# 6.836 Embodied Intelligence–Research Assignment 2

Massachusetts Institute of Technology

Due: Thursday, March 6, 2003

We have seen a number of different architectures where computation is represented as a circuit. How can computations specified in this manner be run on a conventional computer architecture? In this research assignment you will write a compiler which takes a textual description of a network like this one from Friday 21st's lecture on REX:

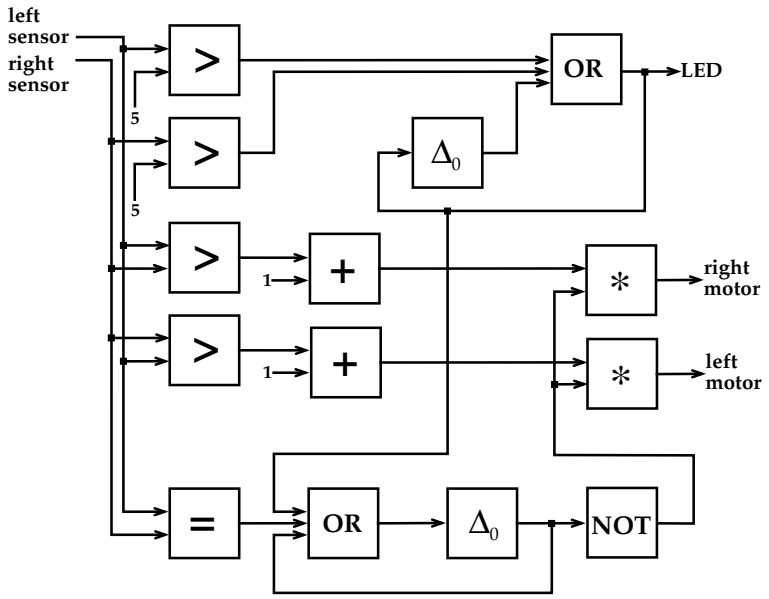


Figure 1: A REX architecture

and compiles it into a language that can run on a conventional machine. Your compiler will produce two procedures, one to initialize the state of the network, and one which gets called repeatedly to simulate one propagation of the network. The output language for your compiler can be any computer language you choose. For instance you might have your compiler produce C code that would effectively do the computation described by the original network diagram.

You won't actually need to compile or run your compiled code, just turn it in. You will also need to turn in your compiler code.

Now **don't panic**. We are giving you a fully functional compiler for a slightly different language and you can just modify that one if you choose (see the end of the assignment).

As an example of another sort of compiler consider the following neural network diagram (Figure 2) which is a possible solution to research assignment 1, question 1b.

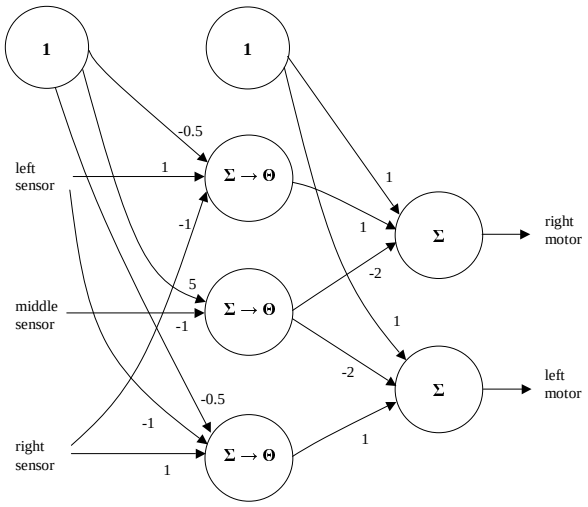


Figure 2: A neural network for obstacle avoidance. It has three inputs from the sensors and two outputs that connect to the motors.

This could be coded as:

```
;;; note that the order of the forms do not matter in this file
;;; things can refer to things not yet defined
;;; also the specification in connections if very loose,
;;; where it can be (name weight) or just name.  in the latter
;;; case a weight of 1.0 is assumed.


(list-inputs left right middle) (list-outputs leftmotor rightmotor)
(define-neuron rightmotor :sum c2 n1 (halter -2))
(define-neuron leftmotor :sum c2 n2 (halter -2))
(define-const c1 1) (define-const c2 1)

(define-neuron n1 :threshold (c1 -0.5) left (right -1))
(define-neuron n2 :threshold (c1 -0.5) (right 1) (left -1))
(define-neuron halter :threshold (c1 5) (middle -1))
```

and a suitable compiler might turn this source into the following C code:

```
void mainloop (double left, double right, double middle,
               double *leftmotor, double *rightmotor) {
  double n1;
  double n2;
  double halter;
  n1 = (-1.0*right + left + -0.5)>=0.0?1.0:0.0;
  n2 = (-1.0*left + right + -0.5)>=0.0?1.0:0.0;
  halter = (-1.0*middle + 5.0)>=0.0?1.0:0.0;
  *rightmotor = -2.0*halter + n1 + 1.0;
  *leftmotor = -2.0*halter + n2 + 1.0;
  return;}
```

**Problem 1.** (2 points) Consider again the REX diagram above. The boxes are labeled with simple computations that they do, like `+` or `*`. For such boxes the output wire is the sum or product of the numbers on the input wires. Some are logical operators like `<`, and in that case the output wire has a 1 or a 0 on it, for true or false respectively. At any time the the input numbers are propagated through the network, and once any race conditions have settled down the output numbers are latched

to the outputs. The boxes labeled things like $\Delta_n$ delay for one clock tick, until the next invocation of the network simulator, the propogation of their input to their output. On the first invocation such a box would output $n$. Backward loops in REX diagrams are only valid if they include a $\Delta$ box.

In lecture we noted that such networks could be unfolded into purely feed forward networks. The REX network above unfolds to:

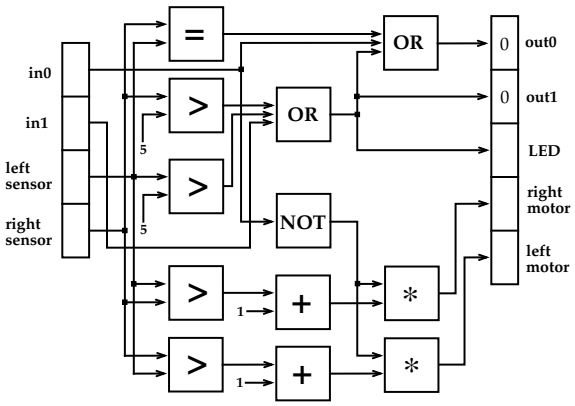


Figure 3: Unfolding of the REX network in figure 1.

Write two C procedures with the following prototypes for this network:

```
void initiliaze ();
void mainloop (int leftsense, int rightsense,
               int *leftmotor, int *rightmotor);
```

You may need to make some additional declarations. The procedure `initialize` will be called only once and will initialize any data structures (e.g., any $n$'s in $\Delta_n$'s). Then the procedure `mainloop` would get called repeatedly to carry out one time step of the network for every invocation.

**Problem 2.** (2 points) Design a specification language that can be used to specify REX networks. Describe the syntax of your language and write down a specification for the original REX network shown above.

**Problem 3.** (2 points) Write the first part of a compiler which reads in a REX specification in your language, and then turns it into the unfolded network–this is done internally, so there is no need to have it draw the diagram. Explain how your algorithm works in words.

**Problem 4.** (2 points) Write the code generation phase of your compiler and show the output code it generates. Also turn in your compiler code.

**Problem 5.** (2 points) A REX solution for problem 3b of RA1 is shown in figure 4. Run your compiler on this diagram and hand in the compiled code.

### An existing compiler

You can find an existing compiler for the neural network example above at **http://www.ai.mit.edu/courses/6.836/handouts/handouts.html**. You can use this compiler as a basis for this research assignment. Or you can choose to write you own from scratch.

This compiler is written in Emacs Lisp. All you need to use it is Emacs. The instructions of how to load it are at the head of `compiler.el`. Once it is loaded simply visit a file named `foo.robot` and type C-x C-g and an output file `foo.c` will be produced.

Most of the compiler is language independent. There are all the utilities you need to manipulate files. There are also syntax checking utilities, graph loop detection utilities, and a utility that orders

a feedforward graph so that code can be generated that only needs backward references.

---

**Project Ideas**

Projects related to this problem set can be writing a compiler for translating a given architecture to:

- reconfigurable hardware such as a Field Programmable Gate Array (FPGA)
- assembler/C code for a specific microcontroller.

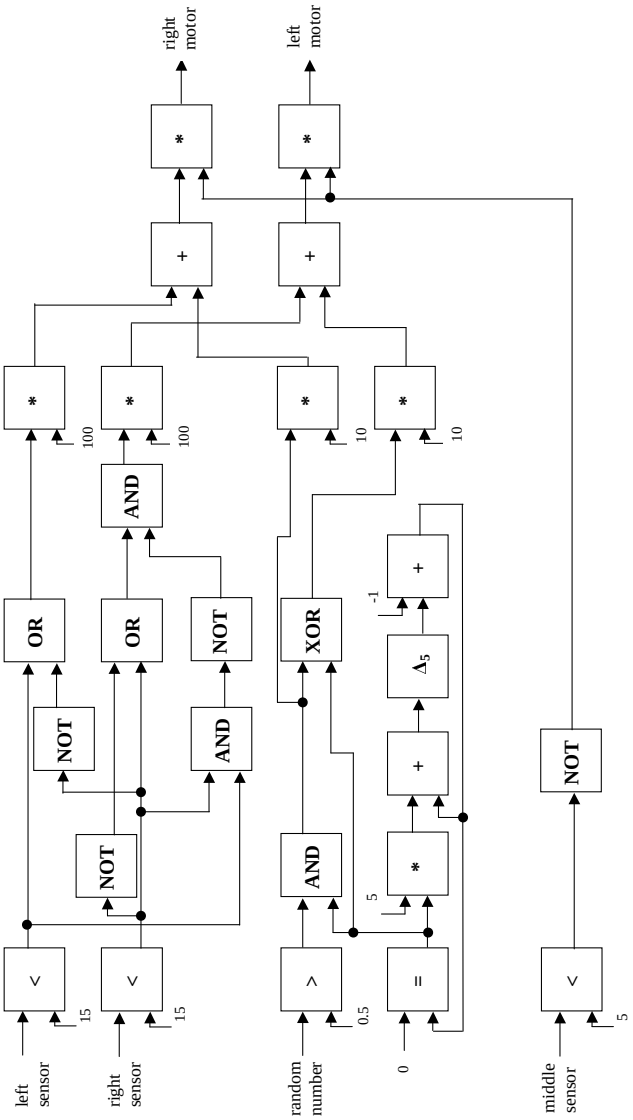


Figure 4: Controller implemented in REX.