru Ikuta