6.836 Embodied Intelligence, 2003

Research Assignment 3

Issued March 7, Due March 20

This assignment is about a Tom Ray-style evolution system inside a simulated computer. We are giving you a number of ANSI C source files that together implement a Tierra-like system, but this one is called Sierra. It is set up so that you can try different virtual machine architectures, and watch the evolution of competing programs.

The biggest of the files is an implementation of a subset of Common Lisp. You absolutely do not need to look inside this file at all. For this assignment it simply provides a convenient front end so that you can poke around inside the emulated machine, and look at the history of the genomes that have evolved.

The system is set up so that it runs in the background and you can type to it in the foreground looking at various things as the system evolves.

The system requires a seed for a pseudo random number generator. Once this is set any given run is completely deterministic and repeatable. If you use the same parameters and the same seed you will get exactly the same results. The random number generator is used to determine when cosmic rays hit, when there are copying errors, and where in simulated memory a new program is placed.

There are really two things of importance given to you in the code. One is the Sierra system itself, and the second is a particular virtual machine architecure called M1. In the fifth and sixth tasks you are asked to copy and change M1 to be a new sort of virtual machine. This will require you to program in C.

The M1 machine

The M1 machine uses four-bit words. Each M1 instruction is one or two words long. We will write their numerical values in hexadecimal using the standard C notation (e.g., `0xb` for 11_{10}).

The first (and perhaps only) word of an instruction is an opcode, representing one of 16 possible instructions. When an opcode requires a second word, that specifies either one or two registers. Sometimes there are unused bits in instructions–these are always ignored–they provide a good place for harmless mutations to accumulate so that evolution can make new constructs. When a single register is specified it is in the low order two bits of the word.

With each program instance in memory there are associated four registers, an accumulator, a one deep stack for procedure calls, and a program counter. The registers and the accumulator are each 32 bits wide, and are initialized to the following values when a program instance is spawned:

acc	−1
r0	0
r1	1
r2	2
r3	3

These can be used by a program to construct interesting constants, although the ancestor program does not do so.

Each spawned program starts with a certain amount of *energy units*. It is the length of the program in words, times a constant `energy_init`. Currently `energy_init` has value

20. The program that successfully spawns a new program is also rewarded with energy. It is given units numbering the length of the new program times a constant `energy_success`, which is currently 10 in the distributed code.

Executing instructions costs energy. If a program runs down to zero units of energy it is dead, and its memory space is returned to the free pool. There is no other form of reaper or queue as existed in Tom Ray's Tierra system.

The sixteen instructions, with the opcodes in the files handed out, are:

`0x0`	`nop0`	no operation
`0x1`	`nop1`	no operation
`0x2`	`find`	find a label and place the address in *acc*
`0x3`	`sto`	store *acc* into register
`0x4`	`ld`	load *acc* from register
`0x5`	`sub`	subtract register from *acc* and put result in *acc*
`0x6`	`add`	add register to *acc* and leave result in in *acc*
`0x7`	`go`	go to a label
`0x8`	`alloc`	allocate a chunk of memory whose size is specified by a register and put address in *acc*
`0x9`	`copy`	copy word pointed to by one register to word pointed to by another; the copy is subject to one bit mutation errors
`0xa`	`inc`	increment a register by 1 and copy to *acc*
`0xb`	`dec`	decrement a register by 1 and copy to *acc*
`0xc`	`bne`	branch to a label if *acc* is non-zero
`0xd`	`call`	call a procedure at a label
`0xe`	`ret`	return from the last called procedure
`0xf`	`spawn`	set an allocated memory space to go off as its own program, starting from an address in memory specified by a register.

Each instruction executed takes one unit of energy. If an instruction encounters an **error**, then it is not executed, and an additional unit of energy is consumed. The specific conditions under which errors can occur are as follows:

`copy` The *from* register is specified by the high order two bits of the second instruction word. The *to* register is specified by the low order two bits of the second word. This instruction is in **error** unless the *from* register is an integer in addressable memory ($[0, 1048575]$ in the initial version of the code), and the *to* register is an address within the bounds of an allocated but unspawned child.

`alloc` If there is already a child chunk of memory allocated to this program then it is deallocated first. The absolute value of the contents of the register is taken to be the size of memory chunk to be allocated. Sierra demands that it fall inside some range, specified by the constants `mingensize` and `maxgensize`. Currently this range is $[16, 128]$. If the requested amount falls outside of this range then it is an **error**. Sierra picks a random location in memory to try to allocate. If that chunk of memory overlaps some existing program or allocated child then it is an **error**. In any case the allocating program is charged energy units, the size of the chunk of memory it tried to allocate. If there is an error then the instruction is re-executed at the next tick of the clock. This means that a program requesting an illegal size chunk of memory will spin on the `alloc` instruction until it dies. It also means that when memory is very full the program may die trying to allocate a child, causing a die off of many programs, and a freeing up of memory.

`spawn` It is an **error** if there is no allocated child, or if the address in the register is outside the range of that allocated child.

`ret` If there has not been a `call` instruction then this instruction causes an **error** and falls through.

`call` If there is a pending call that has not returned then this instruction is an **error** and falls through.

finding a label A label to be found must be three successive `nop`s, from the set of `nop0` and `nop1`. They must form a complementary pattern to the three words following the instruction that needs a label. The pattern can be specified by **any** word patterns, and just their low order bits are used. Thus a sequence of instructions words like `go`, `nop1`, `ld r2`, will match a pattern `nop0`, `nop1`, `nop1`. The found label, however, must consist exactly of `nop`s. If the branch is not taken these three following words are skipped over. The pattern is searched for in an extending radius from the instruction, looking in a range both forward and backward of $[3, 512]$ words from the current instruction. The upper bound is set by the constant `find_limit`. It is an **error** if the complementary label set can not be found within the limit, and then the branch is not taken. If the label is found forward, then the address of the first instruction following it is returned in the accumulator. If the label is found backward then the address of the first word of the label is returned. So a backward branch will end up executing the label. In turn this makes it easier to compute the length of a program without having to do an add of three.

The ancestral creature is specified by filling a character array with symbolic constants corresponding to the opcodes and register names. It looks like this:

```
char ancestcode[] = {
  nop0,
  nop0,
  nop1,
  find,
...
  spawn, r3,
  ret,
  nop0,
  nop1,
  nop1,
  -1};
```

It is more easy to read is the assembled version. This is what gets printed out by the Sierra system when you ask to print the code of a genome (e.g., with `(print-code 0)`–see below). The comments were added manually later:

```
  0: 0x0  nop0              /* start label */
```

```
 1: 0x0  nop0
 2: 0x1  nop1
 3: 0x2  find              /* find start label */
 4: 0x1  nop1
 5: 0x1  nop1
 6: 0x0  nop0
 7: 0x30 sto r0            /* store it in r0 */
 9: 0x2  find              /* find end label */
10: 0x1  nop1
11: 0x0  nop0
12: 0x0  nop0
13: 0x50 sub r0           /* determine length of program */
15: 0x31 sto r1           /* and save that in r1 */
17: 0xd  call             /* call the copy routine */
18: 0x0  nop0
19: 0x1  nop1
20: 0x0  nop0
21: 0x7  go               /* go back to the start */
22: 0x1  nop1
23: 0x1  nop1
24: 0x0  nop0
25: 0x1  nop1             /* copy routine label */
26: 0x0  nop0
27: 0x1  nop1
28: 0x81 alloc r1         /* allocate a chunk of memory */
30: 0x32 sto r2           /* and save its address in two places */
32: 0x33 sto r3
34: 0x1  nop1             /* label for loop */
35: 0x1  nop1
36: 0x1  nop1
37: 0x92 copy r0 to r2    /* copy a word */
39: 0xa0 inc r0           /* and update the from */
41: 0xa2 inc r2           /* and to pointers */
43: 0xb1 dec r1           /* and reduce the count */
45: 0xc  bne              /* see if count = 0 */
46: 0x0  nop0             /* and loop if not */
47: 0x0  nop0
48: 0x0  nop0
49: 0xf3 spawn r3         /* spawn off the child */
51: 0xe  ret              /* and return to main program */
52: 0x0  nop0             /* end label */
53: 0x1  nop1
54: 0x1  nop1
```

The supplied ancestor is thus 55 words of 4 bits each, giving a total of 220 bits, 75 of which (those in the pattern providing nops, and those spare in single register specification words) can be flipped without changing the semantics of the program.

The Sierra system

The Sierra system itself is independent of the details of the M1 machine. You can put a different machine architecture in and the Sierra system will not need to be changed.

The memory space in Sierra is represented as 8 bit bytes, so wider bytes in the machine model could be used (only 7 bits without a change to how the ancestor is parsed by the C compiler–see the `-1` at the end).

When you fire it up it drops you into a lisp listener. You can type Common Lisp expressions at that listener. The ancestor is set up as a program instance of genome 0, whose code was reproduced above.

There are a couple of commands you might like to issue at this stage.

`(setparams <copyerror> <cosmicerror>)`	to set mutation parameters
`(setseed <seed>)`	to set the random number seed

The first mutation parameter `<copyerror>` gives the odds that an error will occur during copying a word. It is set by default to 2,000 which means that one in two thousand copy instructions will flip a bit. The `<cosmicerror>` gives the odds that at each instruction executed a bit will be randomly flipped somewhere in the whole soup memory. It is initially set to 500, so that roughly every five hundred instructions a bit somewhere in memory will be flipped.

You can then run the simulated machine with the `(run)` command. It sets the machine running, using a round robin strategy of giving 100 instructions to each program in turn, unless one dies in less instructions than that. The relevant commands you can type are:

`(run)`	to run for 100,000,000 more instructions
`(run <n>)`	to run for `<n>` more instructions
`(status)`	to get the current status of the run
`(stop)`	to stop it running immediately

Sierra keeps track of instances, i.e., copies of code somewhere in memory with an associated register set and energy level. It also keeps track of genomes, and whenever a new program is spawned it records exactly what genome it had at the time (the genome may later get altered by a cosmic ray). At times the system may print the name of a program (like `m347`, the 347th machine spawned beyond the ancestral program), or a genome (like `g6853`, the 6,853rd genome produced beyond the genome of the ancestral program.

Sierra also keeps track of how memory is used. We will refer to an allocated block of memory, whether yet containing a running program or not, as a *body*. If the block is running as a separate program, i.e., it has been spawned by its parent, we will say it is *alive*.

As the machine is running in the background you can use various commands to see what is happening. These include:

`(rep)`	to report on the current status of the population
`(best)`	to print the ids of the best genomes so far
`(populations)`	prints a list of most populous genomes of each size
`(genomes <size>)`	to list well performing genomes of that size
`(genome-list <size)`	same but returns list of genome objects
`(print-genome <id|gen>)`	tells about that genome's success
`(print-code <id|gen>)`	lists its code

Below is an excerpt from a report that was generated by `(rep)`. This was produced with the code as distributed and doing `(run)` with the default random seed of 255. It has a listing for every size block of memory that is currently active. The `bodies` column says how many blocks there are of that size, while the `alive` column says how many of them are running alive. The difference is those that have been allocated but not yet spawned. The `spawned` column says how many programs currently in the `alive` column have ever successfully spawned a program. It must be the case that `spawned` ≤ `alive`.

The final column, `selfgene`, looks up the genome of each currently alive program, and says whether an instance of that genome has ever successfully reproduced an exact copy of itself. Notice that if there are multiple instances of a particular genome alive it will get counted multiple times. It must be the case that `selfgene` ≤ `alive`.

```
#? (rep)
size  bodies/ alive spawned/selfgene
  16      59/    14       5/      0
  17      52/     5       2/      0
  18      89/    36       1/      0
  19      64/     8       4/      0
  20      45/     6       3/      0
  21      95/    29       8/     13
  22      64/    27       3/      0
  23      37/     2       1/      0
  24      59/    18       3/      0
  25      37/     1       1/      0
  26      49/    15       1/      0
  27     213/   123      10/      7
...
  48       5/     1       0/      0
  49      18/     9       2/      0
  50       7/     2       1/      0
  51      18/    15       2/      0
  52      16/     3       1/      0
  53      26/    16       2/      0
  54      70/    46       9/      0
  55    5146/  3589    1482/   2063
  56      96/    61      16/     32
  57      84/    52      26/     40
...
 121      13/    12       1/      0
 127       4/     4       0/      0

7662 bodies, 393837/1048576 = 37.559%
genomes: 153654, individuals: 213803/4858, ticks: 100000075
```

The last two lines above give some summary information. In this case there are 7,662 blocks of memory allocated, which occupy 393,837 words of the 1,048,576 word soup, meaning that it is 37.559% full. There have been 100,000,075 instructions executed (this is not a multiple of one hundred because sometimes programs died in the middle of their one hundred instruction quotas), and there have been programs with 153,654 different genomes produced. A total of 213,803 programs have been produced, of which 4,858 are still alive.

In searching for interesting genomes, the command `(best)` tells you something about the best genomes at each program size:

```
#? (best)
size  selfrep genome   gen |  ratio  genome   gen
  21      24  g153       1  | 0.5000  g19269    1
  24       3  g3448      2  | 0.3333  g16703    2
  25       1  g4169      3  | 0.5000  g4169     3
  26       4  g21116     3  | 0.8000  g21116    3
  27      37  g318       1  | 0.6667  g74647    2
...
  48       1  g25480     2  | 0.5000  g25480    2
  49       4  g37218     5  | 0.6667  g37218    5
  51       3  g7143      3  | 0.6000  g7143     3
  52       1  g8476      4  | 0.5000  g8476     4
  54       1  g150539    2  | 0.5000  g150539   2
  55   14962  g0         0  | 0.9796  g4107     2
  56      42  g50966     3  | 0.9565  g86811    5
  57      34  g93033     5  | 0.9286  g99878    5
...
  64      58  g83456     4  | 0.9355  g83456    4
  65       4  g73668     3  | 0.8000  g73668    3
  66       7  g81990     2  | 0.8750  g81990    2
...
```

The `selfrep` column tells how many times the specified genome has managed to reproduce an exact copy of itself so far in this run. The `ratio` column indicates for a possibly different genome the proportion of instances of a genome which managed to self reproduce.

Thus, for instance, for genomes of size 55, above, genome `g0` was able to self reproduce 14,962 times, but a more efficient reproducer of size 55 was $g4107$ which reproduced itself in 98% of its instances. The two `gen` columns refer to how many different genomes had to be mutated to get to this from the ancestor.

Another way to look at the best performers is with `(populations)`.

```
#? (populations)
size   genome   total num  selfrep  gen
  21   g153          667       24    1
  24   g3448          14        3    2
  25   g95529         12        1    4
  26   g21116          5        4    3
  27   g318          307       37    1
...
  48   g25480          2        1    2
  49   g37218          6        4    5
  51   g7143           5        3    3
  52   g8476           2        1    4
  54   g150539         2        1    2
  55   g0          15913    14962    0
  56   g50966            60         42    3
  57   g683              44         20    3
  58   g39729            33         20    2
...
```

Here for each size the genome is identified that had the most instances (printed under `total num`) as long as it was able to self reproduce. The `selfrep` column tells how many times it was able to do that, and as before the `gen` column says how many generations it was from the ancestor.

These genomes, and ones of smaller size, are interesting to investigate further. You might use `print-genome` and `print-code` to do that. In fact, let's do it!

```
#? (print-genome 0)
Genome: 0 (parent -1) [generation 0]
 55 bytes. 15913 instances.  46652 children.  14962 selfrep.
 appeared: [0, 99945166]
 ancestor history:
        g0      size: 55
()
#? (print-genome 4107)
Genome: 4107 (parent 211) [generation 2]
 55 bytes. 49 instances.  70 children.  48 selfrep.
 appeared: [11572629, 65631743]
 ancestor history:
        g4107   size: 55
        g211    size: 55
        g0      size: 55
()
```

From this we can see that the 15, 913 instances of genome `g0` produced 46, 652 children of which 14, 962 were exact copies of the parent's genome. On the other hand the 49 instances of genome `g4107`, produced 48 exact copies, but that is a very high number of identical copies over a small sample.

Further we see from the parentheses that the first instance of `g4107` was produced by a program that had genome `g211`, which itself was originally produced by a program with the ancestral genome. So `g4107` is only a second genomic generation from the ancestor. It may have been produced via a chain of many more program instances, but there was only two inexact reproduction events. We also see that the first instance of `g4107` appeared after 11, 572, 629 instructions, whereas the last one appeared after 65, 631, 743 instructions–no doubt it has now died off.

The command `(genomes <size>)` prints a list of well performing genomes of the given size. Those that managed to self reproduce have asterisks next to them. Others may just be ones that produced lots of offspring. Here is an example:

```
#? (genomes 21)

Genomes: g149795* g153* g56112* g86940 g134260 g129259 g2636 g152057
 g151658 g41439* g152148* ()
```

When we investigate one of these more closely we see:

```
#? (print-genome 153)
Genome: 153 (parent 0) [generation 1]
 21 bytes. 667 instances.  345 children.  24 selfrep.
 appeared: [1654280, 99946801]
 ancestor history:
        g153    size: 21
        g0      size: 55
()
#? (print-code 153)

  0: 0x0  nop0
  1: 0x0  nop0
  2: 0x1  nop1
  3: 0x2  find
  4: 0x1  nop1
  5: 0x1  nop1
  6: 0x0  nop0
  7: 0x30 sto r0
  9: 0x2  find
 10: 0x1  nop1
 11: 0x0  nop0
 12: 0x1  nop1
 13: 0x50 sub r0
 15: 0x31 sto r1
 17: 0xd  call
 18: 0x0  nop0
 19: 0x1  nop1
 20: 0x0  nop0
()
```

We see that instruction 12 mutated from `nop0` to a `nop1`, which resulted in an ancestor finding a much shorter piece of code to copy, but it has no instruction to loop back so it is not able to make lots of copies of itself. Genome `g4169`, it turns out, has two mutations, and almost gets it right but still suffers from some problems.

There is another version of `genomes`, namely `genome-list`, which returns a lisp list of genomes. These can be fed to `print-genome` or `print-code`, or they can be used with a number of other procedures which can be found by typing `(morehelp)`. Here is that help message:

```
(number-genomes)               total number of genomes
(get-genome <id>)              returns genome with <id>
(genome-id <genome>)           id of <genome>
(genome-instances <genome>)    how many of these existed
(genome-spawned <genome>)      how many of these reproduced
(genome-selfrep <genome>)      how many exact reproductions
(genome-generations <genome>)  how many generations to produce
```

```
(genome-length <genome>)       length of <genome>
(genome-parent <genome>)       parent genome of <genome>
```

This lets you write little lisp scripts to poke around the genome data base and look for interesting things. Since you can dynamically load new lisp source code with `(load "filename")` you can have a carcass lying around, say after running $1,000,000,000$ instructions, and dynamically decide on new search routines without having to recompile the Sierra system and running the whole thing again.

Hacking on the code

If you wish to make a new machine architecture you will only have to change file `m1.c`. If you want to change the statistics that are collected you will have to change `sierra.c`.

If you want to make different things available on the Lisp side you will have to change `sierra.c`, and possibly `sierra.lisp`. Look at the bottom of `sierra.c` to see how to place a C procedure into Lisp's namespace. Note that such procedures can only take up to three arguments. Also note that genomes are a Lisp data type so it is alright to return one of them, as does `get_ancestor` (to see how it prints in Lisp try `(get-ancestor)`). So you can use the accessors described by `(morehelp)` to write all sorts of automated search procedures in Lisp to investigate the structure of genome space.

The research questions

Get the source files following the instructions on the web at
`http://www.ai.mit.edu/courses/6.836/handouts/handouts.html`. Then `make` in main directory to get an executable image `m1`. Make sure you don't hand in anything without comments in the code as well as qualitative descriptions of what is going on, especially when the answer is in the form of an ancestral machine or a parasite.

1. [**1 point**] Use the supplied files and run the simulation. Try it with different random seeds. To get a fresh run you need to restart the program. Look around for the smallest parasite you can find. All the interesting stuff seems to happen in the first half giga instructions. Once you have found a nice parasite give us a code listing for it, and explain how it could possibly self reproduce.

2. [**2 points**] Describe the "perfect" parasite and how it might possibly evolve from the ancestoral creature. Have you been able to observe a perfect parasite? How can you change the ancestoral creature to make it more likely to arise? Make some runs with such a new ancestoral creature and describe the results.

3. [**2 points**] Experiment with different settings for cosmic rays and the copying errors. Make a few runs at each setting you choose for, say, one hundred million instructions. Observe what happens to the system and come up with some qualitative explanation for what you see. If you notice any particularly interesting genome tell us about it.

4. [**2 points**] You can suppress all copying errors and cosmic rays by using `(errors 0)` at startup (note that `(errors 1)`, will allow errors to happen as normal). Try running the system with no errors. You will still get mutated creatures!! How could that be? (Hint: if you also say `(dealloc 0)` at startup this will force the memory of a dead creature to revert to zero, but still you get some mutated creatures, but perhaps not as many as before.)

5. [2 points] Modify the code for the underlying machine so that when searching for a label it searches for a matching label rather than a complementary label. Modify the ancestral code accordingly. Tell us what happens (and show us the code of the ancestral program and any parasites).

6. [1 point] Make a more drastic set of changes to the simulated machine. Be as creative as you want. Show us the results (hopefully interesting) of what happens when you run this system.