6.836 Embodied Intelligence, 2001
Research Assignment 3
Issued March 9, due March 23.

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 fourth and fifth 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).

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 accumlate 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 also currently 20 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*
`0x6`	`add`	add register to *acc*
`0x7`	`go`	go to a label
`0x8`	`alloc`	allocate a chunk of memory whose size is specified by a register
`0x9`	`copy`	copy word pointed to by one register to word pointed to by another register 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`. 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 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 210 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–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 10,000 which means that one in ten 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 1,000, so that roughly every thousand 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 when ever 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
`(genomes <size>)`	to list well performing genomes of that size
`(print-genome <id>)`	tells about that genome's success
`(print-code <id>)`	lists its code

Below is an excerpt from from a report that was generated by `(rep)`. 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` $\leq$ `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` $\leq$ `alive`.

```
#? (rep)
size  bodies/ alive spawned/selfgene
  17      11/     1        1/     0
  19       5/     1        1/     0
  20      18/     3        3/     0
...
  48     567/   373      219/   353
  49       2/     2        2/     0
  50      12/     4        3/     1
  51       5/     1        0/     0
  52       4/     1        1/     0
  54       3/     1        0/     0
  55      10/     5        4/     1
  57       7/     2        1/     2
  58      14/     7        2/     0
  59     688/   449      248/   384
  60       4/     3        2/     2
...
2169 bodies, 112534/1048576 = 10.732%
genomes: 100456, individuals: 152807/1336, ticks: 100000011
```

The last two lines above give some summary information. In this case there are 2,169 blocks of memory allocated, which occupy 112,534 words of the 1,048,576 word soup, meaning that it is 10.732% full. There have been 100,000,011 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 100,456 different genomes produced. A total of 152,807 programs have been produced, of which 1,336 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  |  ratio  genome
  21        1  g82174  | 0.0182  g82174
  23        4  g13662  | 0.1379  g13662
  25        1  g85340  | 0.0323  g85340
  30       73  g21973  | 0.2819  g21973
...
  48     1860  g85827  | 0.9429  g96040
  49        3  g8640   | 0.5000  g95342
  50        8  g96637  | 0.7500  g199
  51        1  g98855  | 0.5000  g98855
  52       17  g89997  | 0.9444  g89997
  53        2  g99615  | 0.6667  g99615
  54        1  g52267  | 0.2500  g52267
  55     2040  g2      | 0.8571  g313
  56        1  g52     | 0.5000  g52
  57        2  g1895   | 0.6667  g86737
  58        3  g33684  | 0.7500  g33684
  59     3308  g6200   | 0.9231  g75120
...
```

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 possible different genome the proportion of instances of a genome which managed to self reproduce.

Thus, for instance, for genomes of size 55, above, genome **g2** was able to self reproduce 2,040 times, but a more efficient reproducer was $g313$ which reproduced itself in 86% of its instances. 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 2)
Genome: 2(0)[1]
 55 bytes. 2559 instances.  3024 children.  2040 selfrep.
 appeared: [75428, 35672311]
()
#? (print-genome 313)
Genome: 313(2)[2]
 55 bytes. 7 instances.  18 children.  6 selfrep.
 appeared: [8266279, 10093795]
()
```

From this we can see that the 2,559 instances of genome **g2** produced 3,024 children of which 2,040 were exact copies of the parent's genome. On the other hand the 7 instances of genome **g313**, only produced 6 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 **g2** was produced by a program that had the ancestral genome. So **g2** is only a first (the **[1]**) genomic generation from the ancestor. It may have been produced via a chain of many more program instances, but there was only one inexact reproduction event. We also see that the first instance of **g2** appeared after 75,428 instructions, whereas the last one appeared after 35,672,311 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 48)
Genomes: g85827* g99672* g99130* g99310* g96040* g94632* g76151 g86165*
g96766* g98546* g99986* g99864 g99795* g98790* g88510 g99132* g98530
g94170* g100116* g100006 g98378 g96092* g96491* ()
```

When we investigate a couple of these more closely we see:

```
#? (print-genome 76151)
Genome: 76151(1586)[4]
 48 bytes. 1 instances.  137 children.  0 selfrep.
 appeared: [67807280, 67807280]
()
#? (print-genome 85827)
Genome: 85827(83599)[7]
 48 bytes. 2293 instances.  2366 children.  1860 selfrep.
 appeared: [76365935, 99991952]
()
```

One instance of genome `g76151` produced 137 offspring, but never self reproduced. Genome `g85827` which is a seventh genomic generation genome had lots of instances which self reproduced. The ancestor genome for `g85827` was genome `83599`.

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 could write accessors and then be able 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 the fell directory, and `make` in main directory to get an executable image `m1`.

1. 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 (with random seed 37, plenty of parasites of size 40 will pop up in the first hundred million, but it is possible to find much smaller ones). 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. 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.

3. Write a new ancestral machine that has a different decomposition of work between the main program and the subroutine. Experiment (using the old verions of the cosmic and copying parameters) with versions that do give rise to parasites and those that do not. Give a qualitative explanation (if there are always parasites then tell us about that too). Show us the code of the parasites.

4. 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).

5. 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.