ICFP Programming Contest 2014

Specification

Introduction

For this year's ICFP programming contest we thought it would be fun to do a bit of '80s software and hardware archaeology. We just so happen (via friends of friends and bankruptcy asset sales) to have got our hands on a bunch of internal documents from *LamCo*. Back in the '80s, LamCo were a failed manufacturer of arcade games. You've probably never heard of them because they never managed to get a product out and were beaten by their better known rivals.

What we've found from the cache of documents is that they had some rather interesting technology. We also found that they had a spectacularly dysfunctional development process marked by bitter internal rivalries, which we suspect goes a long way to explain their failed projects.

So the contest task involves delving into LamCo's quirky old software and hardware technology, reconstructing the arcade hardware in simulation, and writing software for the arcade platform to play the game—and to play it better than your rivals!

The History We've Uncovered

LamCo were building an arcade game where you control a little character called "Lambda-Man" who runs around in a maze eating pills and evading ghosts. The game bears a striking resemblance to another well known arcade game. We don't know exactly why they ended up being so similar, though we rather suspect LamCo of stealing ideas from their more successful rival. For reasons best known to themselves, LamCo decided to make a two player version of their game where two Lambda-Man characters battle it out to swipe the most pills from under the noses of the ghosts. Because they also wanted to have a one player mode they found that they needed an AI to play the opposing Lambda-Man. This and the usual pressure to meet tight deadlines led to a catastrophic decision by The Management to force two rival engineering teams to work together on the project. The Management got one team to work on the ghost AIs and game mechanics and another team to work on the Lambda-Man AI. The Management claimed this was because they wanted to be proactive not reactive with their blue-sky thinking on this mission critical project, and believed this to be a win-win situation, harvesting low-hanging fruit with client focused deliverables. Because of time pressure and The Management, each team was allowed to use their favourite technology stack and the plan was to integrate it all together at the end. The ghost and game mechanics team were led by engineers who seemed to believe that 8-bit microprocessors are the be all and end all of computing. The Lambda-Man AI team were led by engineers obsessed with LISP and all its arcanery.

Of course the end product ended up reflecting the teams that had built it. The main motherboard had several 8-bit microcontrollers to implement the ghost AIs and the game mechanics. It also had a customised coprocessor for running the Lambda-Man AI. The software also reflected the different teams' philosophies: the ghost AIs and game mechanics were written directly in assembly while the Lambda-Man AI was written in some dialect of LISP.

We don't know exactly why the project failed. It could have been schedule slippage from integrating the different technologies, infighting between the two teams, the horrendous bill of materials or all of the above. But surely not The Management. All we know for sure is that the project was ultimately cancelled and LamCo filed for bankruptcy.

Summary of the Task

According to the engineers' notes that we've found, they did manage to get working hardware and software before the project was cancelled. Unfortunately we have not been able to recover the original software, but we do have fairly detailed design notes on the hardware.

So far we (that is, the judges) have a software simulation of the arcade hardware. What we want you to do is to write the software to run on the arcade hardware.

Lightning Round Summary

For the lightning round we will play the "classic" version of the game where it is just a single Lambda-Man against the ghosts.

Your challenge is to write an AI for the Lambda-Man. It will play against ghost AIs supplied by the judges and in mazes supplied by the judges.

You will be supplied with several of the mazes and with the assembly code for a few (not very smart) ghost AIs.

The lightning round runs for the first 24 hours of the competition, so submissions must be in before 12:00 UTC 26/07/2014.

Full Round Summary

The full round will be broadly the same as the lightning round, but with a twist. The details will be made available at the end of the lightning round. Check the front page for the link once the lightning round has closed.

The full round runs for the full 72 hours of the competition, so submissions must be in before 12:00 UTC 28/07/2014.

The Rest of this Spec

In the remainder of this spec, you will find:

- The complete description of the Lambda-Man game rules;
- The details of how your solution will be scored;
- The description of the microcontroller used for the ghost AIs;
- The description of the processor used for the Lambda-Man AI;
- Details on how to submit your solution.

Reference Material

To help you get started we are providing a reference implementation of the game rules and the Lambda-Man processor.

- http://icfpcontest.org/reference.html

This site also contains other reference material that you might find useful, such as sample maps.

Please note that the reference implementation is provided just as a convenience. We cannot guarantee that it is correct, or that it will always be available. However, if you find any discrepancies between this specification document and the reference implementation (or other material), then please let the judges know as soon as possible, and we will do our best to keep things updated.

The Lambda-Man game rules

Objectives

A *Lambda-Man* lives in a two-dimensional maze made up of *walls*, and must eat as many *pills* as he can, while avoiding the *ghosts* who chase him. Lambda-Man has three lives, and if a ghost catches Lambda-Man, then he loses a life. When there are no more Lambda-Man lives, the game is over. When all the pills are eaten, Lambda-Man has completed the level.

In addition to pills, a Lambda-Man may also eat *power pills*. These gives every Lambda-Man the ability to eat ghosts for a short period of time.

Bonus points are awarded for eating the *fruit*, which appears after a specific period of time at a specific location on the map before disappearing.

By convention, the following symbols are used to represent the various elements of the game:

Symbol	Element
`<space>`	Empty
`#`	Wall
`.`	Pill
`o`	Power Pill
`%`	Fruit
`\`	Lambda-Man
`=`	Ghost

Mechanics

The world is entirely deterministic, and runs on a tick-by-tick basis.

On each tick:

1. All Lambda-Man and ghost moves scheduled for this tick take place. The next move is also scheduled at this point. (Note that Lambda-Man and the ghosts do not move every tick, only every few ticks; see the ticks section below.)
2. Next, any actions (fright mode deactivating, fruit appearing/disappearing) take place.
3. Next, we check if Lambda-Man is occupying the same square as pills, power pills, or fruit:
    1. If Lambda-Man occupies a square with a pill, the pill is eaten by Lambda-Man and removed from the game.
    2. If Lambda-Man occupies a square with a power pill, the power pill is eaten by Lambda-Man, removed from the game, and fright mode is immediately activated, allowing Lambda-Man to eat ghosts.
    3. If Lambda-Man occupies a square with a fruit, the fruit is eaten by Lambda-Man, and removed from the game.
4. Next, if one or more visible ghosts are on the same square as Lambda-Man, then depending on whether or not fright mode is active, Lambda-Man either loses a life or eats the ghost(s). See below for details.
5. Next, if all the ordinary pills (ie not power pills) have been eaten, then Lambda-Man wins and the game is over.
6. Next, if the number of Lambda-Man lives is 0, then Lambda-Man loses and the game is over.
7. Finally, the tick counter is incremented.

Losing a life

If at the end of a tick Lambda-Man is in the same square as a visible ghost and fright mode is not active then Lambda-Man loses a life. In this case, Lambda-Man and all the ghosts are immediately returned to their starting positions and starting directions (so that at the beginning of the next tick, Lambda-Man and the ghosts are in their starting positions).

Power Pills

When a power pill is eaten, all ghosts turn around and move in the opposite direction to their previous move, and fright mode is enabled. While in fright mode, if a ghost occupies the same square as a Lambda-Man, the ghost is eaten. When a ghost is eaten, it is returned to its starting position and starting direction, and is invisible until fright mode expires. While invisible, the ghost can neither eat nor be eaten.

If a power pill is eaten during fright mode, the fright mode tick count is reset.

When fright mode expires, all ghosts become visible.

Scoring

The aim of the game is to achieve the highest score, which is the sum of all the scores achieved on all levels. Points are awarded as follows.

Each pill eaten is worth 10 points.

Each power pill eaten is worth 50 points.

Each fruit eaten is worth points depending on its flavour. The flavour of a fruit is determined by the level as described below:

Level	**Flavour**	**Points**
1	Cherry	100
2	Strawberry	300
3	Peach	500
4	Peach	500
5	Apple	700
6	Apple	700
7	Grapes	1000

Level	Flavour	Points
8	Grapes	1000
9	Galaxian	2000
10	Galaxian	2000
11	Bell	3000
12	Bell	3000
> 12	Key	5000

The level of a map is determined by its area. Given a map of size `mapWidth * mapHeight`, the level is the number which satisfies:

```
100 * (level - 1) < mapWidth * mapHeight <= 100 * level
```

For example, a map of size `15 * 18 = 270` is a level 3 map, since `200 < 270 <= 300`.

While in fright mode, the first ghost eaten is worth 200 points. Each subsequent ghost eaten until another power-pill is eaten is worth double the previous one, up to a limit of 1600 points:

Number of ghosts eaten	Points
First	200
Second	400
Third	800
Fourth	1600
Fifth and subsequent	1600

If Lambda-Man manages to eat all of the pills on a level, he is awarded with a bonus: his score is multiplied by the remaining lives plus one. For example, if the Lambda-Man is on his last life, the score is doubled.

Ticks

The world runs tick-by-tick. The Ultimate Tick Clock (UTC) gives the current tick time, and is counted from 1 at the beginning of each game. A game runs until the End Of Lives (EOL), which happens either when Lambda-Man runs out of lives, or if a particular UTC is reached, whereupon lives are set to 0:

Event	UTC
End Of Lives	127 * mapWidth * mapHeight * 16

Two fruits appear on every level, at a fixed UTC. The second fruit will appear whether or not the first fruit was eaten.

Event	UTC
Fruit 1 appears	127 * 200
Fruit 2 appears	127 * 400

Each fruit remains in the game until either it is eaten, or it expires at a fixed UTC.

Event	UTC
Fruit 1 expires	127 * 280
Fruit 2 expires	127 * 480

Each power pill eaten triggers fright mode, which expires after a fixed duration counted from when the last power pill is eaten.

Event	Ticks duration
Fright mode duration	127 * 20

The Lambda-Man and ghosts move at different speeds. Lambda-Man moves slower when he is eating (pills, power pills or fruit), and the ghosts move slower when they are in fright mode. All moves are at regular intervals, based on their *ticks per move* value which is described below. For example, the first Lambda-Man move occurs at tick 127, the second at tick 254, and so on.

Event	Ticks per move
Lambda-Man	127
Lambda-Man (eating)	137

Event	Ticks per move
Ghost 0	130
Ghost 1	132
Ghost 2	134
Ghost 3	136
Ghost 0 (fright)	195
Ghost 1 (fright)	198
Ghost 2 (fright)	201
Ghost 3 (fright)	204

On a tick when Lambda-Man or a ghost is scheduled to move, their next move is scheduled for the appropriate number of ticks in the future, depending on their current state. For example if Lambda-Man moves into a square with a pill then the next tick on which he will move will be his previous scheduled tick number plus 137.

When there are more than 4 ghosts, these ticks-per-move values are assigned cyclically, such that ghost 4 takes on the values of ghost 0, and so on.

No other event or condition resets the scheduled tick for Lambda-Man or the ghosts (not even entering fright mode, ghosts being eaten, or Lambda-Man being eaten).

Movement

The Lambda-Man can move into any adjacent square that is not occupied by a wall. An adjacent square is one that is up, down, left, or right of another.

When Lambda-Man tries to choose an illegal move, he stops moving.

Ghosts can only move into an adjacent square that is not occupied by a wall. At a tick when a ghost may move, it must move (unless it is surrounded on all four sides by walls). Furthermore, a ghost cannot move in the opposite direction to its current direction, unless that is the only direction available (because it is surrounded on three sides by walls).

The usual consequence of this is that a ghost chooses its direction at a junction and cannot choose to turn back on itself. A junction is a square which has at least three adjacent squares with no walls. For example, the following are all junctions.

```
 # #     # #             # #     # #
## ##    # ##   #####   ## #    ## ##
  =      #=       =       =#      =
#####    # ##   ## ##   ## #    ## ##
         # #     # #     # #     # #
```

Usually, when a ghost encounters a bend, it is forced to continue around the bend. When a ghost encounters a dead end, it is forced to turn around.

An exception is when a powerpill is activated when a ghost is on a bend. Consider the following case, where the ghost comes from above, and the powerpill is subsequently activated.

```
  # #
  # #
### #
   =#
#####
```

When powerpill is activated, the ghost direction is set to up, as if it had come from the wall below its current position. Consequently, the ghost may choose to go up or left. If it goes left, this may seem like the ghost has not had its direction reversed … but it has!

When a ghost chooses an illegal move (or no move at all) at a junction, it is forced to continue in its previous direction if this is legal, and if not, then the first legal direction out of up, right, down, and left, in that order.

At the start of the game, all ghosts and Lambdaman face down.

Tournament scoring

For the lightning round it is your Lambda-Man versus ghosts supplied by the judges and in mazes supplied by the judges.

Your overall score in the lightning round is the sum of your individual scores on a series of games, played in different maps. The actual maps and ghosts used will not be revealed but they will range from easy to hard. The map properties are described below.

Map properties

Maps are rectangular. Map x and y coordinates are indexed from 0. The top left corner is (0,0), so increasing x-coordinates are to the right, and increasing y-coordinates are down.

Maps are like mazes: they consist of corridors 1-square wide, without open spaces. More formally: there are no 2x2 areas consisting only of non-wall squares.

Maps have walls around the edges.

Every pill, power pill and fruit in a map is accessible.

Maps have one and only one fruit location.

The maps vary in size and in the number of ghosts and power pills. Easy maps will be smaller, with fewer ghosts and a higher density of power pills. Harder maps will be larger, with more ghosts and a lower density of power pills. The maximum map size is 256 by 256.

Ghosts and ghost programs

Ghost AI programs are assigned to ghosts in each map. If there are more ghosts in a map than then are ghost AI programs in use then the AI programs are assigned to the ghosts cyclically.

For example, if a map has 4 ghosts and there are 2 AI programs then they are assigned as follows:

Ghost	AI Program
ghost 0	program 1
ghost 1	program 2
ghost 2	program 1
ghost 3	program 2

The order of ghosts is in increasing order of their starting coordinates, where `(x1, y1)` is considered smaller than `(x2, y2)` if `y1 < y2` or if `y1 = y2` and `x1 < x2`.

The system allows at most 4 ghost programs (but up to 256 ghosts, numbered 0-255).

GHost CPU (GHC)

The GHost CPU (GHC) is a conventional 8-bit microcontroller. Each ghost is run by a separate GHC. Although we found a complete copy of the CPU manual, it is quite terse in parts, as it assumes the conventions of the day. For those of you who (because of age or youth) do not remember the 1980s, we have tried to explain some of these conventions.

GHC State

Each register holds an 8-bit unsigned integer (between 0 and 255 inclusive). There are 2 separate memories: a data memory and a code memory, each with 256 locations (numbered from 0 to 255 inclusive). Hence the contents of a register can be interpreted directly as a data memory address, the contents of a data memory location, or a code memory address. The GHC performs all arithmetic modulo 256, so 255 + 1 = 0.

Initialisation and Program Execution

At the start of a game, the GHC's code memory is initialised with a program, as described in the section Code Format. The contents of the code memory does not change during a game. All data memory locations and all registers are initialised to 0.

On a game tick when ghost is scheduled to move, the GHC runs the program, executing up to 1024 instructions:

- At the start of program execution, the PC is initialised to 0.
- An instruction cycle begins with the GHC reading the instruction at the address referenced by the PC from code memory.
- It executes the instruction, as described in the section Instruction Reference, possibly changing the contents

of the data memory and registers.
- At the end of the instruction cycle, if the value of the PC is the same as it was at the start of the instruction cycle, the GHC increments it.
- Execution terminates at the end of an instruction cycle if:
    - the instruction executed was HLT;
    - it was the 1024th instruction executed;
    - or execution of the instruction caused an error.
- The contents of the data memory and registers persist between game ticks.

Code Format

By convention, a GHC program is stored in a file with the extension .ghc (GHost Code).

A program consists of several lines, terminated by newline characters. The contents of a line are whitespace insensitive: multiple consecutive whitespace characters are treated identically to one. A line is either empty (containing only whitespace) or contains an instruction.

A program may contain comments, which are introduced using a semicolon (`;`). Anything from a semicolon until the end of a line (including the semicolon) is ignored. Hence a line containing only a comment is regarded as empty.

An instruction consists of a mnemonic and zero or more arguments. The mnemonic is a case-insensitive sequence of alphabet characters.

An argument is either:

1. a register argument (indicated by its name (`A` to `H` or `PC`));
2. an indirect general-purpose register argument (indicated by its name enclosed in square brackets (`[A]` to `[H]` but not `[PC]`));
3. a constant argument (indicated by its encoding in decimal (`0` to `255`));
4. or the contents of a data memory location (indicated by its address in decimal, enclosed in square brackets (`[0]` to `[255]`)).

The number of arguments required depends on the instruction (see Instruction Reference).

There must be a whitespace character between a mnemonic and the first argument (if any). There must be a comma (`,`) between consecutive arguments.

Instructions may optionally be preceded or followed by whitespace. Arguments and argument-separating commas may optionally be preceded or followed by whitespace.

When the GHC is initialised, each line containing an instruction is stored in the corresponding code memory location. For example, the instruction on the first non-empty line is stored at address 0 and the instruction on the second non-empty line is stored at address 1. As there are only 256 code memory locations available, a program may contain at most 256 instructions.

Instruction Reference

The GHC is able to execute the following instructions:

- `MOV dest,src`

  Copy the value of the `src` argument into the `dest` argument. `dest` may not be a constant.

- `INC dest`

  Add 1 to the value of `dest` and store the result in `dest`. `dest` may not be a constant or the register `PC`.

- `DEC dest`

  Subtract 1 from the value of `dest` and store the result in `dest`. `dest` may not be a constant or the register `PC`.

- `ADD dest,src`

  Add the value of `src` to the value of `dest` and store the result in `dest`. `dest` may not be a constant or the register `PC`.

- `SUB dest,src`

  Subtract the value of `src` from the value of `dest` and store the result in `dest`. `dest` may not be a constant or the register `PC`.

- `MUL dest,src`

  Multiply the value of `src` by the value of `dest` and store the result in `dest`. `dest` may not be a constant or the register `PC`.

- `DIV dest,src`

  Compute the integer division (rounding down towards negative infinity) of `dest` by the value of `src`, and store the result in `dest`. `dest` may not be a constant or the register `PC`. Results in an error if the value of `src` is 0.

- `AND dest,src`

  Bitwise AND the value of `dest` and the value of `src`, storing the result in `dest`. `dest` may not be a constant or the register `PC`.

- `OR dest,src`

  Bitwise OR the value of `dest` and the value of `src`, storing the result in `dest`. `dest` may not be a constant or the register `PC`.

- `XOR dest,src`

  Bitwise XOR the value of `dest` and the value of `src`, storing the result in `dest`. `dest` may not be a constant or the register `PC`.

- `JLT targ,x,y`

  Compare the value of `x` with the value of `y`. If the value of `x` is less than the value of `y`, set the `PC` to the constant value `targ`.

- `JEQ targ,x,y`

  Compare the value of `x` with the value of `y`. If the value of `x` is equal to the value of `y`, set the `PC` to the constant value `targ`.

- `JGT targ,x,y`

  Compare the value of `x` with the value of `y`. If the value of `x` is greater than the value of `y`, set the `PC` to the constant value `targ`.

- `INT i`

  Invoke the interrupt service `i` (see Interrupt Reference).

- `HLT`

  Halt execution of the GHC.

Interrupt Reference

The effect of invoking an interrupt service is architecture-dependent. In the LamCo architecture, the following interrupts are standard:

- `INT 0`
  - In:
    - Register `A`: ghost's new direction

  Set the direction of the ghost. 0 is up; 1 is right; 2 is down; 3 is left.

  The direction of the ghost is set at the end of the game cycle. If the interrupt is called multiple times in a single game cycle, the last interrupt overrides any earlier ones. Using an invalid direction in register `A` is equivalent to retaining the ghost's original direction at the beginning of the game cycle.

- `INT 1`
  - Out:
    - Register `A`: First Lambda-Man's x-ordinate
    - Register `B`: First Lambda-Man's y-ordinate

  Stores the first Lambda-Man's position in registers `A` (x-ordinate) and `B` (y-ordinate). In the single Lambda-Man version of the game, the first Lambda-Man is the only Lambda-Man.

- `INT 2`
    - Out:
        - Register `A`: Second Lambda-Man’s x-ordinate
        - Register `B`: Second Lambda-Man’s y-ordinate

  Stores the second Lambda-Man’s position in registers `A` (x-ordinate) and `B` (y-ordinate). In the single Lambda-Man version of the game, the behaviour of this interrupt is unknown.

- `INT 3`
    - Out:
        - Register `A`: this ghost’s index

  Stores the ghost’s index in register `A`.

- `INT 4`
    - In:
        - Register `A`: ghost index
    - Out:
        - Register `A`: indexed ghost’s starting x-ordinate
        - Register `B`: indexed ghost’s starting y-ordinate

  For the ghost with index read from register `A`, stores its starting position in registers `A` (x-ordinate) and `B` (y-ordinate). If `A` is not a valid ghost index, does nothing.

- `INT 5`
    - In:
        - Register `A`: ghost index
    - Out:
        - Register `A`: indexed ghost’s current x-ordinate
        - Register `B`: indexed ghost’s current y-ordinate

  For the ghost with index read from register `A`, stores its current position in registers `A` (x-ordinate) and `B` (y-ordinate). If `A` is not a valid ghost index, does nothing.

- `INT 6`
    - In:
        - Register `A`: ghost index
    - Out:
        - Register `A`: indexed ghost’s current vitality
        - Register `B`: indexed ghost’s current direction

  For the ghost with index read from register `A`, stores its vitality in register `A`, and its direction in register `B`.

  Vitality:
    - 0: standard;
    - 1: fright mode;
    - 2: invisible.

  If `A` is not a valid ghost index, does nothing.

- `INT 7`
    - In:
        - Register `A`: map square x-ordinate
        - Register `B`: map square y-ordinate
    - Out:
        - Register `A`: contents of map square

  Stores the current contents of map square with index read from registers `A` (x-ordinate) and `B` (y-ordinate) in register `A`. If the co-ordinates lie outside the defined bounds of the map, stores 0.

  Contents:
    - 0: Wall (`#`)
    - 1: Empty (`<space>`)
    - 2: Pill
    - 3: Power pill
    - 4: Fruit

        - 5: Lambda-Man starting position
        - 6: Ghost starting position

- `INT 8`

    - In:
        - Register `PC`
        - Register `A`..`H`

    Sends the current value of the `PC` and all registers to an external debug/trace agent.

Examples

The following GHC programs illustrate the behaviour of some of the instructions:

miner.ghc

```
; Always try to go down.
mov a,2
int 0
hlt
```

flipper.ghc

```
; Go up if our x-ordinate is even, or down if it is odd.
int 3          ; Get our ghost index in A.
int 5          ; Get our x-ordinate in A.
and a,1        ; Zero all but least significant bit of A.
               ; Now A is 0 if x-ordinate is even, or 1 if it is odd.
mov b,a        ; Save A in B because we need to use A to set direction.
mov a,2        ; Move down by default.
jeq 7,b,1      ; Don't change anything if x-ordinate is odd.
mov a,0        ; We only get here if x-ordinate was even, so move up.
int 0          ; This is line 7, the target of the above jump. Now actually set the direction.
hlt            ; Stop.
```

fickle.ghc

```
; Keep track of how long we have spent travelling in each direction.
; Try to go in the direction we've travelled in least.

               ; Count of time spent going in direction 0 is in memory address 0, and so on.
mov a,255      ; A is the min value.
mov b,0        ; B is the corresponding direction.
mov c,255      ; C is the candidate direction for the new min.
               ; Start of loop.
inc c          ; Pick new direction.
jgt 7,[c],a    ; Jump if count of direction C is above best so far.
               ; We have a new min.
mov a,[c]      ; Save new min.
mov b,c        ; Save direction.
jlt 3,c,3      ; Jump target. Loop back if we have not tried all 4 directions.

mov a,b        ; Actually set desired direction.
int 0

int 3          ; Get our ghost index in A.
int 6          ; Get out current direction in B.
inc [b]        ; Increment corresponding count.
hlt            ; Stop.
```

Errors

As mentioned above, if an instruction causes an error then execution halts. If prior to this the new direction of the Ghost was set (with `INT 0`) then this will be the requested move of the ghost and the game will continue. If the direction is not set then the ghost’s requested move will be the same as its previous move.

Lambda-Man CPU

We have been able to recover some documentation about the programming environment used by the Lambda-Man AI team, including their processor ISA and some bits of assembly code. We also know that the Lambda-Man AI team, being LISP fanatics, used some form of LISP but unfortunately we have not been able to find any of their LISP code, nor their compiler.

The LamCo "General Compute Coprocessor" (GCC) is a rather unconventional and—for its time—sophisticated coprocessor. It appears to have been designed as a target for a LISP compiler. Rather than a set of orthogonal instructions, it has a fair number of somewhat specialised instructions that (we presume) must have been useful for a compiler. We did however find a handwritten note by one of the engineers indicating that someone had written a compiler from a variant of Pascal, albeit with some limitations.

Fortunately we do have the original documentation of the processor which describes the instructions and operation in detail, though sadly not very much on how it was intended to be used by a compiler.

The sections below include excerpts from the original documentation along with our own comments.

General architecture

The machine is stack based, with three different stacks used for different purposes. It has—for its time—a relatively large memory. The way the memory is accessed and organised is quite unusual: apart from the stacks that live in memory, the rest of the memory is used for a garbage collected heap, with the GC implemented by the hardware. Because of this there are no general purpose memory access instructions: all memory access is in one of these stacks or in the GC'd heap.

CPU Registers

There are 4 programmer visible machine registers, all of which are for special purposes:

- `%c`: control register (program counter / instruction pointer)
- `%s`: data stack register
- `%d`: control stack register
- `%e`: environment frame register

Memory stacks

Three of the registers point into special data structures in memory:

- Data stack
- Control stack
- Environment frame chain

The remainder of the memory is dedicated to the data heap.

Control register and program code layout

The machine has logically separate address spaces for code versus data. The `%c` register is an instruction pointer, pointing to the next instruction to be executed. Programs are laid out from low addresses to high. The effect of most instructions on the instruction pointer is simply to increment its value by one.

Data stack and register

The data stack is used to save intermediate data values during calculations, and to return results from function calls.

It is a logically contiguous stack. The `%s` register points to the top of the stack.

Many of the instructions simply pop and push values on the data stack. For example the ADD instruction pops two integer values off the stack and pushes back their sum.

Control stack and register

The control stack is used to save return information in function calls. It saves return address and environment frame pointers.

It is a logically contiguous stack.

Only the complex control flow instructions affect the control stack and register. See `SEL`/`JOIN` and `AP`/`RAP`/`RTN` for details.

Environment frames and register

The environment is used for storing local variables, including function parameters. There is an instruction for loading values from the environment onto the top of the data stack. The environment consists of a chain of frames, which is used to implement nested variable scopes within higher level languages, such as local blocks with extra local variables and functions.

The environment is more complex than the two stack structures. Rather than a contiguous stack, it consists of a chain of environment frames. Each frame contains a pointer to its parent frame. In fact the chain of frames is **not** strictly a stack: the lifetime of the frames is not always a simple LIFO stack order. Because of this the frames are managed by the hardware GC.

Each frame consists of a pointer to its parent frame and zero or more data values. The `%e` register points to the local frame. The machine has direct support for loading a value from the local frame or any of its parent frames, which in general requires following the chain of environment frames.

(We believe in the real hardware this feature was implemented in microcode, with internal registers to cache the outermost frame and a fixed number of the inner most frames for quick access).

Data heap and data values

The machine makes extensive use of dynamic allocations and has built-in support for a garbage collected data heap. Values in the data stack and in environment frames can be pointers into the heap.

There are three types of data value: integers, pairs and closures. Integers are represented in the usual way as binary signed 32-bit integers. Pairs and closures are represented by pointers to objects in the data heap. The three types of values are distinguished by tag bits. The tag bits are not software visible (except for the `ATOM` instruction) but are used by the hardware for error checking.

There are three instructions for manipulating pairs: allocating a pair, accessing the first and accessing the second component of a pair.

All program data structures have to be represented using combinations of pairs and integers (and sometimes closures).

Instruction reference

```
LDC - load constant

Synopsis: load an immediate literal;
          push it onto the data stack
Syntax:  LDC $n
Example: LDC 3
Effect:
  %s := PUSH(SET_TAG(TAG_INT,$n),%s)
  %c := %c+1
```

```
LD - load from environment

Synopsis: load a value from the environment;
          push it onto the data stack
Syntax:  LD $n $i
Example: LD 0 1
Effect:
  $fp := %e
  while $n > 0 do            ; follow chain of frames to get n'th frame
  begin
    $fp := FRAME_PARENT($fp)
    $n := $n-1
  end
  if FRAME_TAG($fp) == TAG_DUM then FAULT(FRAME_MISMATCH)
  $v := FRAME_VALUE($fp, $i) ; i'th element of frame
  %s := PUSH($v,%s)          ; push onto the data stack
  %c := %c+1
Notes:
  Values within a frame are indexed from 0.


ADD - integer addition

Synopsis: pop two integers off the data stack;
          push their sum
Syntax: ADD
Effect:
  $y,%s := POP(%s)
  $x,%s := POP(%s)
  if TAG($x) != TAG_INT then FAULT(TAG_MISMATCH)
  if TAG($y) != TAG_INT then FAULT(TAG_MISMATCH)
  $z := $x + $y
  %s := PUSH(SET_TAG(TAG_INT,$z),%s)
  %c := %c+1


SUB - integer subtraction

Synopsis: pop two integers off the data stack;
          push the result of subtracting one from the other
Syntax: SUB
Effect:
  $y,%s := POP(%s)
  $x,%s := POP(%s)
  if TAG($x) != TAG_INT then FAULT(TAG_MISMATCH)
  if TAG($y) != TAG_INT then FAULT(TAG_MISMATCH)
  $z := $x - $y
  %s := PUSH(SET_TAG(TAG_INT,$z),%s)
  %c := %c+1


MUL - integer multiplication

Synopsis: pop two integers off the data stack;
          push their product
Syntax: MUL
Effect:
  $y,%s := POP(%s)
  $x,%s := POP(%s)
  if TAG($x) != TAG_INT then FAULT(TAG_MISMATCH)
  if TAG($y) != TAG_INT then FAULT(TAG_MISMATCH)
  $z := $x * $y
  %s := PUSH(SET_TAG(TAG_INT,$z),%s)
  %c := %c+1


DIV - integer division

Synopsis: pop two integers off the data stack;
          push the result of the integer division of one of the other
          (integer division rounds down towards negative infinity)
Syntax: DIV
Effect:
  $y,%s := POP(%s)
  $x,%s := POP(%s)
  if TAG($x) != TAG_INT then FAULT(TAG_MISMATCH)
  if TAG($y) != TAG_INT then FAULT(TAG_MISMATCH)
  $z := $x / $y
  %s := PUSH(SET_TAG(TAG_INT,$z),%s)
  %c := %c+1


CEQ - compare equal
```

```
Synopsis: pop two integers off the data stack;
          test if they are equal;
          push the result of the test
Syntax: CEQ
Effect:
  $y,%s := POP(%s)
  $x,%s := POP(%s)
  if TAG($x) != TAG_INT then FAULT(TAG_MISMATCH)
  if TAG($y) != TAG_INT then FAULT(TAG_MISMATCH)
  if $x == $y then
    $z := 1
  else
    $z := 0
  %s := PUSH(SET_TAG(TAG_INT,$z),%s)
  %c := %c+1


CGT - compare greater than

Synopsis: pop two integers off the data stack;
          test if the first is strictly greater than the second;
          push the result of the test
Syntax: CGT
Effect:
  $y,%s := POP(%s)
  $x,%s := POP(%s)
  if TAG($x) != TAG_INT then FAULT(TAG_MISMATCH)
  if TAG($y) != TAG_INT then FAULT(TAG_MISMATCH)
  if $x > $y then
    $z := 1
  else
    $z := 0
  %s := PUSH(SET_TAG(TAG_INT,$z),%s)
  %c := %c+1


CGTE - compare greater than or equal

Synopsis: pop two integers off the data stack;
          test if the first is greater than or equal to the second;
          push the result of the test
Syntax: CGTE
Effect:
  $y,%s := POP(%s)
  $x,%s := POP(%s)
  if TAG($x) != TAG_INT then FAULT(TAG_MISMATCH)
  if TAG($y) != TAG_INT then FAULT(TAG_MISMATCH)
  if $x >= $y then
    $z := 1
  else
    $z := 0
  %s := PUSH(SET_TAG(TAG_INT,$z),%s)
  %c := %c+1


ATOM - test if value is an integer

Synopsis: pop a value off the data stack;
          test the value tag to see if it is an int;
          push the result of the test
Syntax: ATOM
Effect:
  $x,%s := POP(%s)
  if TAG($x) == TAG_INT then
    $y := 1
  else
    $y := 0
  %s := PUSH(SET_TAG(TAG_INT,$y),%s)
  %c := %c+1


CONS - allocate a CONS cell

Synopsis: pop two values off the data stack;
          allocate a fresh CONS cell;
          fill it with the two values;
          push the pointer to the CONS cell
Syntax: CONS
Effect:
  $y,%s := POP(%s)
  $x,%s := POP(%s)
  $z := ALLOC_CONS($x,$y)
  %s := PUSH(SET_TAG(TAG_CONS,$z),%s)
  %c := %c+1


CAR - extract first element from CONS cell

Synopsis: pop a pointer to a CONS cell off the data stack;
          extract the first element of the CONS;
          push it onto the data stack
Syntax: CAR
Effect:
  $x,%s := POP(%s)
  if TAG($x) != TAG_CONS then FAULT(TAG_MISMATCH)
  $y := CAR($x)
  %s := PUSH($y,%s)
  %c := %c+1


CDR - extract second element from CONS cell

Synopsis: pop a pointer to a CONS cell off the data stack;
          extract the second element of the CONS;
          push it onto the data stack
Syntax: CDR
Effect:
  $x,%s := POP(%s)
  if TAG($x) != TAG_CONS then FAULT(TAG_MISMATCH)
  $y := CDR($x)
  %s := PUSH($y,%s)
  %c := %c+1


SEL - conditional branch

Synopsis: pop an integer off the data stack;
          test if it is non-zero;
          push the return address to the control stack;
          jump to the true address or to the false address
Syntax:  SEL $t $f
Example: SEL 335 346  ; absolute instruction addresses
Effect:
  $x,%s := POP(%s)
  if TAG($x) != TAG_INT then FAULT(TAG_MISMATCH)
  %d := PUSH(SET_TAG(TAG_JOIN,%c+1),%d)   ; save the return address
  if $x == 0 then
    %c := $f
  else
    %c := $t


JOIN - return from branch

Synopsis: pop a return address off the control stack, branch to that address
```

```
Syntax:  JOIN
Effect:
  $x,%d := POP(%d)
  if TAG($x) != TAG_JOIN then FAULT(CONTROL_MISMATCH)
  %c := $x
```

LDF - load function

```
Synopsis: allocate a fresh CLOSURE cell;
          fill it with the literal code address and the current
            environment frame pointer;
          push the pointer to the CLOSURE cell onto the data stack
Syntax:  LDF $f
Example: LDF 634      ; absolute instruction addresses
Effect:
  $x := ALLOC_CLOSURE($f,%e)
  %s := PUSH(SET_TAG(TAG_CLOSURE,$x),%s)
  %c := %c+1
```

AP - call function

```
Synopsis: pop a pointer to a CLOSURE cell off the data stack;
          allocate an environment frame of size $n;
          set the frame's parent to be the environment frame pointer
            from the CLOSURE cell;
          fill the frame's body with $n values from the data stack;
          save the environment pointer and return address
            to the control stack;
          set the current environment frame pointer to the new frame;
          jump to the code address from the CLOSURE cell;
Syntax:  AP $n
Example: AP 3      ; number of arguments to copy
Effect:
  $x,%s := POP(%s)            ; get and examine function closure
  if TAG($x) != TAG_CLOSURE then FAULT(TAG_MISMATCH)
  $f := CAR_CLOSURE($x)
  $e := CDR_CLOSURE($x)
  $fp := ALLOC_FRAME($n)      ; create a new frame for the call
  FRAME_PARENT($fp) := $e
  $i := $n-1
  while $i != -1 do           ; copy n values from the stack into the frame in reverse order
  begin
    $y,%s := POP(%s)
    FRAME_VALUE($fp,$i) := $y
    $i := $i-1
  end
  %d := PUSH(%e,%d)                     ; save frame pointer
  %d := PUSH(SET_TAG(TAG_RET,%c+1),%d)  ; save return address
  %e := $fp                             ; establish new environment
  %c := $f                              ; jump to function
```

RTN - return from function call

```
Synopsis: pop a return address and environment frame
            pointer off of the control stack;
          restore the environment;
          jump to the return address
Syntax:  RTN
Effect:
  $x,%d := POP(%d)            ; pop return address
  if TAG($x) == TAG_STOP then MACHINE_STOP
  if TAG($x) != TAG_RET then FAULT(CONTROL_MISMATCH)
  $y,%d := POP(%d)            ; pop frame pointer
  %e := $y                    ; restore environment
  %c := $x                    ; jump to return address
```

```
Notes:
  Standard ABI convention is to leave the function return value on the
  top of the data stack. Multiple return values on the stack is possible,
  but not used in the standard ABI.

  The latest hardware revision optimizes the deallocation of the
  environment frame. If the environment has not been captured by LDF
  (directly or indirectly) then it can be immediately deallocated.
  Otherwise it is left for GC.


DUM - create empty environment frame

Synopsis: Prepare an empty frame;
          push it onto the environment chain;
Syntax:  DUM $n
Example: DUM 3      ; size of frame to allocate
Effect:
  $fp := ALLOC_FRAME($n)       ; create a new empty frame of size $n
  FRAME_PARENT($fp) := %e      ; set its parent frame
  FRAME_TAG($fp) := TAG_DUM    ; mark the frame as dummy
  %e := $fp                    ; set it as the new environment frame
  %c := %c+1
Notes:
  To be used with RAP to fill in the frame body.


RAP - recursive environment call function

Synopsis: pop a pointer to a CLOSURE cell off the data stack;
          the current environment frame pointer must point to an empty
            frame of size $n;
          fill the empty frame's body with $n values from the data stack;
          save the parent pointer of the current environment frame
             and return address to the control stack;
          set the current environment frame pointer to the environment
            frame pointer from the CLOSURE cell;
          jump to the code address from the CLOSURE cell;
Syntax:  RAP $n
Example: RAP 3      ; number of arguments to copy
Effect:
  $x,%s := POP(%s)            ; get and examine function closure
  if TAG($x) != TAG_CLOSURE then FAULT(TAG_MISMATCH)
  $f := CAR_CLOSURE($x)
  $fp := CDR_CLOSURE($x)
  if FRAME_TAG(%e) != TAG_DUM then FAULT(FRAME_MISMATCH)
  if FRAME_SIZE(%e) != $n then FAULT(FRAME_MISMATCH)
  if %e != $fp then FAULT(FRAME_MISMATCH)
  $i := $n-1
  while $i != -1 do           ; copy n values from the stack into the empty frame in reverse order
  begin
    $y,%s := POP(%s)
    FRAME_VALUE($fp,$i) := $y
    $i := $i-1
  end
  $ep := FRAME_PARENT(%e)
  %d := PUSH($ep,%d)                    ; save frame pointer
  %d := PUSH(SET_TAG(TAG_RET,%c+1),%d)  ; save return address
  FRAME_TAG($fp) := !TAG_DUM            ; mark the frame as normal
  %e := $fp                             ; establish new environment
  %c := $f                              ; jump to function


STOP - terminate co-processor execution

Synopsis: terminate co-processor execution and signal the main processor.
Syntax:  STOP
Effect:
  MACHINE_STOP
Notes:
  This instruction is no longer part of the standard ABI. The standard ABI
  calling convention is to use a TAG_STOP control stack entry. See RTN.
```

Tail call extensions

```
TSEL - tail-call conditional branch

Synopsis: pop an integer off the data stack;
          test if it is non-zero;
          jump to the true address or to the false address
Syntax:  TSEL $t $f
Example: TSEL 335 346  ; absolute instruction addresses
Effect:
  $x,%s := POP(%s)
  if TAG($x) != TAG_INT then FAULT(TAG_MISMATCH)
  if $x == 0 then
    %c := $f
  else
    %c := $t
Notes:
  This instruction is the same as SEL but it does not push a return address


TAP - tail-call function

Synopsis: pop a pointer to a CLOSURE cell off the data stack;
          allocate an environment frame of size $n;
          set the frame's parent to be the environment frame pointer
            from the CLOSURE cell;
          fill the frame's body with $n values from the data stack;
          set the current environment frame pointer to the new frame;
          jump to the code address from the CLOSURE cell;
Syntax:  TAP $n
Example: TAP 3      ; number of arguments to copy
Effect:
  $x,%s := POP(%s)            ; get and examine function closure
  if TAG($x) != TAG_CLOSURE then FAULT(TAG_MISMATCH)
  $f := CAR_CLOSURE($x)
  $e := CDR_CLOSURE($x)
  $fp := ALLOC_FRAME($n)      ; create a new frame for the call
  FRAME_PARENT($fp) := $e
  $i := $n-1
  while $i != -1 do            ; copy n values from the stack into the frame in reverse order
  begin
    $y,%s := POP(%s)
    FRAME_VALUE($fp,$i) := $y
    $i := $i-1
  end
  %e := $fp                   ; establish new environment
  %c := $f                    ; jump to function
Notes:
  This instruction is the same as AP but it does not push a return address

  The latest hardware revision optimizes the case where the environment
  frame has not been captured by LDF and the number of args $n in the
  call fit within the current frame. In this case it will overwrite the
  frame rather than allocating a fresh one.

TRAP - recursive environment tail-call function

Synopsis: pop a pointer to a CLOSURE cell off the data stack;
          the current environment frame pointer must point to an empty
            frame of size $n;
          fill the empty frame's body with $n values from the data stack;
          set the current environment frame pointer to the environment
            frame pointer from the CLOSURE cell;
          jump to the code address from the CLOSURE cell;
Syntax:  TRAP $n
Example: TRAP 3      ; number of arguments to copy
Effect:
  $x,%s := POP(%s)            ; get and examine function closure
  if TAG($x) != TAG_CLOSURE then FAULT(TAG_MISMATCH)
  $f := CAR_CLOSURE($x)
  $fp := CDR_CLOSURE($x)
  if FRAME_TAG(%e) != TAG_DUM then FAULT(FRAME_MISMATCH)
  if FRAME_SIZE(%e) != $n then FAULT(FRAME_MISMATCH)
  if %e != $fp then FAULT(FRAME_MISMATCH)
  $i := $n-1
  while $i != -1 do            ; copy n values from the stack into the empty frame in reverse order
  begin
    $y,%s := POP(%s)
    FRAME_VALUE($fp,$i) := $y
    $i := $i-1
  end
  FRAME_TAG($fp) := !TAG_DUM
  %e := $fp                   ; establish new environment
  %c := $f                    ; jump to function
Notes:
  This instruction is the same as RAP but it does not push a return address
```

Pascal extensions

```
ST - store to environment

Synopsis: pop a value from the data stack and store to the environment
Syntax:  ST $n $i
Example: ST 0 1
Effect:
  $fp := %e
  while $n > 0 do            ; follow chain of frames to get n'th frame
  begin
    $fp := FRAME_PARENT($fp)
    $n := $n-1
  end
  if FRAME_TAG($fp) == TAG_DUM then FAULT(FRAME_MISMATCH)
  $v,%s := POP(%s)           ; pop value from the data stack
  FRAME_VALUE($fp, $i) := $v ; modify i'th element of frame
  %c := %c+1
```

Debug extensions

```
DBUG - printf debugging

Synopsis: If tracing is enabled, suspend execution and raise a trace
          interrupt on the main processor. The main processor will read
          the value and resume co-processor execution. On resumption
          the value will be popped from the data stack. If tracing is not
          enabled the value is popped from the data stack and discarded.
Syntax:  DBUG
Effect:
  $x,%s := POP(%s)
  %c := %c+1
Notes:
  This is the formal effect on the state of the machine. It does
  also raise an interrupt but this has no effect on the machine state.

BRK - breakpoint debugging

Synopsis: If breakpoint debugging is enabled, suspend execution and raise
          a breakpoint interrupt on the main processor. The main processor
          may inspect the state of the co-processor and can resume
          execution. If breakpoint debugging is not enabled it has no
          effect.
```

```
Syntax:  BRK
Effect:
  %c := %c+1
```

Examples

Although we did not find any code from their higher level language, the instruction reference does include a couple examples. Note that it uses an assembly syntax with symbolic labels rather than absolute instruction addresses, so these programs cannot be executed directly.

```
The following GCC programs illustrate the behaviour of some of the
instructions:
```

local.gcc

```
Minimal example of creating and using a local variable.

  LDC  21
  LDF  body     ; load body
  AP   1        ; call body with 1 variable in a new frame
  RTN
body:
  LD   0 0      ; var x
  LD   0 0      ; var x
  ADD
  RTN
```

goto.gcc

```
Minimal example of mutual recursion. Note that the recursion in this
example does not terminate. It will fail with an out of memory error
due to the stack use.

  DUM  2        ; 2 top-level declarations
  LDF  go       ; declare function go
  LDF  to       ; declare function to
  LDF  main     ; main function
  RAP  2        ; load declarations into environment and run main
  RTN           ; final return
main:
  LDC  1
  LD   0 0      ; var go
  AP   1        ; call go(1)
  RTN
to:
  LD   0 0      ; var n
  LDC  1
  SUB
  LD   1 0      ; var go
  AP   1        ; call go(n-1)
  RTN
go:
  LD   0 0      ; var n
  LDC  1
  ADD
  LD   1 1      ; var to
  AP   1        ; call to(n+1)
  RTN
```

The processor/co-processor interface

In addition to the basic processor description, we found some documentation on the interface between the main processor and the co-processor.

```
On co-processor power up, it is in the halt state with heap and stacks
initialised to empty.
```

```
The primary processor uses the processor/co-processor interface (see
appendix 3.II) to initialise the code, heap, stacks and registers and
initiate execution.
```

(We don’t actually have appendix 3.II, but fortunately we don’t need it for our software simulation.)

```
When execution halts, the co-processor will raise an interrupt
(implementation defined) in the primary processor. The primary processor
may use the processor/co-processor interface to inspect the registers,
stack and heap. It may inspect special status registers to determine the
reason for execution halting: stop, fault, trace or breakpoint.
```

Essentially the main processor sets things up to run, being able to control the full state of the co-processor, then it tells the co-processor to run. When the co-processor halts then the main processor can look at the end state to get the program result. The state of the co-processor is preserved to use in a subsequent run.

```
When using the standard ABI, the primary processor may ensure execution
halts on the final return by installing an entry in the control stack (%d
register) with the TAG_STOP set.
```

The standard calling convention for functions/procedures apparently uses the `RTN` instruction. So for the main processor to call a function on the co-processor it needs a way to get it to halt after that function returns. The way it does that is by putting a special “stop” entry on the control stack so that when the function returns, instead of jumping to a return address the co-processor simply halts.

```
In the standard ABI, the initial entry point is at address 0. The arguments
and return are implementation defined.
```

This is just telling us that by convention the “main” function lives at the beginning of the program code. Whether the main function takes any arguments and what the return type is depend on the application. Fortunately we have the document that explains what these are for the Lambda-Man AI.

Though we never found the ABI document, we believe that in the standard ABI the function calling convention was to put function arguments in the environment (rather than on the stack), while the single function result value is returned on the top of the stack. The main processor will expect that all the entry points it invokes (including the main function) follow this calling convention.

Lambda-Man AI interface

```
For LAMBDAMAN, main is a function with two arguments:

 1. the initial state of the world, encoded as below
 2. undocumented

It returns a pair containing:

 1. the initial AI state
 2. the AI step function (closure)

The AI step function has two arguments:

 1. the current AI state
 2. the current state of the world, encoded as below

It returns a pair containing:

 1. the current AI state
 2. the move, encoded as below

The encoding of the AI state is private. The host should pass the current
AI state on each step but not otherwise inspect it.
```

So what this is telling us is that the AI is implemented as a function that looks at the current state of the game and returns the move it wants Lambda-Man to make. The AI does not have to be completely stateless however, it is able to make use of a private state that it can use and update on each step. It works by returning that new state, and the host (i.e. the game mechanics) is expected to pass that new state to the AI function in the next step. The initial AI state comes from the initial entry point (the ‘main’ function).

A slightly peculiar thing is that ‘main’ returns the AI step function, rather than that being another “well known” address. Apparently this is the convention that only 'main' has a well known address and any other entry points that the host needs must be returned by main. Note that when it says that it returns a function it doesn't mean a code address but a "CLOSURE cell". See the `LDF` and `AP` instructions for what we know about that.

```
The state of the world is encoded as follows:

A 4-tuple consisting of

1. The map;
2. the status of Lambda-Man;
3. the status of all the ghosts;
4. the status of fruit at the fruit location.

The map is encoded as a list of lists (row-major) representing the 2-d
grid. An enumeration represents the contents of each grid cell:

  * 0: Wall (`#`)
  * 1: Empty (`<space>`)
  * 2: Pill
  * 3: Power pill
  * 4: Fruit location
  * 5: Lambda-Man starting position
  * 6: Ghost starting position
```

For example, this map

```
###
#..
```

is encoded as

```
[ [ 0, 0, 0 ]
, [ 0, 2, 2 ]
]

Note that the map does reflect the current status of all the pills and
power pills. The map does not however reflect the current location of
Lambda-Man or the ghosts, nor the presence of fruit. These items are
represented separately from the map.

The Lambda-Man status is a 5-tuple consisting of:
  1. Lambda-Man's vitality;
  2. Lambda-Man's current location, as an (x,y) pair;
  3. Lambda-Man's current direction;
  4. Lambda-Man's remaining number of lives;
  5. Lambda-Man's current score.

Lambda-Man's vitality is a number which is a countdown to the expiry of
the active power pill, if any. It is 0 when no power pill is active.
  * 0: standard mode;
  * n > 0: power pill mode: the number of game ticks remaining while the
           power pill will be active

The status of all the ghosts is a list with the status for each ghost.
The list is in the order of the ghost number, so each ghost always appears
in the same location in the list.

The status for each ghost is a 3-tuple consisting of
  1. the ghost's vitality
  2. the ghost's current location, as an (x,y) pair
  3. the ghost's current direction

The Ghosts' vitality is an enumeration:
  * 0: standard;
  * 1: fright mode;
  * 2: invisible.

The Ghosts' and Lambda-Man's direction is an enumeration:
  * 0: up;
  * 1: right;
  * 2: down;
  * 3: left.

The status of the fruit is a number which is a countdown to the expiry of
the current fruit, if any.
  * 0: no fruit present;
  * n > 0: fruit present: the number of game ticks remaining while the
           fruit will be present.

Lambda-Man's move is a direction as encoded above.
```

This document refers to tuples and lists, but of course the only primitive data structure is a CONS cell—a pair. Though it is not clearly specified anywhere, we believe it uses the following encoding for tuples and lists.

Tuples are encoded as right nested pairs, four example a 4-tuple:

```
(a,b,c,d)  is encoded as     (a, (b, (c, d)))
           which is really   CONS a (CONS b (CONS c d))
```

The encoding for a list is also right nested CONS cells, but slightly different in the base case. The empty list is encoded as the integer 0.

```
     []  encoded as                          0
    [c]  encoded as                   CONS c 0
  [b,c]  encoded as           CONS b (CONS c 0)
[a,b,c]  encoded as   CONS a (CONS b (CONS c 0))
```

Example

```
To illustrate the interface, here is an AI that always goes down.

  DUM  2        ; 2 top-level declarations
  LDC  2        ; declare constant down
  LDF  step     ; declare function step
  LDF  init     ; init function
  RAP  2        ; load declarations into environment and run init
  RTN           ; final return
init:
  LDC  42
  LD   0 1      ; var step
  CONS
  RTN           ; return (42, step)
step:
  LD   0 0      ; var s
  LDC  1
  ADD
  LD   1 0      ; var down
  CONS
  RTN           ; return (s+1, down)
```

We can see from the code that it also (pointlessly) maintains an integer state (starting from 42) and increments it on every step.

Resource constraints

The Lambda-Man CPU had a clock speed of 3.072 MHz. Each processor cycle executed exactly one instruction, and each invocation of the Lambda-Man AI was allowed to run for one second before a move was made. In addition, the Lambda-Man AI was given one minute of time for initialization, before the game began.

From this it follows that each invocation of the AI, before a move, was allowed to run up to 3072 * 10^3 instructions. We assume that taking any more instructions than this would result in catastrophic failure.

The main RAM was—for its time—impressively large, and was able to hold 10 million CONS cells to be used by the Lambda-Man AI. The three stacks and the heap shared the RAM space. The data stack used one CONS cell for every two data values. The control stack used one CONS cell per entry. Each environment frame used one CONS cell plus half of the frame size (rounded down). Again, trying to allocate memory over this limit resulted in an error, which probably lead to catastrophic failure.

The program for Lambda-Man´s AI was limited to 1048576 instructions.

Errors

During the game, if an instruction causes an error then execution halts and no result is returned. In this case Lambda-Man's requested move will be the same as its previous move. The AI state remains unchanged, thus the next execution of the AI step function will be passed this AI state (but obviously the world is updated).

Submission procedure

You do not need to register to enter the competition, but you do need to fill in some details to submit a solution.

Submission is by filling in a web form with your team details, and a URL and SHA1 checksum of a .zip or .tar.gz file containing your solution. The judges will download the file containing your solution after the closing deadline and compare its SHA1 checksum with the one you provided.

- Submission form

We request (but do not strictly require) that you include in your .tar.gz/.zip various additional material:

- All the source code you wrote to help you prepare your solution.
- Any build scripts or other build files needed to run your code.
- Some directions for the judges to tell us how to run your code, including any necessary details of the environment.
- Any documentation/description of your solution that you wish to share with the judges, e.g. the approach you took, why your solution is awesome/funny/quirky/lame, any feedback about the problem or competition.

It is not essential that the judges be able to run your code, however they will try their best to run it for the top few winning entries and any entries under consideration for the judges' prize.

Privacy

Some people will want to show off their participation in the contest while some people will want to take part anonymously. We want to allow both.

The submission form therefore includes a number of optional fields for extra information about your team that we will publish if you choose to supply it; in particular the names of the members of your team and a URL for a team home page.

We do ask for contact details but we will not share these with anyone and only use them if we need to get in contact with you.

We will publish team names and scores/ranks.

We will not publish your full solution but we encourage you to do so yourself via your team home page after the competition ends. Please do not publish solutions to the lightning round until the full competition ends.

We may describe details of your solution or additional material during the final contest presentation at ICFP (if it is one of the winning entries or otherwise notable). If there is anything in your solution or additional material that you would like to remain confidential then please make it clear in the notes included in your submission.

Note that there are some practical constraints on anonymity and confidentially in the case that you happen to win: we need to say something about the winning teams and entries at the contest presentation and it is impossible to pay prize money anonymously. So while you are most welcome to participate while insisting on a high level of confidentiality, it may be impractical to award a prize in that case and the judges may decide to make the award to the next best entry.

Solution format

Your submission file (.zip or .tar.gz) should have the following format:

- Subdirectory `solution` that contains your solution file: `lambdaman.gcc`.
- Subdirectory `code` with the source code you wrote to help you prepare your solution and any auxiliary material that can be helpful to the judges to build your code. You should include a `README` file in this subdirectory with any documentation/description of your solution that you wish to share with the judges.